UNIVERSITY OF CALIFORNIA
Santa Barbara

# Tools and Techniques for the Design and Systematic Analysis of Real-Time Systems

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Paul Zachary Kolano

Committee in charge:

Professor Richard Kemmerer, Chairperson

Professor Tevfik Bultan

Professor Ambuj Singh

December 1999

The dissertation of Paul Zachary Kolano is approved.

---

Tevfik Bultan

---

Ambuj K. Singh

---

Richard A. Kemmerer, Committee Chairperson

June 1999

*For Mom and Dad*

# Acknowledgments

First of all, thanks to Santa Barbara, which besides the floods, earthquakes, fires, uncontrolled rent increases, bomb threats, exorbitant gas prices, hyperactive mold formations, oil platforms, tornadoes, volcanoes, anacondas, and meteor strikes (ok, ok, maybe not those last four), was the perfect place to live.

Thanks to UCSB for its beautiful campus and stimulating environment (and also for those stylish gym uniforms that remind us all that we're never more than a step away from high school).

Thanks to various friends for taking my mind off work with fun stuff like beach volleyball, movies, indoor volleyball, skiing, grass volleyball, Vegas trips, 2x2 volleyball, pool, 3x3 volleyball, basketball, 4x4 volleyball... (well, you get the picture).

Thanks also to my family for their support. Although they forgot my phone number, misplaced my address, and deleted my e-mails, I still always knew they were there. (ok, ok, I guess they did call and write sometimes and they did all make it to graduation).

Thanks to the National Science Foundation for partially supporting my research under Grant No. CCR-9204249. Thanks to my committee members, Tevfik Bultan, Ambuj Singh, and former member Laurie Dillon for all of their useful suggestions (and for straining their eyes on the 362 pages of my draft). Thanks also to Dino Mandrioli for his enthusiasm in my work and for the many interesting and useful discussions and suggestions he provided.

Finally, thanks most of all to Richard Kemmerer for his guidance, support, and friendship (and especially for the never-ending stream of wisecracks and stories that made working much more fun). Although I didn't know what to expect from an advisor in the beginning, I now know I couldn't have found a better one.

# Vita

| | |
|---|---|
| 2/23/73 | Born in Winfield, Illinois |
| 1994 | B.S. with honors in Computer Science,<br>University of Illinois, Urbana-Champaign |
| 1994-95 | State Research Assistant Fellowship |
| 1995-99 | Research Assistant, Department of Computer Science,<br>University of California, Santa Barbara |

# Publications


P.Z. Kolano, Z. Dang, and R.A. Kemmerer. "The Design and Analysis of Real-Time Systems Using the ASTRAL Software Development Environment". *Annals of Software Engineering*, volume 7, 1999. Bussum, The Netherlands: Baltzer Science Publishers.

P.Z. Kolano. "Proof Assistance for Real-Time Systems Using an Interactive Theorem Prover". *Proc. of the 5th Int. AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, Bamberg, Germany, May 1999. Berlin, Germany: Springer-Verlag, pp. 315-333.

R.A. Kemmerer and P.Z. Kolano. "Formally Specifying and Verifying Real-Time Systems". *Proc. of the 1st IEEE Int. Conf. on Formal Engineering Methods*, Hiroshima, Japan, Nov. 1997. Los Alamitos, CA: IEEE Computer Society Press, pp. 112-120.

K.E. Schauser, C.J. Scheiman, J.M. Ferguson, and P.Z. Kolano. "Exploiting the Capabilities of Communications Co-Processors". *Proc. of the 10th Int. Parallel Processing Symp.*, Honolulu, HI, Apr. 1996. Los Alamitos, CA: IEEE Computer Society Press, pp. 109-115.

# Abstract

Tools and Techniques for the Design and Systematic Analysis of Real-Time Systems

Paul Zachary Kolano

As technology progresses and computers become smaller, cheaper, and more powerful, they are increasingly relied on to guarantee the safety of human life and the environment. In most cases, it is not enough to merely provide such safety mechanisms, but is also critical to assure that they will be activated in time to prevent disasters. These *real-time systems* are found in both large-scale projects with highly visible consequences such as nuclear reactors and air traffic control systems as well as in consumer goods such as automobiles and smoke detectors. As more and more reliance is placed on real-time computing systems to perform critical and everyday functions, the need for formal methods to guarantee the correctness of these systems becomes crucial.

Given the time and complexity of applying formal methods, however, there is not merely a need for these methods, but also a need for assurance that they will be used. To provide this assurance and make the design and analysis of real-time systems more practical, it is necessary to provide a real-time specification language that has mechanisms for specifying large and complex systems as well as comprehensive tool and methodological support for design and analysis. In particular, to reduce the technical expertise required to reason about real-time systems, there is a need for systematic analysis guidance that can be consulted by the user to determine which analysis step should be performed next, how it can be performed efficiently using the appropriate tools and techniques, and how the results of different approaches can be combined.

Existing real-time specification languages do not provide the necessary level of support. In this dissertation, an existing real-time specification language is augmented to meet the desired requirements, which also provides general techniques through which other languages may be similarly augmented. A number of language issues are examined and resolved including composition support and the definition of a new parallel refinement mechanism. Systematic analysis guidance is provided for model checker test case generation, proof sketch construction, and theorem prover utilization based on a variety of classification schemes. Finally, an integrated software development environment is presented, which was implemented to include comprehensive support for design, analysis, and reuse.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

With the arrival of the first programmable computers came the less heralded arrival of the first programming errors. The first computers were huge, expensive machines that were limited to retrieving input from the user, computing a result based on some algorithm, and then reporting this result back to the user. In these early systems, errors took the form of results that did not match the expected output of the algorithm for the given data. To find such errors, programs were tested using a small fraction of the possible input cases. If the program gave the correct results for all of the test cases, then it was hoped that it would give the correct results for all input data. For most applications such as the analysis of scientific data, testing provided an adequate level of assurance that a program behaved as desired. In these applications, errors that were not uncovered during testing were a nuisance, but did not have catastrophic consequences. When computers were used in applications where undiscovered errors could have serious repercussions on human life and the environment, however, such as computing flight path data for manned rockets, testing was no longer adequate. For these applications, a higher degree of certainty was needed to guarantee that no unforeseen consequences would occur. To meet this need, *formal methods* such as Hoare logics were proposed for modeling programs and proving that they met a set of functional requirements using well-defined mathematical techniques.

Although the use of these methods resulted in greater assurance, proofs of program correctness were nontrivial to perform and could add significant delays to development time. In addition, they required personnel with a high level of mathematical maturity and expertise to perform the proofs properly. Given the high cost of these methods, they were only feasible for projects in which program correctness was critical to preventing disastrous consequences.

As technology progressed and computers became smaller, cheaper, and more powerful, they were used in an increasing variety of applications. Whereas initially they were stand-alone machines that interacted primarily with a human user, they began to be used as components of larger systems in which they also interacted with other devices and the external environment. As was the case for

stand-alone computers, there were some applications of these systems in which it was critical to guarantee correct behavior. The formal models developed for the program level, however, were not adequate for specifying the behavior of multiple components and the interactions between them. They also were not adequate for expressing all of the properties that might be desired of a system. For example, in systems of interacting components it became vital to express the notion of progress to guarantee that components did not become deadlocked waiting for each other. Thus, there was a need for formal models that dealt with a higher level of abstraction than the program level and could express corresponding properties. To meet this need, formal methods such as process algebras and temporal logics were proposed, which allowed the specification of multiple communicating entities and the notion of progress.

Although temporal logics allow the notion of progress to be specified, for some applications "eventually" is not enough of a guarantee. In particular, as computers are given control of systems whose purpose is to guarantee the safety of human life in an unpredictable environment, it becomes critical to assure that the safety mechanisms of these systems are activated in time to prevent disasters. For example, in a nuclear reactor not only is it necessary for the controller to insert the control rods when meltdown conditions are imminent, but they must also be inserted within a precise amount of time or else a meltdown will occur.

These *real-time systems*, however, are not just found in large-scale projects with highly visible consequences such as nuclear reactors and air traffic control systems. With the availability and low cost of computing devices, these systems are also becoming commonplace in consumer goods such as automobiles and smoke detectors. Even though these items are common and used everyday, the malfunction of their computerized controllers can cause loss of human life. For example, traction control systems in automobiles allow the driver to retain more control of a vehicle on various surfaces. These systems achieve this by various techniques such as monitoring wheel speeds and selectively applying the brake to a specific wheel to prevent a spinout. If the controller does not react fast enough to the changes in wheel speed, the system may cause a spinout instead of preventing one by reacting to conditions that no longer exist.

As more and more reliance is placed on real-time computing systems to perform critical and everyday functions, the need for formal methods to guarantee the correctness of these systems becomes crucial. Given the time and complexity of applying formal methods, however, there is not merely a need for these methods, but also a need for assurance that they will be used. To provide this assurance and

make the design and analysis of real-time systems more practical, it is necessary to provide a real-time specification language that has:

1. A formal and rigorous definition

In order to formally prove that the behavior of a system meets its critical requirements, the language used to specify the system and the requirements must be formally defined. It is not enough, however, to merely have a formal definition. It is also necessary for the definition to have been rigorously reasoned about and tested by applying it to a large variety of systems to assure that any requirements that can be proved in the language actually hold in the model and that any requirements that hold can be proved.

2. Simple facilities for simple systems

A language should allow simple systems to be expressed in a straightforward and intuitive manner. If even simple systems require the use of unintelligible operators and/or complex constructs, it is unlikely that the language will ever be used to specify larger systems.

3. Complex facilities for complex systems

A language must be able to specify a system at a level of detail as close to the implementation level as possible. This means that a language should not be limited to basic operations just to provide fully automatic verification, but should allow any reasonable behavior to be specified. The greater the difference between the level of detail that can be specified and the level that can be implemented, the less assurance that is actually gained about an implementation by formal proofs. Given the complexity of large systems, it is also necessary to provide mechanisms such as modularity, refinement, and composition that allow a complex system to be specified as the aggregation of smaller or simpler components that are easier to specify and verify.

4. Comprehensive and integrated tool support

A language must be supported by a set of tools that allows it to be used correctly and effectively. These tools must provide comprehensive support for design, analysis, and reuse. During design, they should prevent static errors, such as syntax and typing errors. During analysis, they should prevent flawed reasoning and provide as much automated assistance as possible. During maintenance and reuse, they should track specification changes and assist in compositional transformations. These tools should be integrated so that information resulting from one tool can be utilized by another.

5. Comprehensive and systematic analysis guidance

The behavior of real-time systems is complex and even small systems can be difficult to reason about. Analysis tools provide assistance for such reasoning, but are only effective when used properly. Learning to use these tools properly, however, can be an arduous and time-consuming process of trial and error. Thus, it is crucial that a specification language be provided with a systematic analysis methodology that can guide the user precisely as to what step should be performed next during the analysis process. This includes guidance as to how each tool and technique can be used most effectively and how the results of different approaches can be used to complement each other.

Existing real-time specification languages fail in one or more of these areas. Some languages are intuitive for smaller systems, but become unreasonably complex when applied to larger systems. A number of languages are supported by fully automated verification tools such as model checkers, but suffer from limited expressiveness. Other languages are supported by semi-automated tools such as symbolic executors and mechanical theorem provers, but do not provide guidance as to how these tools can be used most effectively. Languages that are supported by both fully automated tools and semi-automated tools usually do not discuss how the results of these tools can be integrated together or be used to complement each other. Very few languages provide analysis guidance and those that do only provide it for certain phases of analysis, leaving other portions entirely up to the user. In the end, no existing specification language meets all of the above requirements.

This dissertation discusses how an existing real-time specification language was augmented to meet the above requirements. ASTRAL is a formal specification language for real-time systems that has been formally defined. ASTRAL is based on state transition systems and first-order logic, which makes it very expressive and also allows simple systems to be described in a simple and intuitive manner. In addition, it has a modular proof system and has facilities for composition and refinement, which are crucial for designing large and complex systems. Thus, the original ASTRAL definition partially met the first three requirements. These requirements were not fully met, however, due to a number of errors and omissions in that definition. The ASTRAL semantics and proof obligations suffered from a number of soundness and completeness problems. In addition, the composition capabilities were not completely described and had no tool support; thus, they were unusable given the number of complex transformations they required. The refinement mechanism was also incompletely described and its expressiveness was limited. ASTRAL had only rudimentary tool support for writing specifications and no tool or methodological support for analysis.

The problems in the ASTRAL definition were correctable, however, and relatively minor given the expressiveness and usability of the ASTRAL language. The lack of tool and methodological support also provided an ideal opportunity to develop a comprehensive set of tools and techniques for the design and systematic analysis of real-time systems that would meet all of the above criteria.

The remainder of this dissertation is organized as follows. Chapters two, three, and four provide additional background for the research. Chapter two presents the set of testbed systems that were specified and verified in ASTRAL to determine the tools and techniques that are most useful during design and analysis. This chapter also gives an overview of the ASTRAL language. Chapter three discusses other existing real-time specification languages and their relation to ASTRAL. It also provides a basis for understanding why existing languages do not adequately fulfill the requirements above. Chapter four discusses the difficulties involved in verifying real-time systems, some existing verification approaches, and the rationale for why existing techniques are insufficient.

Chapters five, six, and seven present the tools and techniques developed for designing real-time systems in ASTRAL. Chapter five describes the design portions of the ASTRAL software development environment, which is an integrated set of design and analysis tools for the ASTRAL language. This includes a discussion of how the composition capabilities of ASTRAL were automated. Chapter six discusses the problems in the original ASTRAL semantics and proof obligations and presents the revised and expanded versions. Chapter seven discusses the problems in the original ASTRAL sequential refinement mechanism and proposes a new parallel mechanism that increases the expressiveness of the language.

Chapters eight, nine, and ten describe the tools and techniques developed for analyzing real-time specifications in ASTRAL. Chapter eight presents the set of classification schemes that were developed as the basis for the systematic analysis methodology. It also describes querying mechanisms that can be used to obtain the classification information as well as other information required during analysis. Chapter nine presents a methodology for systematically determining model checker test cases to guarantee that a property will be adequately tested. In addition, this chapter presents a methodology for systematically proving the requirements of a system by hand based on process and property classifications. Chapter ten discusses how the proof of a property can be carried out within a mechanical theorem prover in a manner similar to the proof by hand.

Finally, chapter eleven discusses conclusions drawn from this work and directions for future research.

# Chapter 2

# Testbed Description and ASTRAL Overview

This chapter describes the set of testbed systems that is used throughout the remaining chapters. In addition, it gives a brief overview of the ASTRAL real-time specification language.

## 2.1. Testbed Systems

In order to determine the tools and techniques that are most useful during design and analysis, a set of testbed systems was developed. These systems consist of a variety of different process and property types. Each system was specified in ASTRAL and their proofs were performed to detect the proof patterns that occurred most often. The specifications were then examined in an attempt to find classification schemes that could predict which proof pattern was most applicable to a given situation. The classification schemes then formed the basis for the systematic analysis methodology. These systems demonstrate the flexibility and expressiveness of ASTRAL, which can be used to specify widely varying systems from distributed mutual exclusion protocols to phone switching systems to production facilities. The full ASTRAL specifications of these systems are located in appendix A.

### 2.1.1. Bakery Algorithm

The bakery algorithm specification describes the distributed mutual exclusion algorithm of [Lam 74], which is shown in figure 2.1.1. Each of n processes is involved in a computation that contains a *critical region*, which only a single process may enter at any given time. The process that is to enter its critical region determines if it is permissible to do so by checking the status of its "siblings". The critical requirement of the system is that only one process will be in its critical region at any given time.

### 2.1.2. Cruise Control

The cruise control system is based on the description in [WM 85]. The system is an automatic throttle control system for an automobile. The driver engages the system at a desired speed and the controller continuously adjusts the throttle according to fluctuations in terrain, etc. that have an effect on the speed of the vehicle. The system consists of four processes. A tire sensor keeps track of the number of times the wheels have rotated. A speedometer computes the current speed of the vehicle based on the number of wheel rotations and the sample rate. An accelerometer computes the current acceleration of the vehicle based on the speed and the sample rate. The controller itself consists of the cruise interface, the accelerator and brake pedals, and the throttle control. When the controller is maintaining speed, the actual throttle will always be the higher of the throttle that the driver is demanding with the accelerator pedal and the throttle that the controller is attempting to set. The cruise control must be disabled as quickly as possible when the brake pedal is applied.

```
boolean choosing[n] = {false, ..., false};
integer number[n] = {0, ..., 0};
while (true) {
    choosing[my_process_id] = true;
    number[my_process_id] = maximum(number[1], ..., number[n]) + 1;
    choosing[my_process_id] = false;
    for (integer j = 1; j ≤ n; j++) {
        while (choosing[j])
            {}
        while (number[j] ≠ 0 &&
                number[j] ≤ number[my_process_id] &&
                (number[j] ≠ number[my_process_id] || j < my_process_id))
            {}
    }
    critical_section();
    number[my_process_id] = 0;
    noncritical_section();
}
```

Figure 2.1.1: Bakery Algorithm

### 2.1.3. Elevator Control System

The elevator system was adapted from a description in [FF 84] as shown in figure 2.1.3. An n story building is serviced by the elevator. A panel of n buttons is located inside the elevator car to request that the elevator move to a given floor. Each floor in the building also has a button panel, which has an up and a down button to request that the elevator stop at the floor and move in the corresponding direction. The elevator must service all the requests in one direction before it can move in the opposite direction. When the elevator arrives at a floor en route to another destination and no request

has been made inside the elevator for that floor, nor has a request been made at that floor's button panel for movement in the same direction, the elevator continues on to its next destination without stopping or opening the door. If such a request has been made, however, then the elevator stops and opens the door. The door is always opened for a duration of t_stop at which point it closes. When the elevator arrives at a floor that is the last request in its direction of movement, the door opens and then its behavior depends on the situation in the building. If the button panel at the elevator's location has requested movement in the same direction, the user must get in and push the desired floor on the elevator's button panel before the door has finished closing. Otherwise, the elevator is free to move in the opposite direction to service another request, if one exists. The critical timing requirement of the elevator system is that the elevator must service any request within t_service_request time of when the button was pushed.

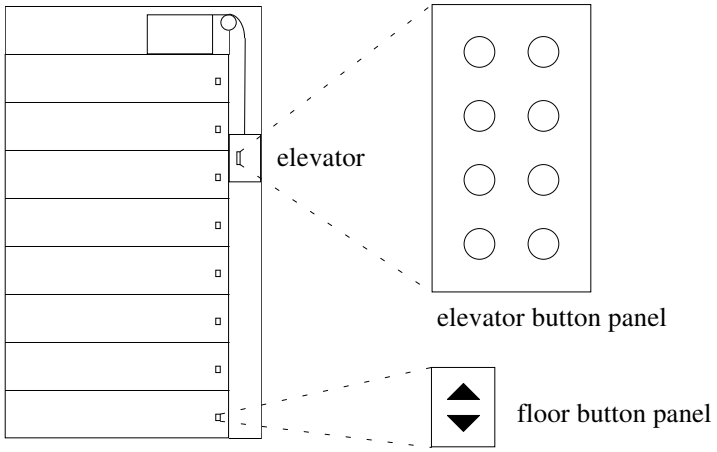



Figure 2.1.3: Elevator Control System

### 2.1.4. Olympic Boxing Scoring System

In Olympic boxing, each bout consists of three 3-minute rounds, with a 1-minute interval between rounds. In 1992, the Olympic Committee decided to use an electronic scoring system for boxing. The following is a description of the system taken from the official 1996 Olympic web site [Oly 96].

> "For the first time in Olympic boxing competition, an electronic scoring system was used at the 1992 Olympic Games in Barcelona, Spain. Under electronic scoring, five working judges are positioned at ringside with a desk-mounted keypad at each judge's position.

> The keypads, each of which are linked to the mainframe computer at the jury table, feature four buttons—red and blue scoring and red and blue warning buttons.
>
> During the course of the bout, judges record scoring blows for each competitor on their keypad. In order for a blow to be recorded by the computer as part of the official (or combined/accepted score), three of five judges must press the same colored button within a one-second interval. The one-second interval begins when the first judge records a blow.
>
> Scores are reported in terms of number of blows recognized by a majority of judges over the course of the three rounds combined."

The specification in appendix A only considers the scoring portion of the system (i.e. warnings are not included).

### 2.1.5. Phone System

The phone system description and specification was taken from [CGK 97]. The system consists of a set of phones that need various services (e.g. getting a dial tone, processing digits entered into the phone, making a connection to the requested phone, etc.) as well as a set of central control centers that perform the services. Each control center is responsible for the phones belonging to its area, and it is provided with all the functionality needed to set up a local call. Control centers are also intended to deal with long distance calls (i.e. calls to other areas). Calls to outside areas are modeled by exported variables (i.e. the data is sent to the external environment), while calls from an outside area are modeled as exported transitions (i.e. they are the information provided in the parameters of a call to an exported transition from the external environment). The example is a simplification of a real phone system. Every local phone number is seven digits long, area codes are three digits long, a customer can be connected to at most one other phone (either local or in another area), and ongoing calls cannot be interrupted. The main requirement of the system is that phones will be given a dial tone within two seconds.

### 2.1.6. Production Cell

The production cell specification is based on the description in [LL 95] and is shown in figure 2.1.6. The purpose of the production cell is to forge metal blanks and convey them to a stockpile. Each blank is added to a feed belt that is continuously moving until a light sensor at the end of the belt detects a blank. The feed belt then stops. When the elevating rotary table at the end of the belt is in the proper position, the feed belt moves the blank onto the table and continues operation until another

blank reaches the sensor. Once a blank is on the table, the table elevates to the level of a robot arm and rotates into the appropriate position. The robot consists of two stationary arms, each of which is positioned at a different vertical position. The robot assembly can be rotated and each arm can extend and retract to pickup and drop metal plates. The first arm picks up blanks from the table and rotates to deliver them to a press. The press forges the blank into a plate, which is then picked up by the second robot arm and delivered to a deposit belt. Like the feed belt, the deposit belt is continuously moving until a light sensor at the end of the belt detects a plate. The belt then stops and waits until the plate is removed by a crane, which transports the plate to a stockpile. The main requirements of the system are that the robot arms do not collide with any of the other pieces of equipment, and that the blanks and plates are only dropped at the proper places and never onto another blank or plate.

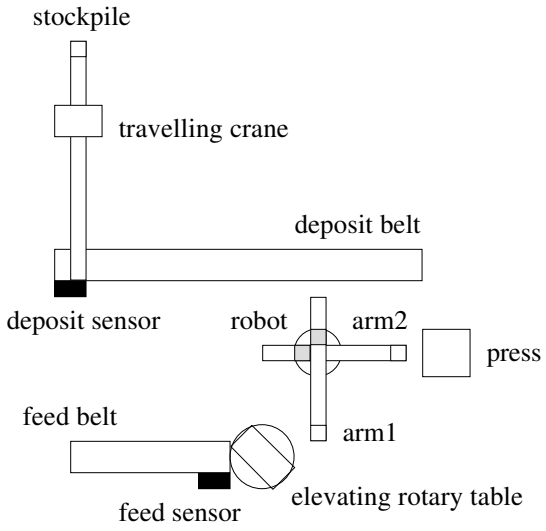



Figure 2.1.6: Production Cell

### 2.1.7. Railroad Crossing

The railroad crossing system is based on the description in [HL 94]. The system consists of a set of railroad tracks that intersect a street by which cars may cross the tracks. A gate is located at the crossing to prevent cars from crossing the tracks when a train is near. A sensor on each track detects the arrival of trains on that track. The region between the sensors and the crossing exit is denoted by R and the crossing, which is a subinterval of R, is denoted by I. Figure 2.1.7 illustrates the railroad crossing with two train tracks. The critical requirements of the system are that whenever a train is in

I, the gate must be down and when no train has been in R for a reasonable length of time, the gate must be up.

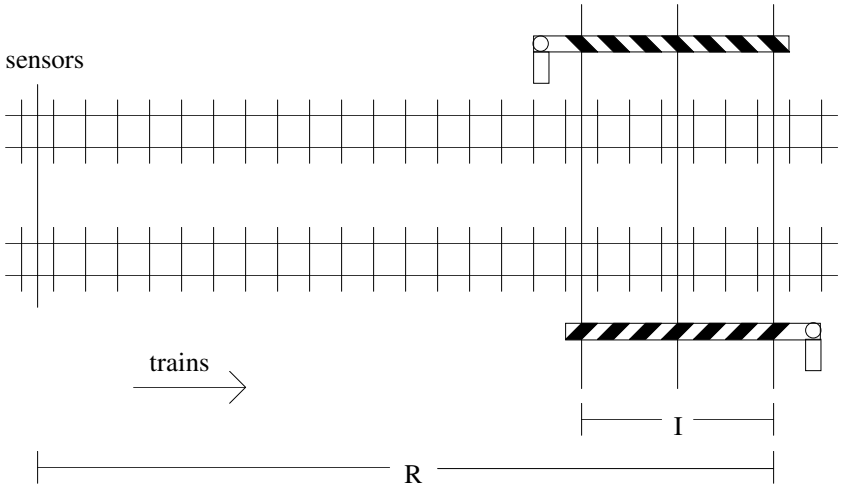



Figure 2.1.7: Railroad Crossing

### 2.1.8. Stoplight Control System

The stoplight specification is for a four-way intersection in which each direction has its own turn lane. This example was adapted from a stoplight system described in [FF 84]. Figure 2.1.8 depicts the intersection of the system. Each direction has two signals. The “arrow” signal is the signal for the left turn lane and the “circle” signal is the signal for going straight and for right turns. The signals can change independently; thus, for example, if the circle of direction one is green, it is not necessarily the case that the circle opposite of direction one is green. The controller can therefore act “intelligently”, such as changing both the arrow and the circle of a direction to green at the same time if no car is in the turn lane of the opposite direction.

One direction is designated as the “main direction” and is assumed to be the direction with the most traffic flowing. The controller will keep the circle and arrow of the main direction green when no cars are waiting at the intersection.

The system makes no assumptions about the behavior of cars, thus a car can sit on a sensor forever and the controller will still cycle the directions with the green appropriately. A direction will stay green for at least min_green time and yellow for at least min_yellow, and a car will wait no longer than max_wait for a green.

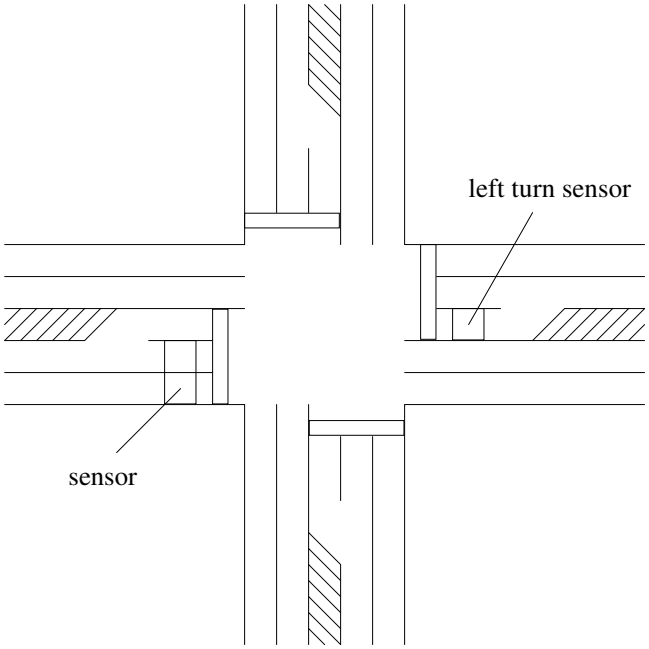



Figure 2.1.8: Stoplight Control System

## 2.2. ASTRAL Overview

A number of papers and technical reports are available that discuss the original ASTRAL language in full detail. [CGK 97] gives a complete overview of ASTRAL using a telephony example throughout. [CKM 94] and [CKM 95] present the intra-level and inter-level proof obligations, respectively, used to formally verify ASTRAL specifications. [CK 93] discusses how individual ASTRAL specifications can be composed into specifications of larger and more complex systems. Finally, [CSK 94] presents the formal semantics of the ASTRAL language. The following description of ASTRAL is based on the description in [CGK 97].

In ASTRAL, a real-time system is described as a collection of state machine specifications, where each specification represents a process type of which there may be multiple statically generated instances. Each process instance in the system executes concurrently and asynchronously with all the other process instances. Additionally, a *global specification* contains declarations for types and constants that are shared among more than one process type, as well as assumptions about the global environment and critical requirements for the whole system. Figure 2.2 presents the syntactic structure for an ASTRAL specification. The grammars for the different sections of an ASTRAL specification are given in appendix E.

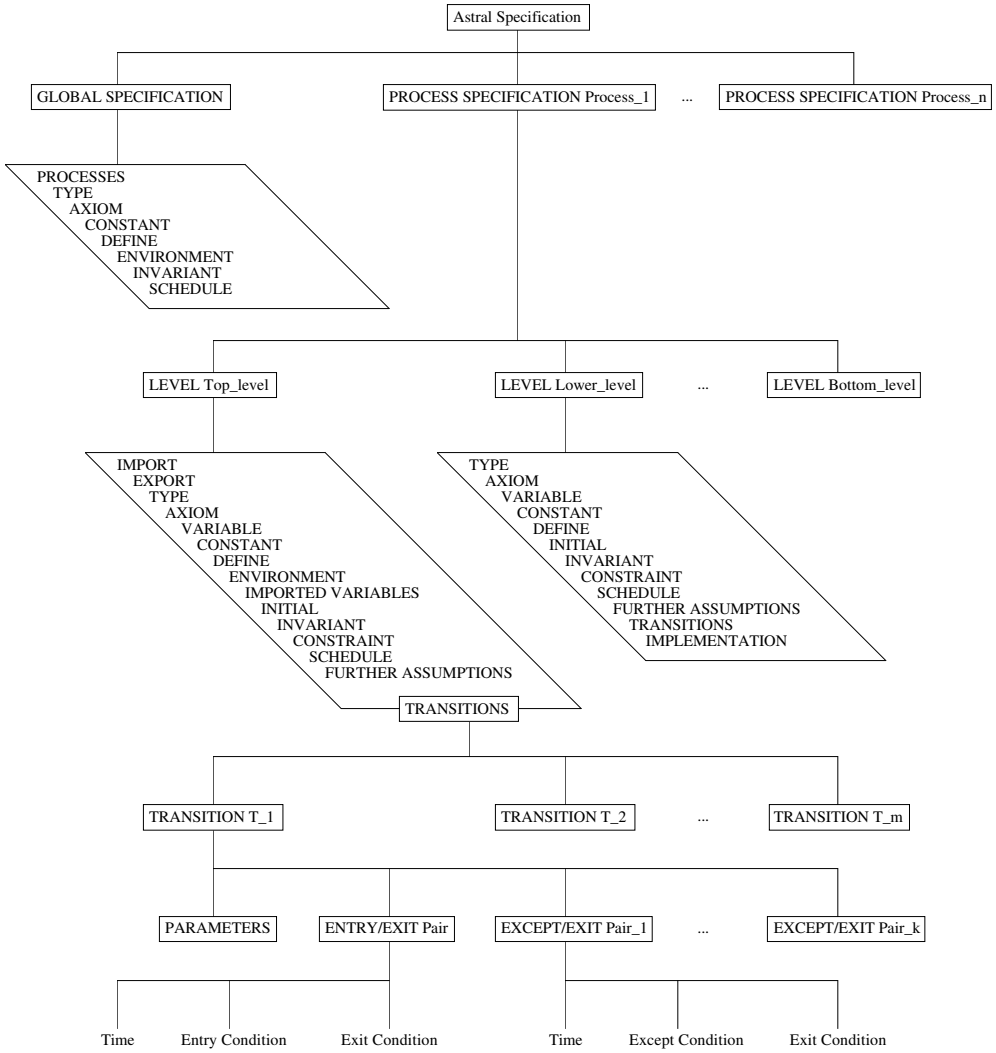



Figure 2.2: The ASTRAL hierarchy

An ASTRAL *process specification* consists of a sequence of *levels*. Each level is an abstract data type view of the process being specified. The first ("top level") view is a very abstract model of what constitutes the process (types, constants, variables), what the process does (state transitions), and the critical requirements the process must meet (invariants and schedules). Lower levels are increasingly more detailed with the lowest level corresponding closely to high-level code.

The process being specified is thought of as being in various *states*, with one state differentiated from another by the values of its *state variables*, which can be changed only by means of *state transitions*. Transitions are described in terms of entry and exit assertions by using an extension of first-order predicate calculus. Transition *entry assertions* describe the constraints that state variables must

satisfy in order for the transition to fire, while *exit assertions* describe the constraints that are fulfilled by state variables after the transition has fired. An explicit non-null duration is associated with each transition. Transitions are executed as soon as they are enabled assuming no other transition for that process instance is executing.

Every process can export both state variables and transitions; as a consequence, the former are readable by other processes while the latter are executable from the external environment. Processes communicate by broadcasting the values of exported variables and the start and end times of exported transitions.

In addition to specifying system state (through process variables and constants) and system evolution (through transitions), an ASTRAL specification also defines system critical requirements and assumptions on the behavior of the environment that interacts with the system. The behavior of the environment is expressed by means of *environment clauses*, which describe assumptions about the pattern of invocation of external transitions. Critical requirements are expressed by means of *invariants* and *schedules*. Invariants represent requirements that must hold in every state reachable from the initial state, no matter what the behavior of the external environment is, while schedules represent additional properties that must be satisfied provided that the external environment behaves as assumed.

The computational model for ASTRAL is based on nondeterministic state machines and assumes maximal parallelism, noninterruptable and nonoverlapping transitions in a single process instance, and implicit one-to-many (multi-cast) message passing communication. In what follows, the main features of ASTRAL are briefly introduced using the specification of the elevator control system of section 2.1.3. The full specification of this system is given in appendix A.

### 2.2.1. Processes

The elevator system consists of three process type specifications: Elevator, Elevator_Button_Panel, and Floor_Button_Panel. The processes declaration,

```
PROCESSES
    the_elevator: Elevator,
    the_elevator_buttons: Elevator_Button_Panel,
    the_floor_buttons: array [1..n_floors] of Floor_Button_Panel,
```

which occurs in the global specification, declares that there is one instance of the Elevator process and one of the Elevator_Button_Panel process. In addition, there are n_floors static instances of the

Floor_Button_Panel process type. Each of these is accessed as "the_floor_buttons[i]", where i is in the range 1 to n_floors.

### 2.2.2. Types

ASTRAL is a strongly typed language. Integer, Real, Boolean, ID, and Time are the only primitive types. All other simple and constructed types used in a process specification must either be declared in the type section of that specification or must be declared in the global specification and explicitly imported by the process specification.

The type ID is one of the primitive types of ASTRAL. Every instance of a process type has a unique id. An instance can refer to its own id by using *Self*. There is also an ASTRAL specification function *idtype(i)*, which returns the type of the process that is associated with the id i.

The global specification has three type declarations. The first two,

pos_integer: TYPEDEF i: integer (i > 0),
pos_real: TYPEDEF r: real (r > 0),

declare pos_integer and pos_real to be subtypes of the integers and reals, respectively, that contain integers or reals greater than zero. The declaration

floor: TYPEDEF i: pos_integer (i ≤ n_floors)

similarly defines floor to be a positive integer in the range 1 to n_floors.

### 2.2.3. Variables

In ASTRAL, one state is differentiated from another by the values of the state variables, and it is the state variables that are referenced and/or modified by the state transitions. All of the state variables must be declared in the variable section of a process type specification.

In the Elevator specification, there are five state variables. The first,

position: floor,

indicates the floor that the elevator currently resides on. The next four,

going_up, door_open, moving, door_moving: boolean,

indicate the direction of movement, the status of the elevator door, and whether or not the elevator and door are moving.

A special variable called *now* is used to denote the current time. The value of now is zero at system initialization. ASTRAL specifications can refer to the current time (now) or to an absolute value for time that must be less than or equal to the current time. The ASTRAL specification function *past* is used to specify the value that an expression had at some time in the past. That is, past(E, t)

represents the value that the expression E had at time t. As a consequence of the restriction on now, ASTRAL cannot express unbounded eventuality properties, which state that some event must occur at an unspecified time in the future. If it is possible to show that an event must occur in the ASTRAL model, however, then it is also possible to give an explicit bound by which the event will occur. In addition, unbounded eventuality is not useful in real-time systems as it provides no guarantee about when the given event will occur.

### 2.2.4. Definitions

In ASTRAL, definitions are used to make the specification more readable. They may contain zero or more parameters. The request_above definition in the Elevator process,

```
DEFINE
    request_above(f0: floor): boolean ==
        EXISTS f: floor
            (   f > f0
            &   (   the_elevator_buttons.floor_requested(f)
                |   the_floor_buttons[f].up_requested
                |   the_floor_buttons[f].down_requested)),
```

is used as a shorthand for determining if a request is outstanding on any button above the given floor.

### 2.2.5. Interface Section

The interface section of an ASTRAL process specification indicates which types, constants, and definitions declared in the global specification are used by the process, which variables and transitions exported by other processes are referenced, and which variables and transitions are exported by the process. These are specified by the *import* and *export* clauses.

State variables and transitions may be explicitly exported by a process, which makes the variable values and information about the transitions visible to other processes. Exported variables and transitions must be explicitly imported to be referenced in another process specification. Exported transitions are visible to the external environment and are executed in response to calls issued by the environment. The export clause for the Elevator_Button_Panel process,

```
EXPORT
    floor_requested, request_floor,
```

indicates that the value of floor_requested can be imported by other processes (in particular, by Elevator) and that the transition request_floor is made available to the external environment.

The import clause indicates which globally declared types, constants, and definitions are used by a process and which variables and transitions exported by other processes are referenced by this process. The import clause for the Elevator_Button_Panel process,

```
IMPORT
        floor, request_dur, clear_dur, the_elevator, the_elevator.position,
        the_elevator.door_open, the_elevator.door_moving,
```

indicates that the floor type declared in the elevator system's global specification is imported, as well as the global constants request_dur and clear_dur, and position, door_open, and door_moving, which are imported from the the_elevator instance of the Elevator process type.

### 2.2.6. Initial Clause

The initial clause of a process specification expresses the restrictions on the initial state of the process type. That is, it places restrictions on the values that the state variables of the process can have at system initialization. If more than one value satisfies the restrictions placed on a particular variable, the initial value of the variable is nondeterministically chosen from the appropriate range. Similarly, if no restrictions are placed on a variable, it can have any value of the appropriate type at system initialization. The initial clause for the Elevator process,

```
INITIAL
        position = 1
    &   going_up
    &   ~door_open
    &   ~moving
    &   ~door_moving,
```

indicates that the elevator is initially stopped on the first floor, with its door closed and up as its direction of movement.

### 2.2.7. Transitions

ASTRAL transitions are used to specify the ways in which an instance of a process type can change from one state to another. A transition is composed of a header, an entry assertion, and an exit assertion. The header gives type information for the transition's parameters and specifies the amount of time required for the transition to execute. The entry assertion expresses the enabling conditions that must hold for the transition to occur, and the exit assertion specifies the resultant state after the transition occurs. That is, it specifies the values of the state variables in the new state relative to the values they had in the previous state.

In an ASTRAL specification, exceptions are dealt with explicitly. That is, a transition can have except/exit pairs in addition to the standard entry/exit pair. An *except assertion* expresses an

exception that may occur when a transition is invoked. The corresponding exit assertion specifies the resultant state after the transition occurs. Each exception has its own duration, thus a transition may have a different execution time depending on whether it was invoked based on the entry assertion or on one of the exceptions.

Each transition is either a local transition or an exported transition. A local transition is enabled when its entry assertion is satisfied. An exported transition, however, is only enabled when both its entry assertion is satisfied and when it has been called (i.e. invoked) from the external environment. A transition is executed as soon as its enabled assuming no other transition in that process instance is executing. If two or more transitions are enabled simultaneously, a nondeterministic choice will occur and only one of them will be executed. Whenever a process instance starts executing an exported transition, it broadcasts the start time and the values of the actual parameters to all interested processes (i.e. any process that has imported the transition). When the transition is completed, the end time as well as the new values of any exported variables that were modified by the transition are broadcast. Of course, any exported variables that are modified by a non-exported transition are also broadcast by the process when the transition completes execution. Thus, if a process is referencing the value of an imported variable while a transition is executing on the process instance exporting the variable, the value obtained is the value the variable had when the transition commenced. That is, the ASTRAL computation model views the values of all variables being modified by a transition as being changed by the transition in a single atomic action that occurs when the transition completes execution.

*Start(T, t)* is a predicate that is true if and only if transition T starts at time t and there is no other time after t and before the current time (now) when T starts (i.e. t is the time of the last firing of T). For simplicity, the functional notation *Start(T)* is adopted as a shorthand for "time t such that Start(T, t)" whenever the quantification of the variable t (whether existential or universal) is clear from the context. $Start_k(T)$ is used to give the start time of the kth previous occurrence of T. References to the end time of a transition T may be specified similarly using *End(T)* and $End_k(T)$. Figure 2.2.7 illustrates the definition of Start for a transition T. In the interval [t1, t4), Start(T) = t1. At t4, however, Start(T) = t4 and $Start_2(T)$ = t1.

The close_door transition represents the time at which the elevator begins to close the door. The value close_dur is the execution time for this transition. The entry assertion expresses the fact that the elevator can only close the door when the door is open and stationary and the time since it became open is at least t_stop time into the past. Notice that in exit assertions, variable names

followed by a prime (′) indicate the value that the variable had when the transition began firing. The exit assertion expresses that the door of the elevator begins moving to the closed position.

```
TRANSITION   close_door
  ENTRY      [TIME: close_dur]
             door_open
     &       ~door_moving
     &       now - t_stop ≥ Change(door_open)
  EXIT
             door_moving
```

The close_door transition shows the use of the ASTRAL predicate *Change*, which is used to denote the last time that an expression has changed its value. That is, *Change(E, t)* is true if and only if the expression E has changed value at time t and there is no other time between t and the current time (now) at which the expression value has changed. Change follows syntactic conventions similar to Start and End, where *Change(E)* is the last time that the value of E changed and $Change_k(E)$ is the kth previous time that it changed.

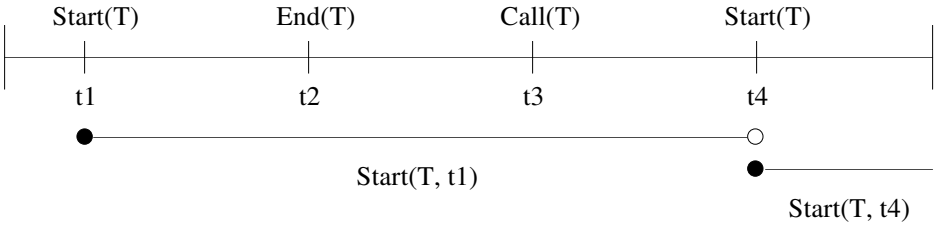



Figure 2.2.7: The last start

### 2.2.8. Environment Clause

Because ASTRAL is intended to be used for designing reactive systems, it is necessary to be able to express assumptions about the external environment in which the system operates. This is accomplished by using environment clauses, which formalize the assumptions that must hold on the behavior of the environment to guarantee certain desired system properties. These assumptions are expressed as first-order formulas involving calls to exported transitions. If T is an exported transition, *Call(T)* may be used in the environment clause to denote the time of the last occurrence of a call to T (with the same syntactic conventions as Start(T) and End(T)). $Call_k(T)$ denotes the time of the kth previous occurrence of a call. It should be noted that there might be a delay from the time a transition T is called until it is actually started. Also, if there are multiple calls to transition T before it fires, then Call(T) is the time of the first of these calls. Figure 2.2.8 illustrates the definition of Call for a transition T. In the interval shown, Call(T, t3) holds at and after t3. Calls to T that occur in the interval (t3, t4] are ignored since there is already a call to T outstanding.

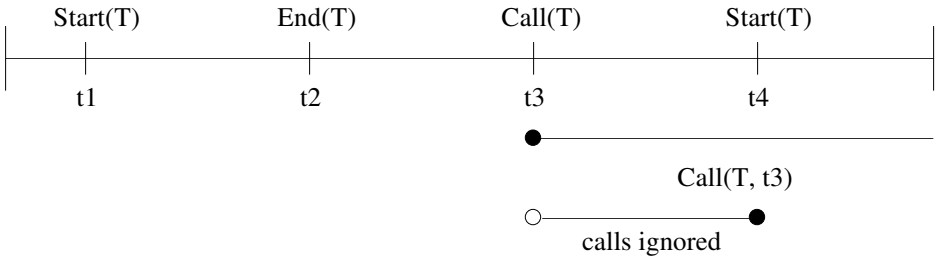



Figure 2.2.8: The last call

There are both local and global environment clauses. Local environment clauses refer to a single instance of the process type they are associated with, while the global clause refers to the system as a whole. The environment of the Elevator_Button_Panel process type,

```
ENVIRONMENT
        Change(the_elevator.door_moving, now)
    &   the_elevator.door_moving
    &   the_elevator.door_open
    &   the_floor_buttons[the_elevator.position] = Self
→       FORALL t: time
            (   Change2(the_elevator.door_moving) ≤ t
            &   t ≤ now
            →       (   past(the_elevator.going_up, t)
            →           ~Call(request_up, t))
            &   (   ~past(the_elevator.going_up, t)
            →           ~Call(request_down, t))),
```

states that requests cannot be made for the elevator to stop at a floor in the current direction of movement between when the door starts opening on that floor until when it starts closing. This assumption is necessary to prevent the elevator from being delayed indefinitely by repeatedly pressing the button at the elevator's current position before the door has closed, which would result in the door repeatedly opening and closing on the same floor without servicing other requests in the building.

### 2.2.9. Imported Variable Clause

ASTRAL also allows assumptions about the system context provided by other processes in the system to be expressed in the *imported variable clause*. This clause describes patterns of changes to the values of imported variables, including timing information about any transitions exported by other processes that may be used by the process being specified (e.g. Start(T) and End(T)). The imported variable clause is optional and is not an essential part of an ASTRAL specification. It is used to aid in proving the correctness of a system in a modular fashion. Figure 2.2.9 illustrates the difference between imported variable assumptions and environment assumptions. Imported variable

assumptions are made about entities within the system, while environment assumptions are made about those that are outside.

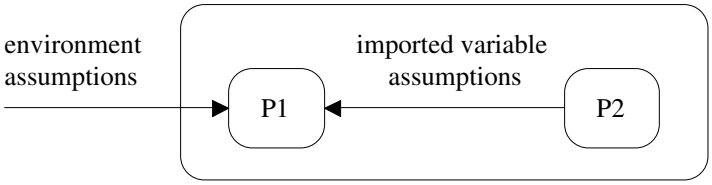



Figure 2.2.9: Imported variable assumptions vs. environment assumptions

The portion of the imported variable clause of the Elevator process type,

```
IMPORTED VARIABLE
    FORALL f: floor
        (   Change(the_elevator_buttons.floor_requested(f), now)
        &   ~the_elevator_buttons.floor_requested(f)
    →       EXISTS t: time
                (   Change2(the_elevator_buttons.floor_requested(f)) < t
                &   t ≤ now
                &   past(position, t) = f
                &   ~past(door_open, t)
                &   past(door_moving, t))),
```

states that a button on the elevator button panel only clears after the elevator has arrived and started opening the doors. This assumption is necessary so that the buttons do not get cleared just before the elevator arrives at a floor and then get requested again after it leaves, causing the elevator to skip the floor, thus delaying the service time.

### 2.2.10. Critical Requirements

For a real-time system, there are two types of critical requirements: behavioral and temporal. In ASTRAL, both types are expressed in the invariants, constraints, and schedules.

#### *2.2.10.1. Invariants and Constraints*

The invariants express the critical requirements that are to hold in every reachable state. That is, they express properties that must initially be true and must be guaranteed to hold during system evolution. The invariant for the Elevator process,

```
INVARIANT
    moving → ~door_open & ~door_moving,
```

states that whenever the elevator is moving, the elevator door must stay closed.

The constraints express the critical requirements that must hold between any two states that correspond to the start and end of a transition. The requirements contained in a constraint could be

expressed in an invariant. Thus, the constraint is just a notational convenience. Invariants can be global or local, where global invariants represent properties that need to hold for the system as a whole, while local invariants and constraints defined at the process type level represent properties that must hold for each process instance. Invariant and constraint properties must be true regardless of the environment or context in which the process or system is running. The constraint for the Elevator process,

```
CONSTRAINT
        (going_up & ~going_up′ → ~request_below′(position′))
    &   (~going_up & going_up′ → ~request_above′(position′)),
```

states that whenever the elevator changes direction, there cannot have been a request outstanding in the old direction when the decision was made.

*2.2.10.2. Schedules*

Schedules are additional system properties that are required to hold under more restrictive hypotheses than invariants and constraints. Like invariants, schedules may be either local or global and obey similar scope rules. Unlike invariants, however, they express requirements that are to hold provided the environment and system context produce stimuli as prescribed in the environment and imported variable clauses.

Process schedule clauses deal with timing requirements for that process only. A process schedule cannot prescribe the values of variables for another process. It may refer only to calls to its own exported transitions and references to the values of imported variables from another process. The portion of the global schedule,

```
SCHEDULE
        FORALL f: floor
            (   the_elevator_buttons.Call(request_floor(f), now - t_service_request)
            → EXISTS t: time
                    (   now - t_service_request < t
                    &   t ≤ now
                    &   past(the_elevator.position, t) = f
                    &   past(Change(the_elevator.door_open, t), t)
                    &   past(the_elevator.door_open, t))),
```

states that whenever a request is made on the elevator button panel for the elevator to stop at a given floor, the elevator must arrive at that floor and open the door less than t_service_request time afterward.

### 2.2.11. Further Assumptions and Restrictions

Schedules are not required to be proved using the basic elements of an ASTRAL specification. It is important, however, to know that the schedule is feasible. There may be several ways to assure that a schedule is satisfied, such as giving one transition priority over another or making additional assumptions about the environment. Even though this kind of decision should often be postponed until a more detailed design phase, it is important to know that if further restrictions are placed on the specification and/or if further assumptions are made about the environment, then the schedule could be satisfied. For this reason, a *further assumptions and restrictions clause* can be included as part of a process specification. Unlike other components of an ASTRAL specification, this clause is only used as guidance to the implementer and is not a hard requirement.

The further assumptions and restrictions clause consists of two parts: a *further environment assumptions* section and a *further process assumptions* section. The further environment assumptions section obeys the same syntactic rules as the local environment. It states further hypotheses on the admissible behaviors of the environment interacting with the system. The further process assumptions section restricts the possible system implementations by specifying suitable selection policies in the case of nondeterministic choice between several enabled transitions, *transition selection*, or by further restricting constants, *constant refinement*. In general, the further process assumptions reduce the level of nondeterminism of the system specification.

# Chapter 3

# Real-Time Specification Languages

A *real-time system* is a system that must perform its actions within specified time bounds. With the advent of cheap processing power and increasingly sophisticated consumer demands, real-time systems have become commonplace in everything from refrigerators to automobiles. Besides such numerous everyday uses, real-time systems are also being employed in more complex and potentially deadly applications such as weapons systems and nuclear reactor control where deviation from critical timing requirements can result in disastrous loss of lives and/or property. To gain assurance that such deviations will not occur, it is highly desirable to be able to extensively test and verify the design of these systems before they are actually built. To meet this demand, an increasing number of formal methods for real-time systems are being proposed [Ost 92, HM 96]. These methods provide a framework under which developers can eliminate ambiguity, reason rigorously about system design, and prove that critical requirements are met using well defined mathematical techniques.

Real-time systems are characterized by concurrency, asynchrony, and dependence on the environment in which they operate. Real-time formal methods must be general enough to model all of these characteristics and powerful enough to express all relevant critical properties in a usable form. There is no consensus about what methodology works the best for real-world systems. Most of the proposed approaches, however, fall into one or more of a small number of classifications, those being *temporal logics*, *state machines*, *process algebras*, *Hoare logics*, and *programming languages*. In this chapter, for each classification, a standard untimed language is presented, along with proposed real-time extensions, verification strategies, and how each is related to ASTRAL. Not all of the existing languages are presented, but those that do not appear are most likely a variant of the ones shown. In the next chapter, it will be argued why the various strategies for automating the proof process discussed below are either inadequate and/or undesirable for reasoning about ASTRAL systems and thus why a new scheme is needed.

## 3.1. Temporal Logics

Traditional temporal logics use abstract operators, such as *henceforth* and *eventually*, to reason about the behavior of systems evolving over time. By using these operators, however, time is completely abstracted away so it is not possible to specify hard real-time requirements. Only properties such as temporal ordering and reachability can be specified. Since temporal logics are very intuitive for expressing system properties, many of the untimed versions have been extended to support the specification of hard real-time constraints. Temporal formulas must be interpreted over a semantic structure (i.e. a set of states temporally related by an interconnection topology, where a state is a mapping from variable names to values in the appropriate domains) to give them meaning. There are four basic types of semantics that have been proposed to interpret temporal formulas: *linear*, *branching*, *interval*, and *partial order*, which are each discussed in the following sections. Time is incorporated into temporal logics by adding a *clock*, which always holds the current time in the system. A clock can be defined over either a discrete or a dense time domain and can be either *explicit* or *implicit*. The value of an explicit clock can be referenced directly in formulas as a state variable. The value of an implicit clock, however, can only be referenced indirectly by using special clock-based operators. The next subsections discuss the various types of temporal semantics and real-time extensions to each that encompass the different varieties of clocks.

### 3.1.1. Linear

Manna-Pnueli temporal logic [MP 92] is an example of an untimed temporal logic interpreted over a linear semantic structure. The structure used is an infinite sequence of states beginning at some initial state. A state is a mapping from a declared set of typed variables to values in the appropriate domains. Formulas in the logic are constructed from first-order expressions in the state variables and temporal operators. These operators include future operators such as next ($\bigcirc$), henceforth ($\Box$), eventually ($\Diamond$), until ($\mathcal{U}$), and unless ($\mathcal{W}$) as well as their past counterparts previous ($\ominus$), has-always-been ($\boxminus$), once ($\diamondminus$), since ($\mathcal{S}$), and back-to ($\mathcal{B}$). From these operators, properties such as safety ($\Box\neg p$), liveness ($\Box\Diamond p$) and fairness ($\Box\Diamond enabled(t) \rightarrow \Box\Diamond taken(t)$) can be expressed. As an example of interpreting a formula over a linear model, consider $\Box(p \rightarrow \bigcirc q)$. This formula is interpreted over a state sequence to mean that in any state of the sequence in which p holds, q will hold in that state's immediate succeeding state. The following sections discuss a number of linear temporal logics that have been proposed to express hard real-time constraints and the strategies they use for verifying these constraints. Real-Time Temporal Logic (RTTL) is a discrete-time explicit-clock

extension of Manna-Pnueli logic, but will be discussed later in the TTM/RTTL framework of section 3.2.1.3.

*3.1.1.1. Timed Propositional Temporal Logic*

Propositional temporal logic is a subset of standard temporal logic that only allows boolean variables and does not allow quantification. Timed Propositional Temporal Logic (TPTL) [AH 94] is a linear-time, implicit-clock real-time propositional temporal logic interpreted over a discrete time domain. Since the goal of TPTL is to express precise timing requirements, the standard linear structure of Manna-Pnueli logic cannot be used. Instead, a *timed state sequence* is used, which is a pair consisting of an infinite sequence of states and an infinite sequence of monotonically nondecreasing times. At the ith time in the time sequence, the system is in the ith state in the state sequence.

TPTL formulas are propositional temporal logic formulas with the addition of two types of expressions not available in the untimed version. The first expression is a timing constraint in the form $t_1$ r_op $t_2$ where r_op is a relational operator and $t_1$ and $t_2$ are either integer constants, time variables, or time variables plus a constant. Time variables are defined through the use of the second extension, the *freeze quantifier* "x.". $x.f(x)$ is equivalent to the first-order logic expression $\exists x.(x = now \wedge f(x))$ where now is the current time in each state. The freeze quantifier binds the time of an event occurrence to the variable specified. For example, $\Diamond x.(p \wedge x \leq c)$ means that p must hold in some state less than or equal to c time units into the future.

The verification of TPTL formulas is performed using a *tableau-based* algorithm. The tableau approach is used to determine the satisfiability of a formula by checking all possible models for the desired property. To verify that the properties p of a system model m hold, the tableau approach is used to determine the satisfiability of $m \wedge \neg p$. If it is satisfiable, then the properties do not hold because there is some execution in which the system is behaving as m, but not as p. If it is unsatisfiable, then no such execution is possible, thus p holds in m.

The tableau approach involves reducing an infinite model to a corresponding finite one that can be searched by brute-force for a satisfying model. The full TPTL tableau algorithm is fairly complex so only the basic strategy will be discussed. The key to the algorithm is to split formulas into present state and next state conditions. For example, $\Diamond f$ can be satisfied by f or $\bigcirc\Diamond f$. Timing constraints can be split given the fact that if $x \leq y + c$, then $x \geq y + c + k$ is false for all $k \geq 1$ so the original timing constraint needs to be split into at most c assertions. For example, from [AH 94], $x.\Diamond y.\psi(x, y)$ can be satisfied by $y.\psi(y, y)$, $x.\Diamond y.\psi(x-1, y)$, $x.\Diamond y.\psi(x-2, y)$, ..., or $x.\Diamond y.\psi(x-c, y)$ if x and y are related by x

$\leq y + c$. Since all the constants in a formula are known and discrete time is used, there are a finite number of these splits possible. From the split formulas, the finite tableau graph is created where nodes correspond to states and are labeled with formulas that hold in that state and formulas that must hold in any immediate successor states (nodes). Any path in this graph that satisfies the original formula is eventually periodic and its length can be bounded. The algorithm thus checks for some path up to that length that satisfies all eventualities and does not violate any invariants to show satisfiability.

The TPTL algorithm is notable because it is able to determine satisfiability over an infinite time domain. Unfortunately, when a dense time domain is used instead or more powerful timing constraints are allowed (e.g. $t_1 \leq 2t_2$), satisfiability of TPTL becomes undecidable [AH 94].

*3.1.1.2. TRIO*

TRIO [GMM 90] is not defined in terms of conventional temporal logic operators. Instead, it uses a combination of first-order logic and two special predicates Futr and Past. Properties are expressed with respect to the current time, which is left implicit. Futr(A, t) states that the formula A is true in the system at the instant t time units into the future. Past(A, t) is similar but is with respect to the past. From these operators, any of the standard temporal operators and many new operators can be derived. For example, $A_1 \mathcal{U} A_2$ is defined as $\exists t.(\text{Futr}(A_2, t) \wedge \forall t'.(0 < t' < t \rightarrow \text{Futr}(A_1, t'))$. A TRIO temporal structure, used to interpret formulas, is a set of variable domains, a temporal domain, a time-independent valuation function for each constant name, and a time-dependent valuation function defined at each point in the temporal domain for each variable name. TRIO formulas containing time-dependent variables must be *temporally closed*. That is, they must be written so that they are true or false for an entire temporal structure and not just a single instant. For arbitrary TRIO formulas, satisfiability is undecidable. When the temporal domain and all variable domains are finite, however, TRIO is supported by a tableau-based verification algorithm.

TRIO is part of the basis for ASTRAL, so the logic used in ASTRAL is similar to TRIO but without the Futr operator and with more built-in operators and an explicit time variable. The ASLAN-based portion of ASTRAL can thus be thought of as defining a TRIO temporal structure, with types defining domains, defines and constants defining the time-independent part, and variables and transitions defining the time-dependent part. The Call operator can be simulated with a parameterized time-dependent predicate. An actual translation from ASTRAL to TRIO is discussed for an earlier version of ASTRAL in [GK 91b].

*3.1.1.3. Temporal Logic of Actions*

The Temporal Logic of Actions (TLA) [Lam 91, AL 91, AL 93] is an untimed temporal logic used for describing concurrent systems that has been extended to real-time through the use of an explicit clock variable. TLA formulas are interpreted over an infinite sequence of states, called a *behavior*, where a state associates values with all variables in the system. A TLA system is defined in terms of *actions*. An action is a boolean expression containing primed and unprimed variables that is interpreted as true or false for a pair of states. An action is essentially a transition in which the conjuncts that do not contain any primes represent the entry assertion and those that do contain primes represent the exit assertion. A pair of states that satisfies an action A is called an *A-step*. An A-step is a *stuttering step* if both its states are the same. $[A]_f$ defines a pair of states that is an A-step or in which $f = f'$, where f is a state function and f′ is f with all variables primed. $\langle A \rangle_f$ defines a pair of states that is an A-step and in which $f \neq f'$. In most cases, f will be a tuple of system variables so the two expressions will allow and disallow stuttering steps, respectively. The canonical form for TLA specifications is $\exists x.(init \wedge \Box[N]_v \wedge L)$. This specification asserts that there is some way to choose values x such that the initial condition is satisfied and each pair of states is an N-step or leaves v unchanged, and L holds. L is a conjunction of constraints such as fairness, disjointness, and timing restrictions. The *weak fairness* condition, $WF_v(A)$, requires either infinitely many A-steps to occur or infinitely many steps in which A is not enabled. Weak fairness is also called *justice*. The *strong fairness* condition, $SF_v(A)$, requires either infinitely many A-steps to occur or only a finite number of steps in which A is enabled.

TLA can represent both *interleaving* and *noninterleaving* models of concurrency. In an interleaving model, only a single transition can occur at any point in time. In a noninterleaving model, however, multiple transitions may occur simultaneously. All TLA specifications are noninterleaving unless explicit interleaving restrictions are given. Such restrictions limit the amount of concurrency allowed by stating which variables can change value at the same time and/or which actions can occur in parallel. ASTRAL uses a noninterleaving approach in which different processes can fire different transitions at the same time.

In TLA, real-time constraints are expressed using the variable now, which is not a language primitive with special semantics, like in ASTRAL, but instead is a simple system variable whose behavior must be explicitly defined. That is, properties of time, such as its domain and that it always advances, must be explicitly given as TLA formulas. Timing requirements are placed on system components using *timers* that restrict the advance of now. A timer t is a state variable with the same

domain as now whose value is updated according to its associated TLA formula. For example, the formula associated with a *volatile timer* declares that the timer is always greater than now unless a given action has been continuously enabled for some constant amount of time at which point the timer keeps the same value. This type of timer can be used to express requirements such as firing deadlines. Given a volatile timer t, $now \leq t \wedge \Box[now' \leq t']_{now}$ states that now can never be advanced past t. If the action occurs at now < t, then t will be reset by the definition of a volatile timer, but at now = t, either the action must occur or time stops, which would contradict the definition of now.

TLA specifications can be written in *assumption/guarantee form* $E \xrightarrow{+} M$, where E and M are TLA formulas (possibly in assumption/guarantee form themselves) and M must hold for at least one step longer than the environment E. This form is for reusable components where the correct operation of a single component depends on certain conditions in the rest of the system. TLA also has a conditional implementation style in the form $G \wedge M' \rightarrow M$, which states that M′ implements M, assuming G holds. Conditional implementation is used in *open systems*, which depend on the behavior of the external environment. These two forms are similar to ASTRAL imported variable and environment clauses, respectively. That is, E contains assumptions about input variables that must be provided by other system components while G contains assumptions about the world in general that cannot be explicitly provided by another system (e.g. laws of nature). While TLA makes this distinction, it does not distinguish between properties that depend on the environment and those that do not, like the separation of schedule and invariant in ASTRAL.

A set of axioms and inference rules is given in [Lam 91] to prove that TLA specifications meet critical requirements. In addition, [AL 93] presents *decomposition* and *composition* theorems to simplify the proof process. Instead of proving properties over a complete system, proofs can be modularly performed on individual components, which are in general much simpler. This is similar to the approach used in ASTRAL where the proof obligations for each process are performed using only the environment and imported variable clauses for that process and are independent from the actual implementations of other processes.

### 3.1.2. Branching

Computation Tree Logic (CTL) [CES 86] is an example of an untimed branching temporal logic. In CTL, a state is a set of atomic propositions that hold in the system. A CTL structure is defined by a finite set of states, a set of possible transitions between states such that each state has one or more successors, and a state designated as the initial state $s_0$. Thus, it defines an *infinite computation tree*

with $s_0$ as the root and each edge originating from a node representing a possible choice (i.e. transition) the system can make. A *path* is an infinite sequence of states starting at the root and traversing down the computation tree, which represents one possible evolution of the system.

CTL formulas are built using atomic propositions, standard logic connectives, and modified linear temporal logic operators that restrict the possible paths of system evolution. Linear temporal logics use the operators $\bigcirc$ and $\mathcal{U}$ to indicate that a formula holds in the next state and that one formula holds up until the instant the second holds, respectively. In branching logic, however, these operators are not well defined because there are several possible next states and hence several possible sequences of states. Therefore, these operators have been modified to describe the paths over which the formulas hold. A $\forall$ or $\exists$ is added before the $\bigcirc$ and $\mathcal{U}$ operators to indicate that they hold in all paths or in some path. For example, $\exists\bigcirc f$ says that f holds in at least one of all the possible next states, while $\forall(T\ \mathcal{U}\ f)$ says that f inevitably holds (i.e. f eventually holds along all possible paths).

Since CTL computation trees are defined to include all possible transitions at each node, every possible path is represented in the tree whether reasonable or not. In particular, it includes *unfair* executions where requests for service are not granted sufficiently often. Proving properties of the system in such instances is often counterproductive because the mere fact that a request is not being serviced most likely means the system is already behaving incorrectly. Since pure CTL cannot limit execution paths to fair sequences, it has been extended to $CTL^F$. The syntax and semantics of $CTL^F$ are identical to CTL with the exception that formulas are interpreted over only the fair paths of the computation tree as given by a set F. The elements of F are sets of states. A path p is said to be fair if for each set G in F, there are infinitely many states in p that are elements of G.

Another extension of CTL, called $CTL^*$ is CTL, but with path quantifiers disassociated from temporal operators. For example, $\forall\Diamond f \rightarrow \bigcirc g$ means that in every path, whenever f holds, g holds in the next state. $CTL^*$ can easily handle fairness conditions without the need for the introduction of F-fair paths. The drawback is that the model checking algorithm used for verification (discussed for TCTL below) becomes PSPACE-complete instead of PTIME with respect to the size of the branching structure.

*3.1.2.1. Timed Computation Tree Logic*

Timed Computation Tree Logic (TCTL) [ACD 90] is a dense-time, implicit-clock, branching logic based on CTL. TCTL extends the CTL branching structure and syntax to allow the expression of hard real-time requirements. A TCTL model consists of a set of states S and an initial state as

defined for CTL above. Instead of a set of possible transitions, however, it contains a mapping f from states to sets of *s-paths*. An s-path is a mapping p from the non-negative reals to S satisfying p(0) = s. That is, it is a timeline that depicts the state of the system at any point along it, which describes an execution path beginning at state s. The mapping f must satisfy the *tree constraint*, which states that for any s-path p in f(s), any prefix of p up to some time t "concatenated" with any s′-path p′ in f(s′) is also in f(s), where s′ = p(t). That is, there is an s-path p″ in f(s) such that p″(t′) = p(t′) if $0 \le t' \le t$ and p″(t′) = p′(t′ - t) if $t' > t$.

The syntax of TCTL is similar to CTL but there is no $\exists\bigcirc$ operator since the next state is not well defined in dense time, and until expressions can be limited with a time expression (i.e. <, ≤, =, ≥, >). For example, $\exists(f_1 \ \mathcal{U}_{<c} \ f_2)$ says that there is a prefix of some path of time length less than c such that $f_2$ holds in the last state and $f_1$ holds at all preceding times. TCTL can be extended in a similar fashion as CTL to $TCTL^F$ and $TCTL^*$.

A TCTL formula is satisfiable if there is a TCTL structure and a state in which the formula holds. Satisfiability of TCTL formulas is undecidable. [ACD 90] introduces *timed graphs* to define TCTL structures. A timed graph is a graph in which nodes represent states and edges are transitions labeled with either reset(x), which resets the clock x of a finite set of clocks or a boolean formula using x r_op c where r_op is a relational operator, x is a clock and c is a constant. A transition can be taken only if its associated condition holds and must be taken before it becomes false. A TCTL formula is *finite satisfiable* if there is a timed graph defining a TCTL structure such that the formula holds in that structure. This problem is also undecidable. It is decidable, however, whether the structure defined by a specific timed graph satisfies a TCTL formula.

TCTL uses a *model checking* algorithm to determine if a timed graph satisfies a TCTL formula. A model checker is characterized by transforming an infinite structure (such as a TCTL computation tree) into a finite structure of equivalence classes that can be analyzed to obtain results for the original infinite structure. Equivalence in this case means that the given formula cannot be distinguished over the structure. In the case of the TCTL model checker, values of clocks in the given timed graph determine the equivalence classes. A *region graph* is constructed in which equivalence classes are nodes and edges represent changes in the system from one equivalence class to another caused by either the passage of time or a transition in the timed graph. After the region graph is built, validity of a TCTL formula is verified by checking the formula over all appropriate paths in the region graph, which is guaranteed to be a finite number.

ASTRAL is similar to TCTL in that from the initial state, the system can evolve in infinitely many ways depending on times of calls and nondeterminism between transitions. Thus, the possible evolution of an ASTRAL system resembles a TCTL structure, where at any time along the dense time domain, a number of new branches may sprout new possibilities for system evolution. In ASTRAL, however, formulas can only refer to the past so it is already determined along which path the system has evolved when a formula is evaluated. The invariant and schedule clauses do capture some of the spirit of the TCTL $\forall\Box$ operator because every possible branch in the future must satisfy the given properties. The TCTL model checking approach is of interest because it is able to deal with a dense time domain.

### 3.1.3. Partial Order

Partial Order Temporal Logic (POTL) [PW 84] is a temporal logic interpreted over a partially ordered structure. POTL is actually a hybrid of linear and branching temporal logics. The semantic structure of POTL is almost identical to that of CTL. A POTL structure consists of a finite set of states, a set of transitions between states such that each state has at least one successor and at least one predecessor, and a state designated as the initial state. Unlike CTL, however, which can only reason about future states, POTL uses the operators $\ominus$, $\boxminus$, and $\diamondsuit$ from linear temporal logic to reason about states in the past. Like CTL, temporal operators must be prefaced with a quantifier to indicate that the properties hold along all possible future (or past) execution paths or at least one execution path. "Partial ordering" comes about from the interpretation of the structure. In CTL, the tree is meant to represent choices made by the system at each point of its evolution. POTL, on the other hand, views the branches not as just choices, but also as splits in execution where a single process forks to become multiple processes that perform different functions that are not ordered between themselves until they are joined or synchronized in some way. Hence, the past operators are used to describe join operations such as when a process commits a transaction then all processes have "once" (i.e. $\diamondsuit$) sent a commit message. This type of property does not have any information on the ordering between the times each process sent the commit message, only that indeed they have sent it, thus the term partial order.

Partial order temporal logics are useful for specifying distributed systems and other systems in which multiple entities are operating concurrently and synchronize only at well defined points in time. Since this type of behavior is typical of real-time systems, there has been an effort to extend untimed partial order logics to be able to express hard real-time requirements. One advantage such extensions have is that they do not make the often unrealistic assumption that all processes have access to a

global clock like ASTRAL and a number of other languages do. Instead, time comparisons are only allowed between events that are known to occur before one another (e.g. send/receive). This eliminates the possibility that a system will be specified that has no realistic feasible implementation because of a global clock assumption.

*3.1.3.1. Distributed Logic*

Distributed Logic (DL) [MP 91] is a first-order logic based on a combination of partial order and linear semantics. The *configuration* of a distributed system is defined by a set of system nodes (i.e. processors), a set of one-way communication channel names, a speed assignment from nodes to real numbers (the lower, the faster), and a partial mapping from nodes and channels to nodes. The partial mapping can be extended to the transitive closure so that all possible node interconnections are represented. A *computation* of a node is a standard totally ordered linear state sequence where states are differentiated at the occurrences of events. The four types of events are *transition events*, which represent changes in the internal state of a node, *external events*, which are occurrences of significance in the environment, *notifier events*, which denote the time a message is sent over a channel, and *notification events*, which correspond to the reception of a message. A run of the system is a partial ordering among all the runs of the individual nodes. A state in which a notifier event occurs on some node (i.e. a message is sent by that node) is ordered before the state in which the corresponding notification event occurs on the receiving node. Thus, all states before the notifier event on the sending node are ordered before all states after the notification event on the receiving node. An *action* represents some instruction execution on a system node. An action is defined by a tuple ($e_r$, $e_s$, $e_e$, $t_u$, $t_l$). $e_r$ is the time that the node is ready for the abstract scheduler to run the action. $e_s$ is the time of the start of the action and $e_e$ is the ending time. $t_u$ and $t_l$ are upper and lower bounds on the running time of the action.

A DL formula is a first-order logic formula with the addition of two operators. $G_i(f)$ indicates that henceforth, f holds on node i. Similarly, $F_i(f)$ indicates that f eventually holds on node i. If c is a channel, $c(f)$ denotes that f is true as per the last message sent over c (i.e. to the best of that node's knowledge, f is true). DL formulas are interpreted over a semantic structure consisting of a configuration, a run, and a mapping from variables to values in the appropriate domains. To define an instance of this structure, a programming model is introduced containing various primitives for defining processes, channels, delays, etc. The semantics of the programming model are axiomatically defined in DL. The semantics are used to generate a set of DL expressions from the

given program. These expressions are assumed as axioms and system properties are proven as theorems using a set of inference rules.

ASTRAL is similar to DL in that the system consists of a finite set of processors that each perform a series of sequential actions. Unlike ASTRAL, however, DL processors must explicitly request values of imported variables whenever they need them to make a decision. Also, there is no global clock that is assumed available in DL with which to synchronize. DL can approximate these features by broadcasting the values of exported variables whenever they change as well as starting and ending action times. With this information available, the global clock is not as important because processors can synchronize on starting and ending times instead, which is essentially the strategy in ASTRAL.

### 3.1.4. Interval

Propositional Temporal Interval Logic (PTIL) [SMV 83] is a temporal logic interpreted with an interval semantics. The use of intervals is independent of the choice of structure used to represent executions. That is, an interval semantics can be associated with linear, branching, or partial order structures, although in practice it is paired most often with linear structures. The idea behind interval logics is that many properties do not necessarily need to hold over an entire computation, but instead may only be required to hold over specific finite intervals. Although such interval properties can be expressed in linear logic by using embedded until expressions, it is not without difficulty and a loss of much of the intuition behind the approach. A PTIL execution is an infinite sequence of states where each state corresponds to a set of propositions holding in that state. PTIL formulas are written in the form [I]f, where I is an interval and f is a formula of the same form or one that does not contain an interval. Every formula is evaluated in the context of some interval so in [I]f, the first occurrence of I is found with respect to the entire system execution, but f is evaluated only in the context of I. If I cannot be found in the system execution, f vacuously holds.

The most basic interval is the *event interval*. An event interval [P], where P is a predicate, is the first interval of length two in the interval of the current context (henceforth called the *current interval*) in which ¬P holds in the first state and P holds in the second. That is, an *event* is a change in the value of a predicate from false to true, and an event interval is the sequence of two states in which this change occurs. Other intervals are built from event intervals by using the begin, end, ⇒, and ⇐ operators. Given an interval I, begin I is the one unit interval containing the first state of I. Similarly, end I denotes the interval containing only the last state of I. The ⇒ and ⇐ operators take zero, one, or two interval operands. When the first operand is missing, the beginning of the new

interval is the beginning of the current interval. Similarly, if the second operand is missing, the end of the new interval is the end of the current interval. The direction of the arrow indicates the direction of the search. For example, [I ⇒] is the interval from the first occurrence of I to the end of the current interval. [I ⇐], on the other hand, is the interval from the last occurrence of I to the end of the current interval. The difference in the two examples is that in the first, the search was forward from the beginning of the current interval, while in the second, the search was backward from the second operand, namely the end of the current interval. [I ⇒ J] and [I ⇐ J] are shorthands for [I ⇒][⇒ J] and [⇐ J][I ⇐], respectively. Note that since the operands are intervals, the new interval is defined as being from the end of the first operand to the end of the second operand. Properties of intervals are expressed using boolean combinations of propositions and henceforth and eventually expressions. [I]p is true if p holds in the first state of I. [I]◊p is true if there is some state of I in which p holds. [I]□p is true if p holds in every suffix of I.

*3.1.4.1. Real-Time Future Interval Logic*

Real-Time Future Interval Logic (RTFIL) [RMM 93, RMM 96] is a real-time interval logic that represents systems graphically as collections of "timelines". An RTFIL formula is composed of *intervals* and *searches*. The interval of the entire system execution is always drawn first as the basis for defining other intervals and searches. A search is drawn as a dotted arrow with the formula to be searched for at the head of the arrow. The search formula can be a simple boolean formula or a full interval expression. A search locates the earliest instant in its outer interval (i.e. the closest interval above the search that fully contains it), starting from the tail of the search, in which the formula holds. A single arrowhead denotes a *weak search* such that if the formula is not "found" (i.e. if an instant in which the formula holds is not found) in the associated interval, the outer formula containing the search vacuously holds. A double arrowhead denotes a *strong search* such that if the formula is not found, the outer formula containing the search is false. Intervals are constructed between points found with searches. Intervals can be *weak intervals* or *strong intervals*, depicted as single and double lines, respectively. A weak interval may be empty if the searches defining it succeed, but a strong interval must always be non-empty when its endpoints are found.

A formula may be placed beneath any interval. If the formula is left justified with respect to the interval, then it must hold in that interval's first state. If the formula is centered, however, then it must hold in all of the interval's states. A ◊ placed on the interval expresses that a centered formula must eventually hold. RTFIL formulas can be combined with conjunction, disjunction, and implication. Timing constraints are expressed by using the len predicate. The len predicate must be

left justified beneath an interval. len($d_1$, $d_2$] requires that the associated interval is more than $d_1$ time units in length and at most $d_2$ in length, where $d_1$ and $d_2$ are non-negative rationals or inf for an infinite bound. For example, consider a train crossing in which sig holds when a train has been detected and open holds when the gate is up. Figure 3.1.4.1 shows the RTFIL formula for this property from [RMM 93] where len(1.0, 2.0] represents the requirement that the gate must be closed within one to two minutes of a train being detected.

RTFIL formulas are interpreted over a structure similar to that of dense time TPTL. A *dense trace* is a mapping from non-negative reals to propositions that hold at each time. A number of restrictions are placed on RTFIL models such as disallowing *instantaneous states* (i.e. multiple transitions occurring at the same time) and requiring *finite variability* (i.e. in any finite segment of the reals, only a finite number of state changes are allowed). Satisfiability of RTFIL formulas is decidable and is used to verify that the properties p of a system model m hold by checking that m $\wedge$ $\neg$p is unsatisfiable. The decision procedure utilizes the algorithm for determining the emptiness of a timed Büchi automaton (TBA) discussed in section 3.2.1.5. The RTFIL formula is transformed into a TBA that is non-empty if and only if the formula is satisfiable. For example, the TBA for the formula above would include a transition between a state in which $\neg$sig holds and one in which sig holds that resets some clock c. Another transition would be defined that moves from sig to $\neg$open only when c > 1.0 and c $\leq$ 2.0. States are annotated with clock predicates and the accepting states are those in which the appropriate predicates hold at the correct times (e.g. $\neg$open $\wedge$ 1.0 < c $\leq$ 2.0 in the example).

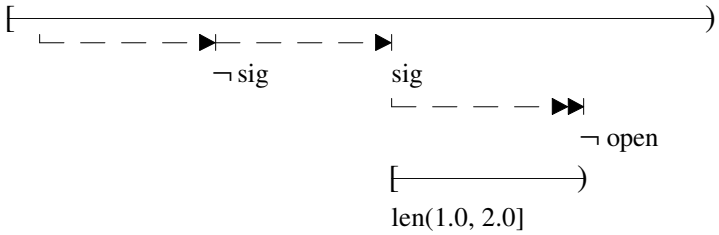



Figure 3.1.4.1: An RTFIL formula

Although ASTRAL does not have interval operators built in to it, intervals are frequently used in invariants and schedules to capture desired properties. In ASTRAL, however, only times in the past can be referenced so instead of searching forward into the future, searches are performed backward up to some point in the past. If the initial search is successful, additional searches are initiated from that point forward up to the present or further backward in the past. For example, the RTFIL formula

shown in figure 3.1.4.1 is expressed in ASTRAL as: Call(sig, Now - 2) → EXISTS t: time (t > Now - 1 & t ≤ Now & End(close, t)). Thus, if a call to sig occurred 2 time units into the past, close must end between 1 and 2 time units after that point.

## 3.2. State Machines

Unlike temporal logics, which have facilities to describe both system behavior and system properties fairly intuitively, state machines are more oriented towards just the behavior specification so they are often incorporated into a *dual-language approach*. In a dual-language approach, one language (in this case, state machines) is used to specify system behavior and the other is used to specify system properties. The semantics of the behavior language are modified such that they define a semantic structure for the property language that formulas can be interpreted over. State machine specifications can often be represented in both textual and graphical form. The languages presented in the following sections are divided into textual and graphical according to their original representations. The graphical section covers those languages that were explicitly graphical from the beginning.

### 3.2.1. Textual

ASLAN [AK 85] is an untimed state machine specification language that has features typical of languages in this class. In its simplest form, an ASLAN specification consists of a set of typed variables, a set of *state transitions*, an *initial condition*, and the *invariant* and *constraint* clauses. The values of variables in an ASLAN specification describe the state of the system. The initial condition is a first-order expression in the variables that limits their values in the initial state. Transitions define how the system can change from one state to the next. Each transition consists of an *entry assertion* and an *exit assertion*. The entry assertion describes the conditions that must hold in the current state for the transition to occur. The exit assertion describes the conditions that must hold in the successive state if the transition is taken. Exit assertions can use primed variables to indicate the value the variable had in the entry state. Note that these assertions do not necessarily prescribe exact values for variables, they only limit the values in the two states. That is, a transition is not like a programming language procedure that performs specific actions on program variables. Instead, transitions limit the possible implementations to those in which the restrictions placed upon the variables hold.

The properties of an ASLAN specification are divided into invariants and constraints. An invariant is a property that must hold in the system in every reachable state. A constraint is a property that

must hold between successive states. That is, it limits the possible changes to variable values by transitions. ASLAN uses an inductive proof system to verify that invariants and constraints hold in the transition model defined. The obligation initial → invariant is proved for the base case and the obligation invariant′ & entry′ & exit → invariant & constraint is proved for each transition.

A similar language of note is the Z notation [Spi 90]. A Z specification is a collection of *schemas*, where a schema describes the state space of the system along with the operations that change the system from one state to the next. For example, the Select schema of figure 3.2.1 from [Spi 90] describes a transition of a real-time kernel specification. The upper portion of a schema declares its state space. The Select schema declares that assertions will be made about changes to State, which is itself a schema. In the case of transitions (i.e. schemas with at least one Δ in their declaration sections), the lower portion is divided into a *precondition* and a *postcondition*, identical to ASLAN entry and exit assertions. Select declares that whenever the CPU is idle, the scheduler will nondeterministically choose a process to run in the background from the set of ready processes and the other components of State will remain unchanged. Z has a *schema calculus* that allows schemas to be related into a hierarchical specification. Disregarding notational differences, Z and ASLAN are essentially the same language with the most notable difference being the "location" of invariants, which are always "globally" located in ASLAN, while in Z, "local" invariants can be given for each schema. This does not, however, result in any difference in expressive power.

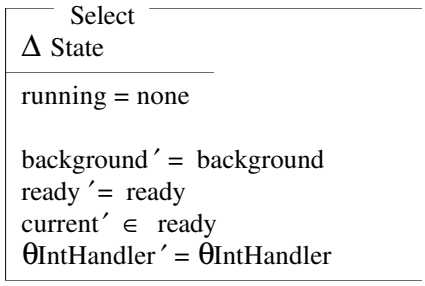



Figure 3.2.1: A Z schema

ASLAN is the basis of much of the ASTRAL language. In essence, ASTRAL becomes ASLAN when the system only has a single process and is closed (i.e. no environment). Given their ability to model system behavior in a natural and intuitive way, there have been many attempts to extend the basic state machine model in order to specify real-time systems.

*3.2.1.1. RT-ASLAN*

An RT-ASLAN [AK 86b] specification consists of a set of *communicating processes* and a set of *interface processes*. Each process executes on its own processor. An interface process is essentially a monitor, which controls access to shared variables. It is defined as an ASLAN specification with a few modifications. Transitions within interface processes, called *interface transitions*, can be declared as either exclusive, meaning that only a single communicating process can access the transition at any given time, or inherited, meaning that any number of communicating processes can access the transition. Exclusive transitions can be thought of as running on the interface processor, while inherited transitions can be thought of as being in-lined on the calling communicating processor, where variables are accessed through shared memory. Interface transitions can have in and out parameters, which are similar to Pascal value and var parameters, respectively. Interface transitions are visible to all communicating processes. In addition, the types and constants of interface processes can be made available explicitly with the visible keyword.

Communicating processes are also modified ASLAN specifications, but with different modifications than those of interface processes. Exit clauses of transitions in these processes are in the form do $A_1$ before ... before $A_n$ od, where each assertion $A_i$ executes atomically and $A_i$ completes before $A_{i+1}$. In addition, transitions must be given a classification of either external, cycle t, or timing t. External indicates that the transition is called from the environment. These transitions are viewed as being instantaneous. Cycle t indicates that the transition must execute at least once every t time units. The execution times of lower level transitions implementing a cycle transition must be less than or equal to the cycle time. Finally, timing t declares that the execution time of the transition is at most t time units. The current time in the system is represented by the time variable.

RT-ASLAN and ASTRAL are both real-time extensions of ASLAN. ASTRAL is based on a message passing paradigm while RT-ASLAN uses a shared memory model. Basing the language on shared memory necessitated introducing interface specifications to restrict access to shared variables. Otherwise, two processes might assert contradictory conditions on the same variable in an exit clause, which has no realizable implementation. This is not possible using the message passing model of ASTRAL because even though an imported variable can be read at any point, only the process that exported the variable can write to it. Another difference is that ASTRAL has a much richer set of time related operators. RT-ASLAN has no built-in mechanism besides the time variable to express timing properties, so to simulate an operator such as End(T), the exit assertion of T must explicitly "record" the time it ends in a variable to be used elsewhere.

*3.2.1.2. Timed Automaton Model*

A Timed Automaton Model (TAM) [LV 91] specification is based on a set of *timed automatons*. Each automaton consists of a set of state variables and a set of *actions* (i.e. transitions). All variables are given initial values that together specify the initial state of the system. Each action is associated with a *precondition* that must hold for the action to occur and an *effect* that describes the new state after it occurs. These closely correspond to ASTRAL entry and exit assertions. State variables in precondition and effect statements are prefixed by "s." to refer to the current state and "s′." for the new state. Every timed automaton has a variable now, which is a non-negative real, initially 0, representing the current time in the system. A timed automaton must also have a time-passage action $\nu(\Delta t)$, which is the only action permitted to modify now. The precondition and effect of the $\nu$ action must be specified explicitly by the user. Essentially, the precondition must be written so that no significant event in the system is skipped by the passage of time. That is, the $\Delta t$ parameter can never be greater than the amount of time into the future that some event (e.g. the end of an action) is to occur. The effect of $\nu$ must always increase the value of now by $\Delta t$, which must always be positive. Actions do not necessarily have a duration associated with them but this can be simulated by recording now plus its duration in a variable in the effect of the transition and then writing the $\nu$ precondition appropriately. For example, in the railroad crossing example of [HL 94], raise has $s'.last(up) = now + t_{up}$ in its effect and $\nu$ has $s.now + \Delta t \leq s.last(up)$ in its precondition. Thus, the up action must occur before $t_{up}$ time from the end of a raise occurrence thereby bounding the duration of raise. $\infty$ can be used as a duration to specify external events that can occur at any time. The behavior of the environment must be specified explicitly as one or more timed automatons.

Before the verification of TAM systems can be discussed, some preliminary definitions must be given. Actions are classified as either *internal*, *external input*, or *external output*. The external input and external output actions are called the *visible* actions of the system. A *timed execution* is an alternating sequence of time-passage actions and non-time-passage actions. *Admissible timed executions* are those timed executions in which the value of now in each state reached by a transition in the sequence approaches infinity. A *timed trace* of a timed execution is the timed execution with only its visible actions and their times of occurrence present. That is, a timed trace is the behavior of the system visible to the environment independent of the internal implementation. The *admissible timed traces* of a timed automaton are the timed traces of the admissible timed executions of the system.

A TAM specification consists of one automaton P to specify the desired visible system behavior (i.e. the properties) and another automaton I that additionally specifies the internal implementational behavior. To verify that the system is consistent, it must be shown that every admissible timed trace of I is an admissible timed trace of P. This can be done by finding a *simulation mapping* from I to P with respect to some invariant, which by a theorem in [HL 94], proves the desired result. For invariants $I_A$ and $I_B$ of timed automatons A and B, a simulation mapping from A to B with respect to $I_A$ and $I_B$ is a binary relation f over the states of A and B that must satisfy three conditions. First, if $(s_a, s_b)$ is in f, then $s_a$.now = $s_b$.now. If $s_a$ is an initial state of A, then f must contain $(s_a, s_b)$ where $s_b$ is an initial state of B. Finally, for all states $s_a$ and $s_a'$ of A, such that $s_a'$ is reached from $s_a$ by an action T, if $s_b$ is a state of B and $(s_a, s_b)$ is in f, then there must exist a state $s_b'$ of B such that $(s_a', s_b')$ is in f and $s_b'$ is reachable from $s_b$ by a timed execution sequence that has the same timed visible actions as the step from $s_a$ to $s_a'$.

A timed automaton is very similar to an ASTRAL process. The real difference between TAM and ASTRAL is in the underlying semantics of both models. TAM is a generic approach that does not have a lot of built-in semantics, as is associated with ASTRAL. For instance, it does not have durations directly associated with actions nor implicit movement of a global clock. It also does not have many of the built-in predicates of ASTRAL such as Start and past. Instead, TAM has a very simple semantics and the flexibility necessary to simulate many of these items by keeping track of appropriate variables in the state and picking the appropriate precondition and effect for the ν action.

*3.2.1.3. Timed Transition Model/Real-Time Temporal Logic Framework*

The TTM/RTTL framework [Ost 89, OW 90] is a dual-language approach with the Timed Transition Model (TTM) acting as the modeling language while Real-Time Temporal Logic (RTTL) is used to express system properties. A TTM is defined by a set of typed state variables, a boolean-valued initial condition in the state variables, and a finite set of state transitions. These three components are very similar to ASTRAL variable, initial, and transition clauses. Two special variables are defined for every TTM. A natural number t expresses the current time in the system similar to Now in ASTRAL. The next transition variable n expresses the transition in the TTM that will next occur to bring the TTM into its successive state. The variable n is well defined because TTMs are based on an *interleaving semantics*. That is, the behavior of the system is a sequence of global states and only one transition is allowed between states. Note that multiple transitions can occur at the same "time" as defined by t, but not between two consecutive states.

A state is a mapping from variables (including t and n) to appropriately typed values. A transition is a tuple consisting of an *enabling condition*, a *transformation function*, and a lower and upper time bound. This is essentially equivalent to an entry assertion, an exit assertion, and a duration, respectively, in an ASTRAL transition. One difference is that TTM upper bounds can be infinite, which simulates external events that may occur at any time. Two special transitions are defined for every TTM: the tick transition (true, [t: t + 1], -, -) and the initial transition (true, [], 0, 0). No time bounds can be given for the tick transition because all other time bounds are defined relative to the number of times tick has occurred. Thus, it is not well defined for tick to have any such bounds.

Tick is the only transition that is allowed to modify t. Initial is the next transition (i.e. n = initial) at system initialization. No other transition is allowed to have lower and upper time bounds both equal to zero. If a transition is enabled at time $t_0$ (i.e. enabling condition holds in some state with $t = t_0$) and continues to be enabled up to some time between $t_0$ + lower and $t_0$ + upper, the transition can instantaneously occur and the new state becomes the previous state changed by the transformation function and n set to that transition. If $t_0$ + upper is reached and the enabling condition still holds, the transition must occur. Although the "instantaneously" seems different from ASTRAL, in actuality it is similar because the changes described in an ASTRAL transition's exit assertion occur atomically and instantaneously at the time the transition ends. A difference, however, is that in TTM, the enabling condition must hold for the entire time up until its transformation function is invoked, whereas in ASTRAL, the entry assertion may not necessarily hold after the transition has started but before it has completed. This property of TTMs allows preemption to be specified, which is not possible in ASTRAL.

Two TTMs can communicate over a shared channel c with c!m denoting that a message m is sent over c and c?r denoting that m is received over c and is assigned to r. TTMs can be composed with the $\|$ operator. The composite system has all the transitions of the component systems. Transitions that have the same name become *shared transitions*. Given two transitions ($e_1$, $h_1$, $l_1$, $u_1$), ($e_2$, $h_2$, $l_2$, $u_2$), the shared transition becomes ($e_1 \wedge e_2$, "$h_1$; $h_2$", $\max(l_1, l_2)$, $\min(u_1, u_2)$). Transitions on the opposite ends of a channel are combined similarly, with the addition that r is assigned m in the new transformation function. In essence, individual TTMs lose their "individuality" during composition and become a single set of transitions, whereas in ASTRAL, each process still retains its modularity.

A legal TTM trajectory is an infinite sequence of states that satisfies a number of properties: the initial state holds with n = initial, a transition T occurs within its lower/upper bounds when enabled and its transformation function holds when n = T, tick occurs infinitely often, and finally, only a

finite number of states can occur between ticks. RTTL is interpreted with respect to the set of legal trajectories in a TTM. RTTL is Manna-Pnueli temporal logic without past operators and with the addition of the two special TTM variables n and t. A TTM/RTTL specification is thus the combination of a TTM describing system behavior and an RTTL formula describing the critical requirements.

TTM/RTTL specifications are verified using a deductive proof system based on the proof system of Manna-Pnueli logic [MP 92]. The system is composed of a set of axioms that are assumed to hold in the logic and a set of inference rules to derive new theorems based on axioms and existing theorems. In the TTM/RTTL system, the TTM specification is used to generate new theorems for each transition by inference rules based on its entry and exit conditions and its upper and lower bounds. The attempt is then made to prove the RTTL formula as a theorem. If successful, the properties of the formula hold in the TTM.

*3.2.1.4. Software Cost Reduction Requirements Notation*

A Software Cost Reduction (SCR) requirements notation [AG 93] specification contains a set of *modeclasses* and a set of *environmental assumptions*. Each modeclass is a state machine consisting of a number of modes of operation. The set of modeclasses represents a concurrent set of state machines. Each modeclass has exactly one mode active at any given time. A modeclass is defined by a set of modes, an initial mode, and a set of mode transitions. The collection of the initial modes of all modeclasses describes the initial state of the whole system. A transition is defined with a source mode, a destination mode, and a triggering event. Events are constructed from *environmental conditions*, *state conditions*, and *timing conditions*. Environmental conditions are propositions about the state of the environment. State conditions are propositions about the current mode of a modeclass and are written in the form In(m), where m is a mode. Finally, timing conditions are written in the form In(m, t), where m is a mode and t is a time. The time domain can be dense or discrete, but the model checker used for verification only supports discrete time, so the discrete case will be discussed. In(m, t) holds if the system has been in mode m for at least the past t time units. There are three types of events that can be defined for a transition. The *primitive event* @T(A), where A is a condition, occurs at time t if A is false at time t - 1 and true at time t. Similarly, @F(A) indicates a change in A from true to false. A *conditioned event* @T(A) WHEN [B], where A and B are conditions, occurs at time t if @T(A) occurs at time t and B is true at time t - 1. Combinations of @F and ¬B can be used to indicate that @F(A) must occur at t and B must be false at time t - 1, respectively. The last type of event is the *timeout event* @TO(In(m, t)). This event occurs if

@T(In(m, t)) occurs at time t - 1 and @F(In(m)) occurs at time t. That is, a timeout event specifies a deadline by which the system must move out of mode m. Environmental assumptions place constraints on the times that environmental conditions hold such as ordering and mutual exclusion between conditions.

Properties of SCR specifications are written in a subset of CTL, which was discussed in section 3.1.2. In this subset, formulas can only contain universal path quantifiers or a single existential path quantifier evaluated in the initial state of the system. In addition, the only propositions that may be used in formulas are conditions that appear at some point in the SCR specification as triggering events or environmental assumptions. That is, the only timing properties that can be expressed are those incorporating In(m, t), where t is from the finite set of times that have actually appeared in the SCR specification. CTL formulas are interpreted over the *compact timed reachability graph* produced from the SCR specification. Each node in the graph consists of a set of modes that the system can be in and a deadline on the maximum amount of time that the system can spend in the composite mode before taking some transition to a new composite mode. In addition, the modes in each composite mode are labeled with the minimum and maximum times that the system can spend in each mode waiting for another mode to change to bring the system into the composite mode. Transitions connect nodes and are labeled with a delay and a deadline bounding the times the transition can fire. Each node in the compact graph represents many possible system executions depending on the times external events occur and the times transitions fire. Although this serves to reduce the size of the graph, which often becomes large in timed systems, it requires the CTL syntax restrictions discussed above to preserve the soundness of verification over the graph. Verification is performed by model checking the formula over the compact graph. Since the graph is finite, all paths in the graph are checked for the appropriate properties similar to other model checking approaches discussed in this chapter.

The SCR approach is similar to that of ASTRAL, where transitions are enabled based on the occurrence of external events and the internal state of the system. Also, unlike many of the other state machine approaches, which simulate the external environment as an explicit automaton in the system, SCR specifications list environmental assumptions that must be met for proper system behavior like the environment clause in ASTRAL. SCR, however, is less expressive than ASTRAL. Timing requirements are limited to the use of In(m, t) conditions whereas in ASTRAL, the explicit clock and first-order logic allow much more sophisticated requirements to be expressed. Also, the

use of typed state variables in ASTRAL allow transitions to have very specific effects unlike SCR transitions, which can only specify mode changes.

*3.2.1.5. Timed Büchi Automata*

A timed Büchi automaton (TBA) [AD 90] is a simple timed extension to classic finite-state machines. A Büchi automaton (BA) is defined by a tuple (A, S, $S_0$, E, F), where A is the input alphabet, S is the set of states, $S_0$ is the set of initial states, E is the set of edges (i.e. transitions), and F is the set of accepting states. An edge is a tuple (s, a, s′), where if the system is in state s and encounters input a, it moves to state s′. A BA accepts a set of *traces*, where a trace is an infinite sequence of letters in the input alphabet. If ($a_0$, $a_1$, ...) is a trace accepted by a BA M, then there is an infinite sequence of states of M ($s_0$, $s_1$, ...) such that for states $s_i$ and $s_{i+1}$, M has an edge ($s_i$, $a_i$, $s_{i+1}$) and the accepting states of M appear infinitely often in the trace. The input alphabet is used to represent observable events in the system and the accepting states distinguish fair executions. A TBA is a BA but with a finite set of clocks and modified edges. Each clock holds a non-negative real value, corresponding to a value in the time domain. A TBA edge is a BA edge but with two additional components: the set of clocks R to reset when the edge is taken and a boolean combination of clock formulas in the form `x r_op c`, where x is a clock, c is a constant and r_op is a relational operator. A TBA edge is taken when the system is in state s, the input is a, and the clock predicate holds. The system then moves to state s′ and resets all clocks in R. A TBA accepts a set of *timed traces*, where a timed trace is a trace with each event associated with a time such that the first time is zero, the times increase strictly monotonically, and for any non-negative real t, there is a time in the sequence greater than t. If (<$a_0$, $t_0$>, <$a_1$, $t_1$>, ...) is a timed trace accepted by a TBA M, then there is an infinite sequence (<$s_0$, $v_0$, $t_0$>, <$s_1$, $v_1$, $t_1$>, ...), where $s_i$ is a state and $v_i$ is a mapping from clocks to non-negative reals, such that the accepting states of M appear infinitely often in the sequence and for all i, M has an edge ($s_i$, $a_i$, $s_{i+1}$, R, P), P($v_i$) holds, and $v_{i+1}(r) = 0$ for r in R and $v_{i+1}(r) = v_i(r)$ otherwise.

Verifying that a system modeled by a TBA M meets its critical requirements specified as a TBA P is performed by proving that the timed traces of M are a subset of the timed traces of P. Inclusion is shown by checking whether the intersection of the timed traces of M and the timed traces of the complement of P is empty. Intersection and emptiness are decidable for TBAs. The emptiness algorithm relies on the fact that for two tuples <$s_i$, $v_i$, $t_i$>, <$s_{i+1}$, $v_{i+1}$, $t_{i+1}$> of a timed trace, if $s_i = s_{i+1}$, the integral parts of all clocks agree in $v_i$ and $v_{i+1}$, and the fractional parts of all clocks have the same ordering in $v_i$ and $v_{i+1}$, then the two tuples are essentially equivalent. It is also the case that for each

clock, there is a maximum value, statically determined from the clock predicates in the edges of M, after which the integral value of the clock no longer affects the behavior of M. Based on these two facts, there is a finite set of equivalence classes for each state such that all timed behaviors of the system when in that state are represented by a class in the set. The emptiness of a TBA M is then determined by constructing an untimed BA M′ whose states are the equivalence classes of all states of M with edges between states based on the edges and the corresponding reset sets of M. Then the emptiness algorithm for BAs is used.

Although emptiness of TBAs is decidable, TBAs are not closed under complementation so the inclusion problem is undecidable. Instead of specifying system properties as a TBA, deterministic timed Muller automatons (DTMA) are used. A TMA is similar to a TBA, but the accepting set F is a set of sets of states instead of a simple set of states. A TMA accepts a timed trace only if some set of states in F appears infinitely often in the trace. TMAs accept the same class of timed languages accepted by TBAs so TMAs are no more suitable for automatically verifying properties than are TBAs. DTBAs and DTMAs allow only a single edge per state to be defined for any input symbol and are strictly less powerful than their nondeterministic counterparts. DTMAs are preferable over DTBAs because they are both more powerful and are closed under complementation. Thus, verifying that a system modeled by a TBA meets its critical requirements specified as a DTMA is decidable.

### 3.2.2. Graphical

Graphical notations do not have any advantage in expressive power over textual notations. These notations have been explored because it is usually easy for designers to look at a graphical specification and immediately grasp the control and data flows of the system while the same cannot always be said of textual specifications.

Most graphical formalisms are based on state transition diagrams. Basic state transition diagrams are unsuitable for concurrent systems because although concurrency can be represented by creating a state for each possible combination of concurrent elements, the number of states increases exponentially with each new element added. Thus, each of the formalisms presented have solved this problem in some way to represent concurrency concisely.

#### *3.2.2.1. Petri Nets*

A Petri net (PN) [Pet 62] consists of a set of *conditions*, denoted by circles, and a set of *transitions*, denoted by bars. A set of directed arcs connect conditions to transitions and vice-versa. When an arc connects a condition to a transition, the condition is said to be an *input* of the transition. Likewise,

when an arc connects a transition to a condition, the condition is called an *output* of the transition. A PN is initialized with a set of tokens, denoted by dots placed in condition circles, which represent the conditions that hold in the initial state. The system then evolves by the occurrence of transitions. A transition is enabled whenever all its input conditions hold (i.e. they all have at least one token) and fires at some time after that assuming it has not been disabled by the firing of another transition. When a transition fires, one token is removed from each input and a new token is placed in each output.

The PN model is adept at modeling concurrency, asynchrony, and nondeterminism. Transitions occur independently of one another with no need for synchronization provided they do not share input conditions. Nondeterminism is captured by sets of transitions that all require the same input condition to hold in order to fire. When the input condition gains a token, any one of the enabled transitions may fire, disabling the others. This model, however, cannot express hard real-time requirements such as timeouts and delays, nor qualitative properties such as fairness and liveness. Since PNs represent some systems so naturally, several extensions to the basic PN model have been proposed to incorporate the intuition of PNs while allowing more sophisticated systems to be represented.

*3.2.2.1.1. Time Petri Nets*

Time Petri nets (TPN) [MF 76] are PNs with one modification. Each transition T is associated with a minimum firing time $t_{min}(T)$ and a maximum firing time $t_{max}(T)$, where $t_{max}(T)$ is possibly infinite. T can nondeterministically fire at any time between $t_{min}(T)$ and $t_{max}(T)$ from when it was first enabled, provided it has been continuously enabled for the entire period. This is unlike an ASTRAL transition, which must fire as soon as it is enabled and its processor is idle. When T has been enabled for $t_{max}(T)$ time units, it must fire, provided it is not disabled at the same instant by another transition firing. The time domain is dense so T can fire at any instant between $t_{min}(T)$ and $t_{max}(T)$. The TPN in figure 3.2.2.1.1 depicts a system consisting of a sensor and a receiving process that uses the sensor data, which must synchronize to exchange data. The two processes are initially involved in local computations at SLC and RLC. Periodically, every $t_s$ time units, the sensor samples the environment ($T_{sample}$) and obtains data to give to the receiver (S). If the receiver is ready to receive (R), $T_{rendezvous}$ fires immediately and the system reverts to the initial state. If the receiver has not requested a value ($T_{request}$) within $t_{to}$ time units, however, the sensor aborts the send ($T_{timeout}$) because the data has become stale and repeats its computation. The receiver needs a value to proceed with its computation so must block indefinitely waiting for the sensor to send.

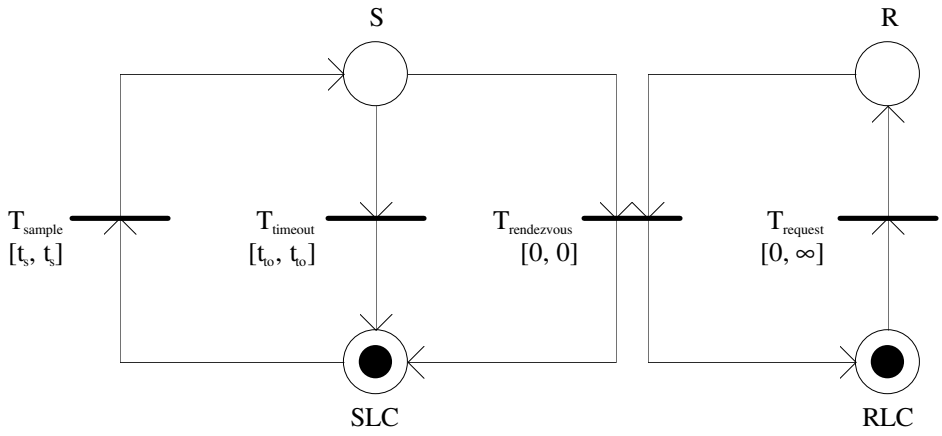



Figure 3.2.2.1.1: A TPN

A PN is *bounded* if the number of tokens in any place in any reachable marking is less than some constant. The boundedness problem is undecidable for TPNs. There are decidable conditions, however, that are sufficient for boundedness. In [BD 91], some of these conditions are presented along with a system for analyzing properties of bounded TPNs. A system is represented by a tree of *state classes*. A state class is a marking of the TPN along with a system of inequalities based on $t_{min}$ and $t_{max}$ as well as the transitions that fired to reach the marking. A state class describes the possible intervals in which each transition can fire in a particular marking. For example, from [BD 91], $1 \le t(T_1) \le 4$, $2 \le t(T_2) \le 3$, $1 \le t(T_3) \le 6$ describes a state class in which $T_1$, $T_2$, and $T_3$ are enabled and must fire between times 1 and 4, 2 and 3, and 1 and 6, respectively. It is assumed for these equations that the current time is 0. The root of the tree of state classes is the initial state class. For each transition T that is enabled in the initial marking, the equation $t_{min}(T) \le t(T) \le t_{max}(T)$ is in the initial state class.

The number of children of each class is less than or equal to the number of enabled transitions in its associated marking. Each child class is derived by allowing one enabled transition T to fire (if its $t_{min}$ is less than or equal to the $t_{max}$ of all the other enabled transitions). The possible firing time of T (denoted $pft(T)$) is described by the equation $t_{min}(T) \le pft(T) \le \min(t_{max}(E))$ for all enabled transitions E in the TPN. That is, T cannot be the next transition to fire at a time past the $t_{max}$ of some other transition, because by the semantics, all transitions must fire at or before $t_{max}$. For each transition E that remains enabled in the new class, its firing inequality must be adjusted to account for the time that has advanced because of the firing of T. Let the firing inequality for E be $l \le t(E) \le u$. The new lower bound for E becomes $\max(l - pft_{max}(T), 0)$ because the soonest possible time that E can fire is when T fires at its latest possible time. Likewise, the new upper bound becomes $u$ -

$pft_{min}(T)$ because the latest possible time that E can fire is when T fires at its earliest time. For example, in the situation described above, the new state class resulting from the firing of $T_1$ would be $0 \leq t(T_2) \leq 2$, $0 \leq t(T_3) \leq 5$ plus the inequalities from any transitions that become enabled with the firing of $T_1$. This relaxing of timing constraints requires that additional inequalities be added for every pair of transitions that remain enabled so that the maximum separation between firing times remains the same. For instance, the original maximum separation between $T_2$ and $T_3$ above is 3, but in the new constraints it is 5, thus $t(T_3) - t(T_2) \leq 3$ and similarly $t(T_2) - t(T_3) \leq 2$ must be added.

When a TPN is bounded, the state class tree constructed from that TPN is finite. Thus, it is possible to check the tree for properties such as liveness or to verify assertions made in a different language over it. The next section describes a dual-language approach incorporating TPNs and TRIO. Although the verification system presented is a deductive proof system, another possibility would involve the work just presented.

*3.2.2.1.2. Time Petri Nets and TRIO*

In [FMM 94], a formal semantics is defined for TPNs in terms of TRIO so that properties can be proven using the deductive proof system given. Two TRIO predicates are introduced to express properties of TPNs. Fire(r) is true if transition r fires at the current instant. TokenF(r, s, d) is true if transition r fires at the current instant producing a token that causes transition s to fire d time units into the future. From these two predicates, it is possible to express properties such as freedom from deadlock and periodicity requirements. A complete system is specified as a TPN and a TRIO formula. The TPN is used to generate a set of theorems from the axiom system that are assumed to hold. Then the TRIO formula is proved as a theorem by using axioms and inference rules. Tokens in TPNs are anonymous objects so it is not possible to express data dependencies with them. The next section discusses a timed extension of *colored Petri nets*, which use colored tokens to represent data values.

*3.2.2.1.3. Interval Timed Colored Petri Nets*

A colored Petri net (CPN) is a PN in which values (i.e. colors) are associated with each token in the net. Depending on the colors of the input tokens, a transition may produce a variety of different output tokens. This extension allows many types of systems to be described more succinctly since a variety of situations can be represented by utilizing token colors rather than adding additional conditions and transitions to the net.

Interval Timed Colored Petri Nets (ITCPN) [Aal 93] are a combination of CPNs and TPNs. Each token in the system is described by a triple <p, c, t>, where p is the place where the token currently resides, c is a color, and t is a timestamp. Like TPNs, each transition in the ITCPN is associated with a time interval. Unlike a TPN interval, however, which describes the minimum and maximum times required for a transition to be enabled before it can fire, an ITCPN transition fires immediately and its associated interval describes its duration. In addition, each output arc of the transition can be associated with a different interval. When a transition fires at some time ft, each output token of an arc A is nondeterministically given a timestamp in the interval $[ft + t_{min}(A), ft + t_{max}(A)]$. A transition can only fire when the proper number of input tokens with the proper colors are present and all the input tokens have a timestamp less than or equal to the current time. That is, any tokens with a timestamp greater than the current time are actually still "in transit" because of the firing duration of the previous transition. Thus, they cannot be used until they have "arrived". Like ASTRAL transitions, ITCPN transitions are *eager* to fire meaning as soon as they are able to fire, they do fire. After firing, the *transition function* describes the numbers and colors of tokens generated to the output places. ITCPNs follow the interleaving approach in which system evolution can be represented by an infinite state sequence (a state being a set of tokens) where only one unique transition fires to bring the system from one state to the next. ASTRAL, on the other hand, follows the noninterleaving model in which multiple transitions may fire at the same time as long as they are in different processes.

Verification of ITCPNs employs *reachability analysis*. A *reachability graph* is a tree of states with the initial state of the system as the root. From this graph, properties can be verified by checking all paths in the graph for the desired conditions. For example, to prove that a state is always reached, all paths in the tree are checked for the presence of that state. Obviously, this technique can only be applied when the reachability graph is finite. In ITCPNs, two states are different even if only a single component of any token is different. Since an output token can be timestamped with any time in the output arc's interval, any transitions with an output arc in which $t_{min}$ does not equal $t_{max}$ has an infinite number of immediate successors. Thus, to apply reachability analysis, the infinite reachability graph must be represented finitely. To achieve this, [Aal 93] defines an alternate semantics for ITCPNs. Instead of timestamping tokens with an explicit time value, tokens are associated with time intervals. Thus, the firing time of a transition is actually an interval instead of a specific time. The earliest time a transition can fire is the earliest time that all inputs tokens can arrive (i.e. the maximum lower bound among all input tokens). The latest time a transition can fire is the latest time that any input token can arrive (i.e. the maximum upper bound among all input

tokens). Thus, if transition T fires at time interval [$ft_{min}$, $ft_{max}$], all output tokens of output arc A of T have a time interval component of [$ft_{min} + t_{min}(A)$, $ft_{max} + t_{max}(A)$]. In this new definition, a single state actually represents an infinite number of states in the original ITCPN semantics. The modified semantics are sound with respect to the original semantics. That is, if $s_2$ is directly reachable from $s_1$ in the old semantics, then there exists states $s_1'$ and $s_2'$ in the new semantics such that $s_1'$ covers $s_1$, $s_2'$ covers $s_2$, and $s_2'$ is directly reachable from $s_1'$. The modified semantics are not complete, however, because the use of intervals in tokens causes timing constraints between tokens to be relaxed. For example, if two output arcs $A_1$ and $A_2$ of transition T have intervals [1, 2] and [3, 4], then the maximum separation between the tokens produced by $A_1$ and $A_2$ in the old semantics is 3. In the new semantics, the firing time of T is an interval (e.g. [0, 1]), so in this case, the maximum separation is 5. Thus, it is not possible to prove the presence of properties (e.g. reachability) because a state that has a property in the new semantics may not be possible in the old semantics. Thus, the reduced reachability graph is mainly used to prove the absence of properties.

*3.2.2.2. Statecharts/STATEMATE*

STATEMATE [HLN 90, i-L 91a] is a commercial tool for developing real-time systems in which a system is described in three different approaches: the *structural view* specifies modules and communication links, the *functional view* specifies capabilities and the flow of information, and the *behavioral view* specifies control and timing. These three views are represented by three different languages: *module-charts*, *activity-charts*, and *statecharts*. Statecharts specify the real-time aspects of a system, but it is worthwhile to look at module-charts and activity-charts as well because all three are interrelated and together give a complete description of a system.

Module-charts depict how the actual system will be implemented. A module-chart is a hierarchy of modules (drawn as rectangles) with a unique root module, which are connected by arrows depicting physical data links. A module is an ancestor of all modules that it contains. Environment modules are placed outside the root module to depict components that are not part of the system but which interact with internal components. Storage modules represent physical memory and disk devices. Execution modules are the most common components of module-charts and show subsystems that perform specific tasks. Each execution and storage module may be associated with an activity-chart to describe its functionality in more detail. The upper left portion of figure 3.2.2.2 is a module-chart depicting a railroad system, which consists of a gate, a CPU, and two external sensors. Physical data links connect the sensors to the CPU and the CPU to the gate.

Activity-charts depict the functions (i.e. capabilities) of each subsystem and how information flows between them. Similar in structure to module-charts, an activity-chart is a hierarchy of activities with arrows connecting them. The arrows represent information flow with solid lines indicating data flow and dotted lines indicating control flow. Flow lines can originate and terminate at or within any activity. Each activity and flow line is associated with a *form*, which textually describes it in more detail. For example, flow line forms describe the type of information being transported (e.g. events, conditions, data items, etc.). Also like module-charts, activity-charts may contain a number of different types of activities such as environment activities and data-stores. Each activity can contain a single control activity, shown as a rounded rectangle. Whereas "regular" activities represent simple functions that transform inputs to outputs regardless of content (e.g. multipliers), control activities represent complex decision procedures whose behavior depends on input values and timing constraints. Control activities are described by statecharts. The upper right portion of figure 3.2.2.2 is an activity-chart depicting the activities of the CPU, which consists of two simple activities: lower and raise, a control activity: CPU_control, and three external activities: gate, sensor_in, and sensor_out. Data flows from the sensors to the control, which in turn sends control information to the lower and raise activities to transform into actual parameters to be given to the gate.

Statecharts [Har 87, i-L 91b] are an extension of conventional state-transition diagrams. Each transition has a *triggering condition* and an optional *action* associated with it. A triggering condition, similar to an ASTRAL entry assertion, is a boolean combination of event occurrences (e.g. entering/exiting states, changes in data values, timeouts) and expressions in data values and current states of the system. An action, similar to an ASTRAL exit assertion, assigns values to variables, schedules other actions, signals activities to begin and end execution, etc. When the triggering condition holds, the transition is instantly taken and the action performed. Statecharts are built hierarchically similar to module-charts and activity-charts. In addition to simple containment, statecharts can also be composed using *and-composition*, denoted by a dashed line splitting a state into a number of portions, called *orthogonal components*. If a state does not use and-composition, it is called an *or-state* and only one of its immediate children (i.e. states) can be active at any time. Each orthogonal component of an *and-state* behaves like an or-state. That is, there is one immediate child active in each component of the and-state and all components operate in parallel. A transition is not allowed to connect any states in different orthogonal components. A conventional state transition diagram may require exponentially many more states than a statechart using and-composition because all possible combinations of parallel states must each be represented by a unique state. An or-state can have one child designated as the default state, which means that any time a

transition ends at the or-state, the default state is entered. An and-state can designate one default state for each orthogonal component. A state can also contain a *history* entrance where the system will resume execution in the state last enabled in the parent. Whenever any state inside an and-state is directly enabled by a transition, the default states in the other orthogonal components are implicitly enabled. A similar situation occurs when one of the states is disabled.

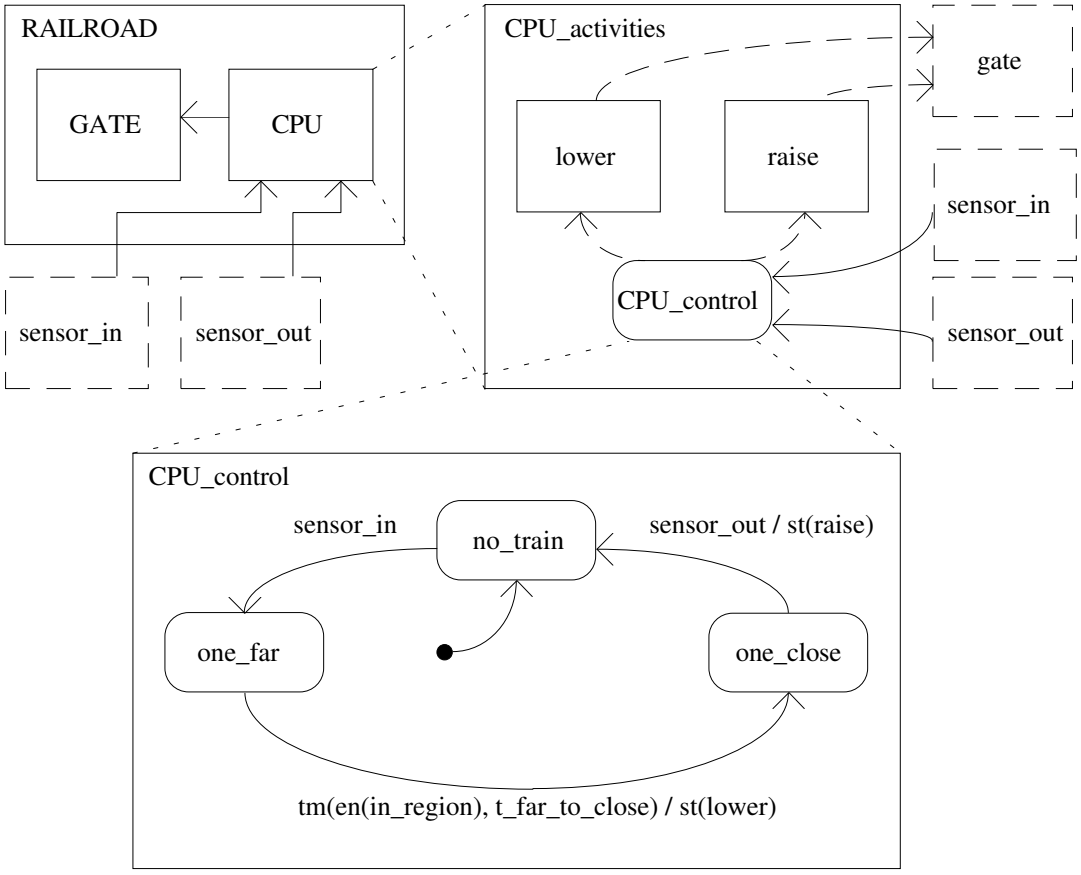



Figure 3.2.2.2: A STATEMATE specification fragment

Time is advanced after every *step*. A step is a change from one set of states and data values to another, which depends on events and data values from the previous system status. The use of time in specifications is limited to timeout events and scheduled actions. Tm(e,t) is a new event that occurs t time units after the last occurrence of e. Every time e occurs, the timer is reset. Sc!(g,t) schedules action g to occur t time units from the current time. Other real-time behaviors such as periodic execution can be modeled by adding additional states to the system and using scheduled actions. The lower portion of figure 3.2.2.2 is a statechart depicting the behavior of CPU_control in CPU_activities, which consists of three states: no_train, one_far, and one_close, with no_train being

the initial state. The states are cyclically connected by three transitions. Only the transition between one_far and one_close has a timing requirement. This transition fires when t_far_to_close time has passed since a train has entered the region and upon firing, signals the lower activity to begin execution.

Formal verification of STATEMATE specifications is not yet supported. A simulator is available to test specifications. The simulator can dynamically determine reachability, nondeterminism, deadlock, and usage of transitions, although for large systems, these procedures may become highly unmanageable due to their brute-force approach. Some of the simulator overhead can be avoided by automatically generating a rapid prototype of the specified system in C or Ada code. The need for a well defined verification approach along with more powerful timing expressability has prompted the development of a new language based on statecharts, called Modechart, which is discussed in the next section.

*3.2.2.3. Real-Time Logic/Modechart*

*3.2.2.3.1. Real-Time Logic*

Real-Time Logic (RTL) [JM 86] does not have an explicit clock variable like Now in ASTRAL. Instead, RTL uses the *occurrence function* on various *events*. Events are markers in time for significant occurrences in the system. There are four types of events. The first two types are the start and stop times of *actions*. Actions are "schedulable units of work" that take non-null duration. For an action A, $\uparrow$A denotes the event occurring at the time A starts and similarly $\downarrow$A denotes the event occurring at the time A stops. A state attribute is a boolean assertion about the system. A *transition event* (S := T) denotes the event occurring when state attribute S becomes true. Finally, *external events* such as button pushes and interrupts are denoted $\Omega$event.

The occurrence function @(E, i) denotes the time of the ith occurrence of event E from system initialization. RTL is akin to the logic used in ASTRAL except that the occurrence function in RTL starts at the first occurrence in the system, while ASTRAL $Call_i$, $Start_i$, and $End_i$ start at the last occurrence. @($\Omega$T, i) is similar to $Call_i$(T, t), @($\uparrow$T, i) is similar to $Start_i$(T, t) and @($\downarrow$T, i) is similar to $End_i$(T, t). RTL is defined over a discrete time domain, while ASTRAL is defined over both discrete and dense time domains.

In [JM 86], RTL formulas are interpreted over an *event-action model* that describes events and actions in the system as well as timing constraints such as when <event>, execute <action> with deadline = <time>, separation = <time>. The event-action model is defined in terms of RTL so to

prove an RTL formula over a model, a set of theorems is generated from the RTL definition and then inference rules are used to prove the formula as a theorem. More recently, RTL has been used in the definition of Modechart.

*3.2.2.3.2. Modechart*

Modechart [JM 94] is a graphical language similar to statecharts used to specify system behavior in a dual-language approach with RTL. A *mode* is similar to a condition in a PN. At any given instant, the system is in some set of modes, similar to the set of conditions that hold (i.e. have tokens) in the PN model. A system is modeled as a hierarchy of modes with a single root mode, which may or may not have a set of modes as children. Every mode, which is depicted as a box, is either a *primitive mode* (i.e. one that has no children), a *serial mode*, or a *parallel mode*. A mode that is contained in another mode is the *child* of the outer mode. If no other child of the outer mode contains the inner mode, the inner mode is an *immediate child* of the outer mode. Modes are classified as serial modes and parallel modes corresponding to or-states and and-states in statecharts, respectively. When a serial mode is entered, only the one immediate child mode designated as the *initial mode* is activated and only one of its immediate children will be enabled at any one time with the exception of the instant a transition between two modes occurs. When a parallel mode is entered, all of the mode's immediate children are entered and continue to operate concurrently. The modechart in figure 3.2.2.3.2 illustrates many of the definitions. M0 is the root node. M2, M3, and M4 are primitive modes. M2 and M3 are immediate children of M1, but are not immediate children of M0. M4 is the initial model of M0.

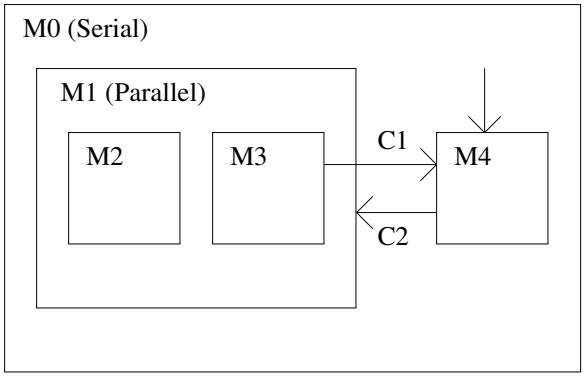



Figure 3.2.2.3.2: A modechart

All modes not operating in parallel can be connected with arrows that depict mode transitions. Each transition is associated with a disjunction of *triggering conditions*, which are modified RTL formulas

involving times of events and truth values of predicates, and *lower/upper bound restrictions*, which specify the delay before the transition can be taken and the deadline by which it must be taken. Whenever the disjunction holds, the transition is taken and the system enters a new set of modes. If more than one transition can be taken at a given instant, precedence is considered, where transitions originating from modes higher (i.e. closer to the root) in the hierarchy have precedence over those originating from lower modes. Transitions are instantaneous so at the instant a transition is taken, the system is in both the modes it connects. The event associated with a transition is denoted $M_i$-$M_j$ so for example, @($M_i$-$M_j$, 1) is the first time a transition between modes $M_i$ and $M_j$ occurred. Every mode can be associated with an *action* of non-null duration that is performed when the mode is first entered via a transition. An action is an RTL formula that describes the values of state variables after the transition is taken.

All modes that a transition arrow crosses out of are disabled when the transition is taken. When a parent mode is disabled, all its children are also disabled. Thus, if an arrow leaves one child of a parallel mode and proceeds to leave the parent mode, all siblings of that child as well as all of their children are disabled when that transition is taken. In figure 3.2.2.3.2, when M1 is exited via C1, then both M2 and M3 are disabled. The events of entering and exiting modes are denoted by $M_i := T$ and $M_i := F$, respectively. When the system is initialized, the root mode is entered and the system evolves by the enabling and disabling of modes as described above.

A transition-action pair is closely related to an ASTRAL transition with the Modechart transition condition being the entry assertion and the action formula being the exit assertion. The difference is that Modechart transitions are instantaneous and not all modes are associated with actions to provide the non-null duration of ASTRAL transitions. Another difference is that Modechart actions can be preempted, unlike ASTRAL transitions, which are atomic and uninterruptable. Atomicity can be simulated by manipulating the structure of the Modechart specification so that action modes are not explicitly or, more notably, implicitly disabled. Modechart models concurrency as a partial ordering of events (i.e. two events are concurrent if neither precedes the other one), whereas in ASTRAL, events are totally ordered based on the global clock and every individual process instance is operating in parallel to the others.

The semantics of Modechart are defined in terms of RTL similar to the definition of the event-action model. Thus, Modechart specifications can be verified in a similar manner with a deductive proof system. More interesting is the automated verifier discussed in [JS 88, Stu 90, YMS 95]. In this approach, a finite computation graph is built from the specification, which represents infinite

behaviors of the system. A node in the graph represents a system state and is labeled with the events that occur in that state. Edges represent state transitions and are labeled with integers. Positive weighted edges depict delays between states while negative edges depict deadlines. Given that RTL satisfiability is undecidable, it is not possible to verify arbitrary RTL formulas. Formulas must be in one of a number of restricted forms. From these forms, it is possible to verify properties such as reachability, starvation, and separation.

*3.2.2.4. Hierarchical Multi-State Machines*

Hierarchical Multi-State (HMS) machines [GF 88, FG 89, GF 90] are extended state transition diagrams similar to statecharts and modecharts. A specification consists of a set of states, an initial marking, and a set of transitions. Transitions can be designated as either deterministic or nondeterministic. Deterministic transitions must fire as soon as they are enabled, whereas nondeterministic transitions may fire, but are not forced to do so. A transition is defined by its *primaries*, *controls*, and *consequents*. The primaries and consequents are possibly empty subsets of system states. A control is a tuple $(s, t)$ or $(\neg s, t)$, where s is a state and t is an interval in the form $<t_1, t_2>$, $[t_1, t_2]$, or $<t_1, t_2>!$, with $t_1 \leq t_2 \leq 0$. The times are negative representing the history of the system where 0 is the current time. The control $(s, <t_1, t_2>)$ holds if the system was in state s at sometime in the interval between $t_1$ and $t_2$. Similarly, $(s, [t_1, t_2])$ holds if the system was in state s at all times in the interval. Finally, $(s, <t_1, t_2>!)$ holds if the system was in state s sometime in the interval, but was not in state s immediately preceding the beginning of the interval (i.e. at $t_1 - 1$). Controls in the form $(\neg s, t)$ are similarly defined with the requirement that the system is not in state s at the respective times. A transition is enabled when the system is currently in all the states of the primaries and all controls hold. After the transition fires, the system moves out of the states of the primaries and moves into the states of the consequents. HMS machines are noninterleaving so at each moment in time, all enabled deterministic transitions fire and a possibly empty subset of the enabled nondeterministic transitions fire. Thus, an HMS execution is a sequence of sets of states.

Like statechart and modechart specifications, HMS machines are hierarchical, where each state is refined by a lower level HMS machine. HMS machines can also be multi-level in a different sense. An HMS machine may have a number of different executions depending on the times that its nondeterministic transitions fire. Although such general behavior may be acceptable for the initial specification of a system, it is often desirable to be able to limit a system to specific executions in the implementation phase. A *policy HMS machine* restricts the nondeterministic behavior of lower level (possibly policy) HMS machines. A policy machine H′ for an HMS machine H consists of a subset of

the states of H and a set of *policy transitions*. While a standard transition is instantaneous, a policy transition may take a number of time units to complete. A policy transition is defined similarly to a standard transition but instead of one set of controls, there are three sets of controls for the beginning, the middle, and the end of firing. A policy transition is enabled when the system is in all the states of its primaries and all its beginning controls hold. One time unit after the transition begins firing, it continues firing while its middle controls hold and completes firing when its end controls hold. After the policy transition stops firing, the system moves into the states of its consequents like a standard transition. Policy HMS machines serve to reduce nondeterminism in specifications, similar in purpose to ASTRAL transition selection clauses. The difference is in the form of nondeterminism. In HMS machines, there is no restriction about the number of transitions that can fire at any instant, so nondeterminism arises strictly from the choice of firing or not firing nondeterministic transitions. In ASTRAL, however, transitions must always fire when they are enabled, but only a single transition can fire per process so nondeterminism arises from the choice of which transition from several enabled transitions actually fires.

A multi-level HMS specification consists of a set of HMS machines where the lowest level machine is a basic machine and each higher-level machine is a policy machine for the previous level. To prove a liveness property, such as a particular state being reachable, it is necessary to show that the requirement is feasible with respect to the constraints that each policy machine places upon its predecessor. An execution plan for a basic HMS machine is a sequence of sets of transitions. A plan for a policy machine is similar except the sequence contains three sets of transitions for those transitions that begin firing, those that continue firing, and those that finish firing at each point. A restricted verification approach is presented in [GF 90], which given a plan for reaching a state in the lowest level basic machine, can determine whether or not the plan is feasible within the constraints proscribed by a higher-level policy machine. First, a time delay variable is added between each step in the given plan. Then, a set of precondition and postcondition laws are applied to obtain the set of conditions that are true before and after each step in the plan. Finally, these conditions are used to define a system of inequalities in terms of the time variables, which is solved to determine the actual delay between each step of the plan. Negative solutions represent reorderings of transitions. If no solution is possible, the state cannot be reached with the given plan under the constraints of the policy machine.

Safety properties can be incorporated directly into HMS machines by adding additional states that are only reachable when the properties are violated. For example, for the property $\Box(s_1 \mid s_2 \mid s_3)$, a

failure state F is added, accessible only through a transition with primaries $\{\neg s_1, \neg s_2, \neg s_3\}$. Thus, F can only be reached when the system is not in any of the three states, thereby indicating unsafe behavior. Verification of safety properties is therefore reduced to demonstrating the unreachability of unsafe states. In [FG 89], a method is presented for proving unreachability in HMS machines based on correctness-preserving transformations. The transformations include control addition, transition deletion, and delay sharpening. To prove that a state is unreachable, the machine is transformed into an equivalent machine in which the unreachability is obvious. For instance, the unsafe state may become disconnected from the rest of the machine by transition deletion transformations.

## 3.3. Process Algebras

The Calculus of Communicating Systems (CCS) [Mil 80] is an example of a typical untimed process algebra. Process algebras are adept at expressing concurrency and nondeterminism and are characterized by compact syntax definitions. A CCS system consists of a set of concurrent processes. At any step in the computation, each process is engaged in some *action*. Actions are either input/output actions or internal actions. For each input (output) action a, there is a corresponding output (input) action a′ that interacts with a (i.e. a/a′ represent a sending/receiving pair). An execution of the system is a sequence of the sets of actions that each process is executing at any given step. The syntax of CCS is representative of most process algebras. The $0$ process represents deadlock and cannot execute any action. The *action operator* $a.P$ represents a process that executes action a and then behaves as process P. The *choice operator* $P_1 + P_2$ denotes a process that nondeterministically chooses to behaves as either $P_1$ or $P_2$. The *parallel operator* $P_1 \mid P_2$ indicates a process in which $P_1$ executes concurrently with $P_2$. The *hiding operator* $P\backslash A$ represents a process that behaves as P but with the set of actions A hidden. That is, the input/output actions in A will not synchronize with their corresponding actions that may be executing in other processes. Recursive processes are defined with $fix(X.P)$, which allows the specification of infinite behaviors. For example, $P' = fix(X.(a.X))$ means that P′ behaves as an infinite sequence of A actions. Properties of CCS systems are themselves specified as a collection of CCS processes. The property processes define acceptable executions of which the executions of the behavioral processes must be a subset.

### 3.3.1. Calculus of Communicating Shared Resources

The Calculus of Communicating Shared Resources (CCSR) [GL 90, GL 92] is a timed process algebra based on CCS, which can specify resource handling explicitly. CCSR takes neither the interleaving approach nor the noninterleaving approach, but a combination of the two. Only one

*event* can execute at any given time on the same resource. The number of resources in the system determines the amount of concurrency possible. This is similar to the ASTRAL approach where each process is assumed to have its own processor and only a single transition may be firing on any given processor at the same time. What is different is that CCSR assumes that all communication is synchronous on *connection events* while ASTRAL assumes that sufficient resources (e.g. communications coprocessors) exist for asynchronous message passing.

An event is the basic unit of computation and represents items such as executing code and sending or receiving messages. Time in CCSR is discrete and each event takes one unit of time. An *action* is a set of events such that a unique event in the set is occurring on each resource in the system. The set can also contain *annotation events* that are not associated with any resource, which indicate special conditions in the system such as the ability to terminate or the occurrence of an error. At any given time, the system is performing some action, thus an execution is a sequence of actions. Connection events designated by $e!$ and $e?$ denote sending and receiving a message, respectively (or alternatively, executing and waiting for an interrupt). Each event in the system is associated with an explicit priority, which determines the event that executes on a resource when more than one event is waiting for it.

CCSR contains all the operators of CCS with the modifications discussed below as well as additional operators to define resource and timing constraints. The *parallel operator* $P_1 \,{}_i\|_j\, P_2$ is similar to that of CCS but $P_1$ must execute synchronously with $P_2$, and $P_1$ and $P_2$ are limited to the sets of resources i and j, respectively. Synchronous execution means that $P_1$ and $P_2$ may execute actions $A_1$ and $A_2$ simultaneously only if $A_1$ and $A_2$ contain corresponding connection events (i.e. $e!$, $e?$). The *close operator* $[P]_i$ denotes a process that behaves as P and occupies exactly the resources of set i. That is, if any resource assigned to P does not have an event to execute, "idle" events will be executed to "pad" that resource. A variable $X$ used without $\mathsf{fix}$ denotes an *open variable* whose behavior is defined by the external environment.

Timing constraints are expressed using the *scope operator*. The scope operator $P\ \Delta_t^{(B,C)}(Q, R, S)$, where B and C are sets of annotation events and Q, R, and S are processes, represents a process that behaves as P with conditions on its termination. If P does not terminate within t time units (t can be $\infty$), the process behaves as R (i.e. an exception handler) with the annotation flags of C. If P does terminate within t time units, the process behaves as Q, but may terminate immediately if B contains the *termination event* $\surd$. Any time in the execution of P, P may be interrupted by S, at which point the process behaves as S without time restrictions. The scope operator allows CCSR to specify

timeouts, interrupts, periodic behavior and exceptions. For example, message? $\Delta_{timeout}^{(\{\},\{error\})}$(message_handler, error_handler, 0) represents an uninterruptable process that is waiting for a message. If the message arrives before a timeout occurs, the process executes the message handler. If a timeout does occur, however, an error is indicated and an error handler is called. Proofs of CCSR specifications are performed using an equivalence relation based on bisimulation of CCSR formulas. The equivalence relation states equivalencies such as the commutativity and associativity of the choice operator. A CCSR specification consists of a process definition of system behavior as well as a process definition of system properties. A proof that the model meets its critical requirements consists of a sequence of steps that transforms the formula for the model into the formula for the properties.

Instead of specifying systems directly in CCSR, systems can be written in a layer on top of CCSR called Communicating Shared Resources (CSR). CSR has most of the functionality of CCSR but puts it into the form of a programming language. A CSR specification is written in two parts. A program written in the CSR *application language* specifies the behavior of the system independent of resources and priorities. The atomic primitives of the language are executing actions and sending or receiving messages over a channel. Statements can be executed nondeterministically between given time bounds. The language contains nonterminating loops, but more sophisticated periodic requirements can be expressed using the *every loop*. Every t do S od cyclically executes statement S every t time units. If S takes more than t time units to execute, it will be aborted and restarted at time t. If S takes less than t time units, the loop idles until time t at which point S is again executed. Statements can be interleaved so that if one statement is idle, another statement can use that time to execute. Finally, CSR contains a scope operator similar to that of CCSR, which executes a statement and can define interrupt and timing conditions upon which execution is aborted and a given handler enacted. Once the application program is written, a *configuration schema* is written in the CSR *configuration language* to provide a context in which to operate. A schema binds processes to resources and associates input channels with output channels. It also assigns priorities to various items and binds free time variables to specific values.

Requirements of CSR programs are declared in scope timeouts and every loops by adding the hard keyword to their declarations. In a scope expression with a hard timeout, if the statement does not finish executing or no interrupt occurs before the timeout, it is considered an error condition and is marked as such in the execution. Similarly, an error condition is generated if the loop body of a hard every loop does not terminate before the next iteration begins. To verify that these error conditions

do not occur and that more general properties hold, the CSR program is first translated into an equivalent CCSR specification. From this point, either the standard CCSR proof system can be utilized, or for verifying the absence of error conditions, a reachability analyzer is available, which checks all executions for error conditions. The analyzer relies on the fact that a translated CSR program always results in a CCSR specification with finite-state executions. The CSR to CCSR translation is similar in purpose to the ASTRAL to TRIO translation proposed for an earlier version of ASTRAL in [GK 91b].

### 3.3.2. Timed Communicating Sequential Processes

Timed Communicating Sequential Processes (TCSP) [DJR 91] is a timed extension of Communicating Sequential Processes (CSP) [Hoa 85]. A TCSP process describes a sequence of observable events. TCSP events are synchronization (i.e. communication) events between two processes or between a process and the environment. The syntax contains CSP variations of all the CCS operators and additionally contains two timed operators. The *idle operator* $\mathsf{WAIT}\ t$ denotes a process that does not synchronize on any event for t time units and then terminates. The *timeout operator* $P \triangleright^{t} Q$ represents a process that behaves as P unless no synchronization events occur before time t, at which point P is interrupted and the process behaves as Q.

An execution of a TCSP process is called a *timed failure*. A timed failure (s, ℵ) consists of a *timed trace* s and a *timed refusal* ℵ. A timed trace is an ordered set of timed events (t, a), where t is a non-negative real and a is a synchronization event. A timed trace represents the observable events that have occurred in the system. If (t, a) is in a timed trace, then synchronization a was observable in the process at time t. A timed refusal is also a set of timed events (t, a), but if (t, a) is in a timed refusal, then the process refused to synchronize on event a at time t. For example, the $\mathsf{STOP}$ (deadlock) process refuses all synchronization events at all times while $\mathsf{WAIT}\ t \rightarrow P$ refuses all synchronization events for t time units and then refuses those events that P refuses.

Properties of TCSP processes are boolean expressions using first-order logic, standard set operators, and special trace operators, which limit the acceptable timed failures (s, ℵ) of the system. The empty trace is denoted <>. If $T_1$ and $T_2$ are traces, $T_1 \hat{}\ T_2$ is the concatenation of $T_1$ and $T_2$. $\#(T)$ is the number of timed events in trace T. $T_1 \leq T_2$ holds if $T_1$ is a prefix of $T_2$ and $T_1\ \mathsf{in}\ T_2$ holds if $T_1$ is a prefix of some suffix of $T_2$ (i.e. a continuous subsequence). The operators $\mathsf{begin}(T)$ and $\mathsf{end}(T)$ return the first and last event in a timed trace, respectively. Traces can be shifted in time using $T + t$, which adds t to the time of every event in T. A timed trace can be converted into an untimed trace

using tstrip(T), which removes the time component of each tuple in trace T. Finally, the *during operator* $T \uparrow I$ returns the subsequence of trace T that lies in time interval I. An example property using these operators is $\langle(t, \text{send\_error})\rangle \text{ in } s \rightarrow \exists t_1.(t - t_{error} \leq t_1 \wedge t_1 < t \wedge \langle(t, \text{send})\rangle \text{ in } \aleph)$. This formula states that whenever a send error occurs, a send must have been refused within the past $t_{error}$ time units.

A TCSP system consisting of a process declaration P and a property specification S is verified using a deductive proof system to show that P satisfies S. Additional proof rules are given for proofs of open systems. In these rules, instead of showing that P satisfies S, which is not always feasible in a hostile environment, the weaker condition P satisfies ($E \rightarrow S$) is proven, where E is an assumption about the behavior of the environment. This approach is similar to ASTRAL where schedules are proven using assumptions in the environment clauses.

## 3.4. Hoare Logics

Hoare proposed verifying systems by reasoning directly with actual implementation code written in a high-level language [Hoa 69]. A program S is augmented with a *precondition* and a *postcondition*. The precondition P is an assertion about the variables of the program at system initialization. The postcondition Q is an assertion about the variables at program termination. That is, a postcondition specifies the critical requirements of the program (i.e. its functionality). Together, {P} S {Q} is known as a Hoare triple. A triple is read "if P holds in the initial state of S and S is executed, Q is guaranteed to hold at program termination". Hoare logic can only specify partial correctness (i.e. properties of terminating computations). To verify that Q does indeed hold, a deductive proof system is used. A set of inference rules is defined for each expression in the high-level language. S is annotated with the assertions that are true at each point in the code, which are derived from the inference rules based on P, S, and Q. The proof that Q holds at program termination is then performed by linking the assertions into a chain of reasoning such that P is the first assertion, Q is the last, and each step is justified by an inference rule. Given its ability to reason directly with high-level language code, Hoare logic has been extended in several ways to allow reasoning about real-time programs.

### 3.4.1. Hoare Logic with Time

[Sha 89] presents an extension to Hoare logic for dealing with time in high-level language programs. The main idea is that although precise execution times for individual statements cannot be determined due to compiler transformations, register/memory contention, etc., in most cases they can

be bounded by minimum and maximum execution times. Thus, although it may not be possible to reason about the exact time at which each statement in a program occurs, it may still be desirable to be able to reason about an approximate, but bounded time. The approximate current time in the system is represented by an interval $[rt_{min}, rt_{max}]$ (called RT), in which the actual real world time is always contained. The execution interval T(S) of each primitive statement S is assumed to be known beforehand. From these, the intervals of composite statements can be determined.

The standard Hoare triple {P} S {Q} becomes $\{P\}\ <RT_0 = RT;\ S;\ RT = RT_0 + T(S)>\ \{Q\}$. P and Q can reference RT to make assertions about time. For example, {RT + T(S) = [delay, deadline]} S {RT = [delay, deadline]} specifies that statement S must finish execution no earlier than delay and no later than deadline. It is not possible to give all operations meaningful time bounds because some operations can only proceed upon the occurrence of an event in the environment (e.g. a mouse click) or an action of another process in the system (e.g. releasing a lock). For these operations, the upper bound can be infinite thereby rendering the remainder of any timing analysis meaningless. In the latter case, it may be possible to derive an upper bound by analyzing the timing behavior of the other process and determining the interval in which the lock can possibly be released. In the former case, however, even this type of analysis does not help.

[Sha 89] advocates associating a timeout with each operation of this type. If the operation does not complete before the timeout is reached, an error is returned and the program can react accordingly. For example, to recognize a double mouse click, the program waits for the first full click, and then waits for a down event with a timeout. If the down event occurs before the timeout expires, the program registers a double click. Otherwise, the program registers a single click. Thus, the analysis can be simplified even though the down event may not occur for an infinite period.

### 3.4.2. Real-Time Hoare Logic

Like untimed Hoare logic, the timed version discussed above can only express partial correctness. The real-time extension to Hoare logic presented in [Hoo 94] (henceforth called RTHL), however, can also express total correctness. A system is modeled as a collection of processes. The behavior of each process is represented by the values of its *objects* over time. Objects include items such as communication channels, variables, and constants and are classified as either *observable* or *local*. Observable objects are visible to all processes and represent the interface of the process, while local objects are only visible to the declaring process and represent implementation details. *Observable actions* are operations on observable objects. *Observable events* are occurrences of observable

actions. The time domain is the non-negative reals along with ∞. An execution of a system is represented by a set of states, where each state has a component now, which holds the current time in that state. A state contains the values of all local objects at the current time and the times of occurrence of all observable events from system initialization up to the current time.

An RTHL triple is written in the form ≪A≫ P ≪C≫, where A is an *assumption*, P is a process, and C is a *commitment*. If A holds at the initialization of P, then after the execution of P, C is guaranteed to hold. Properties of the values of local objects at the start and end of the execution of P can be expressed in the assumption and commitment, respectively. The values of local objects are undefined when now = ∞ so assertions about these values in nonterminating computations cannot be made. It is also not possible to reason about the values of local objects at any time between the start and end of execution because this is implementation dependent. It is possible, however, to reason about the times of occurrence of observable events at any time in the execution. Thus, properties that must hold in all possible implementations of the system can be expressed. If O is an observable action, O at t holds if O occurs at time t. Similarly, for a time interval I, O during I holds if O occurs at all times in I and O in I holds if O occurs at some time in I. By using the variable now in the assumption and commitment, assertions can be made about the starting and ending times of the process. Thus, the total correctness of P with respect to A and C can be written ≪A ∧ now < ∞≫ P ≪now < ∞ ∧ C≫. That is, if A holds, then P must terminate and C must hold.

Processes can be composed in parallel with the ‖ operator. $P_1 \parallel P_2$ denotes a new process that consists of the union of the observable actions of $P_1$ and $P_2$ and similarly the union of the local actions of $P_1$ and $P_2$, but with the restriction that $loc(P_1) \cap loc(P_2) = \emptyset$. RTHL specifications are proven with a deductive proof system similar to that of untimed Hoare logic. The proof system contains an inference rule for parallel composition that allows the proofs of processes to be performed individually and then combined for the proof of the entire system. This simplifies the proof process since the proofs of individual processes are usually much simpler than those for the system as a whole. This is similar to the ASTRAL approach where the invariants and schedules of processes are proven separately and then combined to derive global properties.

## 3.5. Programming Languages

Although it is possible to develop real-time systems using existing traditional programming languages by taking advantage of real-time operating system constructs, programs using such methods are often very low level and do not follow the intuition of the programmer. In addition,

traditional languages are not always amenable to formal analysis. Instead of using existing languages and verifying properties using the logics of the previous section, new programming languages are being developed, which offer high level real-time mechanisms and are designed with formal analysis in mind from the start.

### 3.5.1. LUSTRE

LUSTRE [HCR 91] is a programming language for implementing reactive systems. It is based on the *synchrony hypothesis*, which states that programs react instantaneously to external events. In practice, this is akin to requiring that a program can always finish reacting to one event before any other event occurs. All data in LUSTRE is represented by *flows*. A flow is an infinite sequence of values (e.g. booleans, integers, reals, tuples) and an infinite sequence of times at which those values are defined, known as a *clock*. The *basic clock* is the clock that defines the smallest unit of time in the system. No data can change values at any finer grain than the basic clock allows. All other clocks in the system are infinite sequences of boolean values (eventually) defined in terms of the basic clock, such that true indicates the occurrence of a tick. For example, in the following, C′ is defined in terms of C, which is defined in terms of the basic clock:

| ticks of basic clock | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| clock C | F | T | F | F | T | F | T | T | F | ... |
| clock C′ | | F | | | T | | F | T | | ... |

Operations on multiple flows are only legal when all operands (i.e. flows) share the same clock. Thus, besides traditional arithmetic, boolean, relational, and conditional operations, LUSTRE has two operators to change the clocks associated with flows. The *sampling operator* E when B takes a flow E and a clock B and produces a flow that has the values of E, but only at the ticks in which B is true. That is, the sampling operator "slows down" the clock of the given flow. For example,

| clock B | F | T | F | F | T | F | T | T | F | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| flow E | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ... |
| E when B | | 2 | | | 5 | | 7 | 8 | | ... |

Similarly, the *interpolation operator* current E takes a flow E whose clock is derived from another non-basic clock B, and produces a flow that has the values of E, but when E's clock is false and B is true, the flow takes the previous value of E in the sequence. That is, the interpolation operator "speeds up" the clock of the given flow. For example,

| clock B | T | T | T | F | T | F | T | T | T | ... |
|---|---|---|---|---|---|---|---|---|---|---|
| E's clock | T | F | F | | T | | F | T | F | ... |
| flow E | 1 | | | | 5 | | | 8 | | ... |
| current E | 1 | 1 | 1 | | 5 | | 5 | 8 | 8 | ... |

In addition to these two operators, LUSTRE has two operators to define flows. The *previous operator* pre E takes a flow E and "shifts" it to the right, creating an undefined first element while keeping the same clock. The *followed-by operator* $E \rightarrow F$ takes two flows E and F and produces a flow that is the same as F except its first element is the first element of E. A flow is then defined by a recursive equation such as $n = 0 \rightarrow pre(n) + 1$, which defines the flow of natural numbers defined at the ticks of the basic clock.

LUSTRE code can also contain assertions in the form assert E where E is a boolean flow. These assertions can be used to express assumptions about the environment that need to hold for correct program execution. LUSTRE flows are basically equivalent to linear temporal logic state sequences so temporal properties can hold or not hold over them. For example, $\Diamond E$ is true if E is true at some tick of the basic clock. Temporal logic formulas can be written directly in LUSTRE code, thus both a program and its specification can be written in the LUSTRE notation. For example, from [HCR 91],

```
node since(X, Y: bool) returns
(XsinceY: bool);
let
    XsinceY = if Y then X else (time → X or pre(XsinceY));
tel.
```

is equivalent to the temporal logic expression $\Box\neg Y \vee \Diamond Y \Diamond X$. The LUSTRE compiler transforms a LUSTRE program into an equivalent finite-state machine. To verify a system, the implementation is compiled into a finite-state machine and properties are checked over the finite reachability graph.

### 3.5.2. ESTEREL

ESTEREL [BS 91] is a language for reactive programming also based on the synchrony hypothesis. ESTEREL uses an iterative style, rather than the data flow approach of LUSTRE. The behavior of a reactive system is divided into a set of instants at which significant events occur, namely the reception of and reaction to *signals*. Signals are the basic communication mechanism of ESTEREL. An ESTEREL process can test for the presence of, emit, and read the values associated with signals. Signals are classified as input or output signals and are only present at a specific instant. That is, they do not have a duration associated with them and are instantaneously broadcast to all other processes. All decisions based on these signals are made instantaneously and the results of the

decisions may cause other signals to be emitted and/or reactions to occur. When the system reaches a steady state (i.e. no more decisions can be made or signals emitted), the system moves to the next instant.

In addition to traditional programming language operators, ESTEREL possesses a number of operators to handle signals. Emit S makes the signal S present at the current instant. Emit S(v) is similar except that S is emitted and associated with the value v, which can be queried with ?S. Present S tests for the presence of signal S at the current instant. Await S stops the execution of the process until the instant that signal S becomes present. Do B watching S performs the body B up until the instant S becomes present. Do $B_1$ watching S timeout $B_2$ does the same thing except if S becomes present and $B_1$ has not yet terminated, the body of the timeout $B_2$ is executed. Not all programs expressible in ESTEREL syntax are admissible because of causality problems. For example, present S else emit S is contradictory because if S is present then it is not emitted but if it is not present then it is emitted. Another example is emit S(?S + 1), which says that S has different values at the same instant.

Explicit time can be introduced into ESTEREL by the introduction of a process that regularly emits a "tick" signal that other processes can refer to. This approach is similar to the basic clock in LUSTRE in that the difference in consecutive tick emissions represents the smallest interval of time needed to distinguish all events in the system. Also like LUSTRE, ESTEREL is compiled into a finite-state machine and the resulting finite reachability graph is used for the verification of temporal properties. ESTEREL is similar to ASTRAL in that both view the transmission of messages between processes as instantaneous. In ESTEREL, messages are signals and any associated values, while in ASTRAL, messages are the values of exported variables and the start/end times of transitions broadcast to all "interested" processes. Unlike ESTEREL and LUSTRE, however, ASTRAL does not follow the synchrony hypothesis, because reactions to events (i.e. transitions) must have a non-null duration so the system cannot react instantaneously to events.

### 3.5.3. PAISLey

The Process-oriented, Applicative, and Interpretable Specification Language (PAISLey) [Zav 82] is an operational approach to specifying real-time systems. A system is specified as a set of processes, each of which is declared as a typed function describing its behavior. The execution environment of the system must be specified explicitly as a process. Functions cannot be recursive and no mechanism for iteration is available. It is possible, however, to execute a function a constant number of times. The system evolves by applying the functional definition of each process to itself after each

execution cycle is complete. Thus, processes behave cyclically with the previous outputs becoming the next inputs.

Interprocess communication is accomplished through the use of *exchange functions*. An exchange function may be one of three types. An x-c type exchange function communicates over channel c and blocks until another exchange function of any type is executed on c. The xm-c type operates similarly to the x-c type but only accepts a rendezvous with x-c and xr-c type exchange functions. The xr-c type communicates over channel c, but is non-blocking so it only exchanges messages when an x-c or xm-c type exchange function is waiting. The xr-c type exchange function is used for real-time interaction such as event notifications from the environment. One typical application of an xr-c function is in the definition of a clock process that returns the current time to any process rendezvousing with it. When an exchange function rendezvous occurs, each function returns the value that was input to its "matching" function. For example, if two processes call x-c(1) and x-c(2), respectively, then x-c(1) returns 2 and x-c(2) returns 1 (i.e. they exchange values).

Every function in the system can be associated with reliability and timing requirements. Reliability requirements specify the probability that a function will fail. Upon failure, a value in the specified error domain is returned. Timing requirements include minimum, maximum, mean, and constant running times. Since functions can only perform a bounded number of iterations, it is possible to analyze the timing information of each process and verify properties about it. The main emphasis in PAISLey, however, is on the executability of specifications. The specifier is still able to gain the benefits of a formal and abstract notation, but can also immediately execute the specification and determine if its behavior is adequate.

# Chapter 4

# Problem Overview

This chapter discusses the focus of this dissertation, the rationale for why such research is necessary with respect to existing work, and the potential benefits of the end results. It also presents the goals that will be completed in this research. The first section discusses specification approaches and the second section discusses verification approaches.

## 4.1. Formal Specification

In chapter one, the main requirements needed in a specification language for designing real-time systems were discussed. Besides tools and a formal definition, it is necessary for a language to be

- intuitive
- expressive
- modular
- composable
- refinable

The following sections discuss these qualities in the different types of specification languages presented in chapter three.

### 4.1.1. Temporal Logics

Real-time temporal logics are very expressive and can specify complex properties in simple and intuitive forms. They are not intuitive, however, for specifying system behavior. This is because behavior is specified by placing constraints on system executions rather than by specifying the *evolution* of a system execution. That is, instead of describing how the system changes from one state to the next, temporal logic specifications describe a collection of properties about system executions that any system implementation must satisfy. Although this restricts possible system implementations as little as possible, it does not fit the way in which most system architects design a system. Using a collection of properties also means that temporal logic specifications lack structure, which means they have limited facilities for modularity, composition, and refinement. Without these facilities, using temporal logics to specify large and complex systems is impractical.

### 4.1.2. State Machines

Real-time state machines are very well suited for expressing system behavior. They allow the behavior of a system to be specified as a description of how the system changes from one state to another. They are also naturally represented in graphical fashion, which allows system architects to visualize the structure of the system. State machines are very structured and in most cases provide facilities for modularity, composition, and refinement, thus they are well suited for specifying large and complex systems. By themselves, however, they are not adequate for expressing system properties. This is because properties are specified as an abstract state machine that executes in the desired manner. For complex properties, it is very difficult and unintuitive to construct the appropriate machine.

### 4.1.3. Process Algebras

Real-time process algebras have very powerful mechanisms for modularity, composition, and refinement. Any process may be refined as a collection of other processes or composed with other processes using special operators built-in to the language. In some sense, however, they are *too* modular because in order to interpret the behavior of a process, it must be broken down into a collection of simpler processes that becomes very large for processes of even moderate complexity. Some of the specialized operators in process algebras also have very complex semantics, which makes them unintuitive. Process algebras are well suited for specifying interactions between system components. It is difficult, however, to specify implementation details of a system. Like state machines and for similar reasons, process algebras are not adequate by themselves to express complex system properties.

### 4.1.4. Hoare Logics

Real-time Hoare logics are well suited for reasoning about actual implementations of systems. They are modular and composable at the level of the programming language that they are associated with. As Hoare logic specifications are already at the level of the implementation, however, they are not refinable. For large programs, the complexity of the specification becomes unmanageable. Due to the low level nature of Hoare logics, they are useful for the single purpose of verifying actual implementations, but are not suitable for specifying and verifying systems in general.

### 4.1.5. Programming Languages

Real-time programming languages have the same qualities as real-time Hoare logics. Namely, they are modular and composable, but are at the implementation level, which makes them unsuitable for the specification of systems in general.

In the end, the real-time specification languages that are most likely to meet the criteria of chapter one are those that combine the modeling capabilities of a state machine or process algebra with the property expressiveness of a temporal logic. The ASTRAL language is one such specification language. ASTRAL combines a timed transition system with an explicit-clock first-order logic. This results in a language that can specify both system behavior and system properties in an intuitive manner.

ASTRAL also has a modular proof system and has facilities for composition and refinement. There were, however, some errors and incompleteness in the original ASTRAL definition. The ASTRAL semantics and proof obligations suffered from a number of soundness and completeness problems. In addition, the composition capabilities were not completely described and had no tool support; thus, they were unusable given the number of complex transformations they required. The refinement mechanism was also incompletely described and its expressiveness was limited. ASTRAL had only rudimentary tool support for writing specifications. Thus, to meet the design criteria, it was necessary to correct these deficiencies.

## 4.2. Formal Verification

Although the process of writing a specification is in itself of much benefit to the designer, only full formal verification can provide the absolute assurance that a specification meets all of its critical requirements. Performing this verification, however, is much more difficult and technically demanding than simply writing a specification. Verification reasoning is fairly difficult even in untimed systems and when time is added, it becomes significantly harder. This is because the behavior of real-time systems is much more complex. In untimed transition systems, it is possible to take advantage of *frame axioms* [BMR 95], which state that elements of the state that are not explicitly changed to a new value in the postcondition of a transition remain unchanged from the precondition. This means that a property need only be verified in the initial state and at each time a postcondition changes the state. In timed transition systems, however, the current time is an element of the state. Thus, since time is continuously changing, it is not possible to use a frame axiom

because even though a property may hold when a transition started and when it ended, the property may have been violated in between due to a change in time.

Figure 4.2 illustrates the various situations that may cause a requirement to be violated while a transition T1 is executing on a process P1. One type of violation occurs when a timing deadline of a requirement expires while a transition is executing. In the figure, the dashed lines indicate timing requirements that were initiated by events occurring in the same process, in other processes, and in the external environment, which expire at the dotted lines. As an example, suppose P1 is a process that samples conditions in the external environment and reports these conditions to another process. A reasonable timing requirement would be that whenever P1 is in sampling mode, every stimulus occurring in the external environment is processed within the timeframe shown in the figure. Suppose T1 turns off sampling mode. In this case, the timing requirement holds when T1 starts and when T1 ends, but does not hold in between when the deadline expires. Violations can also occur due to concurrent activities occurring in other processes of the system. For example, there may be a requirement stating that whenever P1 is in sampling mode, T4 cannot be in some state s. If T4 sets the state of P2 to s, then a violation occurs when T1 is executing. Once again, however, the property holds when T1 starts and when T1 ends, so frame axioms are not sufficient to catch the violation.

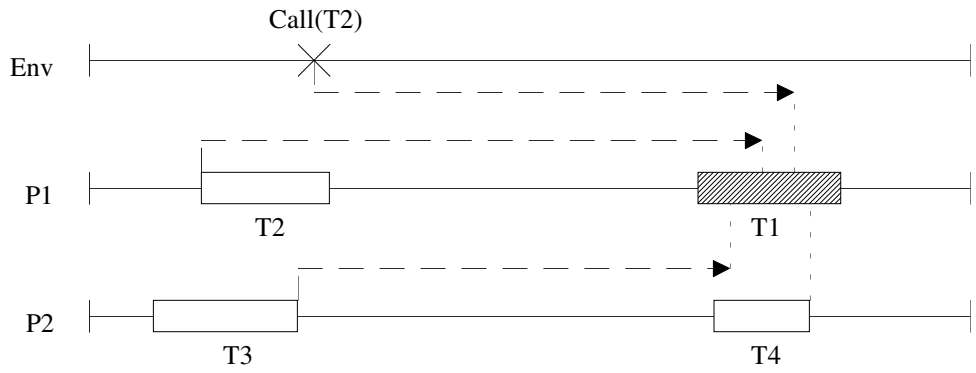



Figure 4.2: Some violations that can occur while a transition is firing

Since the state may change while a transition is executing, the user must reason about a property at all possible times instead of only the ends of transitions, and about multiple processes and the external environment. This makes even small real-time systems difficult to analyze. For this reason, there is a need for automated tools and analysis techniques that reduce the technical complexity, error, and burden associated with real-time formal verification.

### 4.2.1. Fully-Automated Analysis Techniques

The most desirable tools, and consequently the ones most explored by researchers, are fully automated procedures such as model checkers, which verify specifications without the need for any user assistance. Fully automated approaches are desirable because they enable anyone who can specify a system and its requirements to prove that the requirements hold or do not hold in the system without any expertise with the proofs or in-depth knowledge of the underlying semantics. Due to the robustness of the ASTRAL language, a fully automated decision procedure for ASTRAL proof obligations is not possible, even when some significant limitations are placed on the language, as shown in the undecidability results of appendix B.

Some languages are supported by fully automated decision procedures, but only because the expressiveness of these languages and/or the formulas that can be proven automatically are severely limited. Thus, languages supported by these procedures cannot be used to specify many interesting systems and/or their verifiers cannot prove many interesting properties. For instance, the TPTL language, discussed in section 3.1.1.1, is supported by such a procedure [AH 94]. One major limitation of TPTL, however, is that the time variables introduced by freeze quantifiers can only be related by constant distances (e.g. $t_0 \leq t_1 + c$). Thus, TPTL can express neither system models nor system properties in which time variables need to be related by variable distances. For example, consider the elevator system of section 2.1.3. A reasonable property of this system is that the time it takes to move from a floor i to a floor j takes less than $|j - i|$ times the maximum amount of time that can be spent on each floor. This type of property renders TPTL undecidable, however, so it is not allowed in the language. Therefore, since TPTL cannot be used to specify this property, its decision procedures are of no benefit to the user for this and many other reasonable systems.

The Modechart language, discussed in section 3.2.2.3.2, is also supported by a fully automated decision procedure [JS 88, Stu 90, YMS 95]. Proving that an arbitrary RTL formula holds in a Modechart specification is undecidable, so formulas to be automatically verified must be in one of a set of restricted forms. Although these forms allow a number of useful properties such as reachability, starvation, and separation to be verified, a large number of other properties do not fall into these categories. For example, consider the stoplight system of section 2.1.8. To satisfy the requirement that the main direction must always be green when no cars are present, it is necessary to prove that between the time that the light for the main direction turns green until the time it turns red, a car has arrived in either a left turn lane or a normal lane in one of the other directions. Even though RTL allows such a property to be specified, the Modechart verifier cannot prove properties in

which one interval contains a choice between events. A user specifying a property not in a form supported by the verifier must then prove it by hand using only the inference rules of RTL.

In the two verification approaches discussed above, limitations are placed on the systems that can be specified and/or the properties that can be verified. These restrictions are necessary because some systems and properties are inherently undecidable so they cannot be verified automatically. Even when such algorithms exist, however, they cannot be applied to real-world systems, which are large and complex. This is because almost all of the fully automated decision procedures for real-time languages have exponential running time due to the vast number of executions that become possible when time is added to system specifications. The larger and more complex the specification to be verified, the greater the need for automated assistance, but the larger and more complex the specifications to be verified, the less likely existing decision procedures are able to verify them in a reasonable amount of time. This leads to the unfortunate conclusion that the greater the need for automated assistance, the less existing decision procedures are able to help. Clearly, there is a need for tools and analysis techniques that do not place such undesirable restrictions on the user.

### 4.2.2. Semi-Automated Analysis Techniques

The approaches in the previous section are aimed toward verifying system properties without the need for any human interaction. When these methods can be applied, they have the distinct advantage of allowing anyone who can specify a system to prove the necessary properties without any knowledge of or experience with the underlying theories behind the proof process. Unfortunately, the disadvantages of these methods often outweigh this benefit as discussed above. Since fully automated verification is all too often unfeasible, other semi-automated tools have been developed that, while still requiring technical expertise of the user, eliminate tedious and error-prone tasks and provide valuable assistance during the verification process. More importantly, they can often be applied to larger systems and be used to prove more significant results.

Interactive theorem provers provide mechanical support for deductive reasoning. To prove the properties of a system with a particular theorem prover, the system and its proof obligations are first expressed in the specification language associated with the prover. The obligations are then discharged by reducing the high-level proofs of the obligations into simpler subproofs using the axioms and inference rules of the prover's specification language. The goal of this reduction is to simplify the proofs enough so that each subproof can be automatically discharged by the prover's basic built-in decision procedures that support arithmetic and boolean reasoning. Theorem provers

provide a number of forms of assistance, which include preserving the soundness of proofs, finishing off proof details automatically, keeping track of proof status, and recording proofs for reuse.

In [AH 96], the PVS theorem prover is used to reason about timed automata, which are discussed in section 3.2.1.2. In this approach, timed automata specifications are expressed in the PVS specification language and the operation of the automata are defined by a reachability function that is true or false for a particular transition and initial state. To prove properties of timed automata, the concept of induction was encoded in the PVS language. The hand proof of the system is set up and the corresponding steps in the PVS description are executed to check the proof. This approach provides assurance that the proofs developed for a particular system are indeed correct. With user guidance as to the order that proof steps should be performed, interactive theorem provers can prove fairly significant results in a reasonable time. Without such guidance, however, they can still prove fairly significant results, but the time they take is no longer reasonable. This is due to the fact that the system does not have the intuition that human provers do as to the ordering of proof steps. As a result, they may spend long periods of time working with large formulas that could have been significantly reduced in size if the steps were performed in a different order. Thus, although interactive theorem provers can often prove results unaided, they are mostly reduced to proof checking because in setting up the proofs into a form usable by the system and providing the guidance necessary for reasonable performance, users are essentially performing most of the complex reasoning themselves.

Other tools exist that are not necessarily geared toward verification, but are nonetheless useful during the verification process. For example, simulators and symbolic executors allow the user to examine the behavior of transition-based specifications during explicit execution scenarios. The user sets up the state of the system and the times that external events are to occur. The simulator then determines the evolution of the system for a specific period of time based on the information given. Even undecidable transition-based languages can be simulated because given a specific state and the events that are to occur, a simulator can simply apply the transition semantics of the language to produce the next state, which is significantly less complex than proving a property for arbitrary executions. Throughout the verification process, a simulator can be used to test the system executions that the prover thinks may violate critical requirements. Thus, although the burden of determining the potentially problematic scenarios still rests on the user, the tedious and error-prone task of actually executing all the steps in those scenarios is performed by the simulator. The disadvantage of simulation is that a single run of a simulator only shows the presence or absence of errors for that

particular run. It does not show the absence of errors for the entire system. Nonetheless, it is still a valuable component of a verification system given the fact that a single run can almost always be performed in reasonable time (e.g. proportional to the number of transitions and state variables in the specification and the number of time steps to be executed).

### 4.2.3. Hybrid Analysis Techniques

The tools discussed above are all valuable during the formal verification process. By themselves, however, none of them provide the level of assistance required in the verification of real-world systems. In [Ost 99], some of the problems associated with individual tools are alleviated by employing several analysis techniques during verification of TTM/RTTL systems, which are discussed in section 3.2.1.3. In this approach, instead of reasoning about an entire system with a model checker or a theorem prover, a decomposition and refinement strategy is developed that allows systems that might not otherwise be verifiable in a reasonable amount of time by either tool individually to be verified using a combination of both tools. The user attempts to decompose a full model m that must satisfy a requirement r into two modules m1 and m2, such that m1 satisfies a requirement s1, m2 satisfies a requirement s2, and $s1 \wedge s2 \rightarrow r$, as shown in figure 4.2.3-1. Since m1 and m2 as well as their requirements s1 and s2 are less complex than m and r, the model checker is more likely to be able to verify them. Similarly, proving $s1 \wedge s2 \rightarrow r$ is a simpler task for the theorem prover than proving the system as a whole.

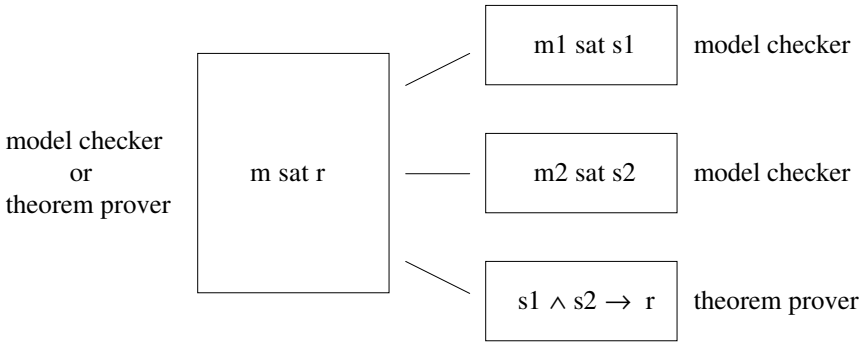



Figure 4.2.3-1: Decomposition

The other technique employed is refinement. The most abstract level of the system is verified and then each level is shown to be observationally equivalent to the level below. Figure 4.2.3-2 shows the process of refinement where more lines indicate more detail. Since the top level has a smaller state space than the more detailed levels below, model checking will again have a better chance of being able to verify the system. The proof of observational equivalence can be completed in polynomial

time for certain classes of TTMs, and for the rest the procedure is still worthwhile if the work saved by being able to model check the top level is more difficult than the work required in the hand proof of observational equivalence. Although the decomposition and refinement strategies do alleviate the size restrictions on model checking somewhat, the individual components, for the most part, still suffer the same limitations as discussed above. The model checker can still only verify appropriately restricted systems and the theorem prover still requires a significant amount of human reasoning that is not assisted by any tools. In addition, the user is not given any guidance as to how a system can be decomposed or refined to most effectively take advantage of these techniques.

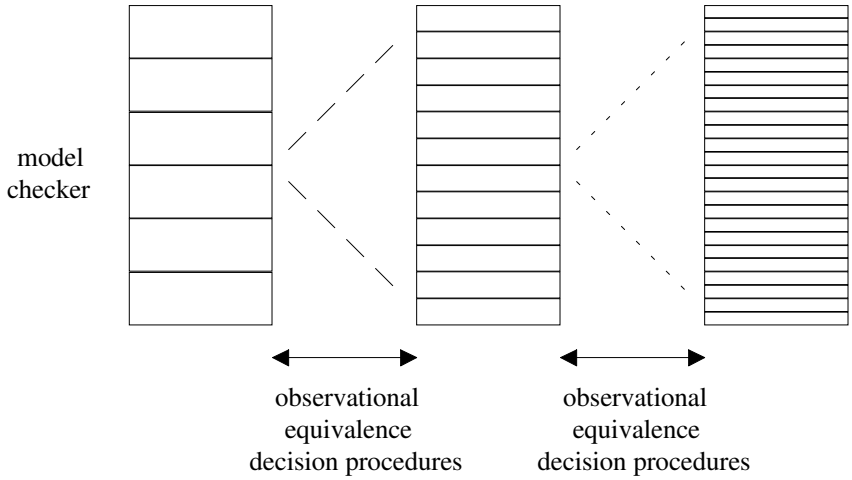



Figure 4.2.3-2: Refinement

In [MS 96], a tool is presented that combines model checking and simulation. The idea of this approach is similar to the one above. Namely, it attempts to limit the state space that must be explored so that the use of the model checker is more computationally feasible. In this case, a simulator is used to simulate the system from the initial state up until a state of interest at which point the model checker is invoked to verify the possible system executions after that point as shown in figure 4.2.3-3. Since the computation graph from the state of interest is potentially just as large as the graph from the initial state, termination conditions must be specified to limit the number of states that are explored from the state of interest. The model checker aborts the current execution path whenever a state that satisfies a termination condition is reached. Termination conditions include a specific number of timesteps or a specific number of events after the state of interest. If termination conditions were specified directly from the initial state, the computation graph might get too large by the time the state of interest is reached. The use of the simulator allows the graph to be pruned of uninteresting states before invoking the model checker. Although this technique is useful, it is only

suitable for finding errors and does not provide absolute assurance that a property holds. In addition, the model checker can still only verify appropriately restricted specifications.

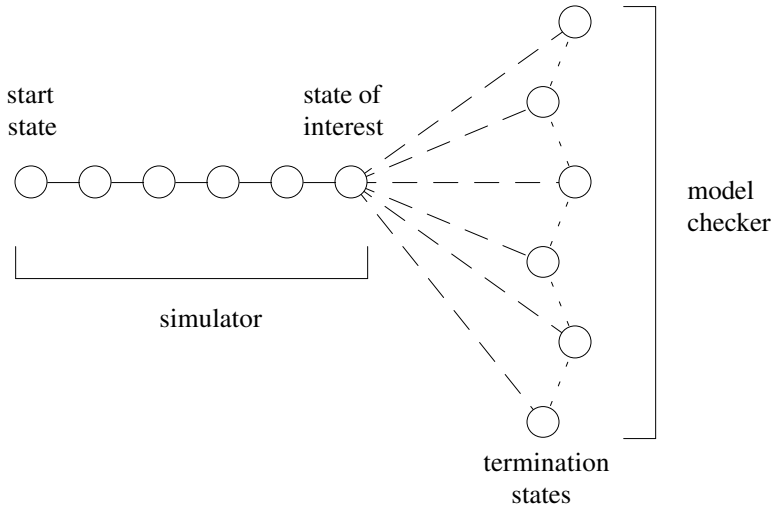



Figure 4.2.3-3: A simulator-model checker hybrid

### 4.2.4. Analysis Guidance

In order to effectively apply semi-automated techniques, it is necessary to provide the user with guidance for using the portions of the technique that are not automated. There have been some attempts to provide such guidance during analysis. In [AH 97], a number of lemmas and PVS strategies were developed to support reasoning about the Timed Automaton Model of section 3.2.1.2 in PVS. The PVS strategies correspond closely to steps that are used in hand proofs, thus the user can perform the PVS proofs in a similar manner as the proofs by hand. Although [AH 97] does provide several useful techniques for allowing the PVS proofs to correspond closely to hand proofs, what is lacking is any guidance on how the hand proof is to be constructed. This means that the user may not be able to recognize how a proof should be constructed and thus will not be able to fully take advantage of the appropriate strategies.

Guidance as to how proofs should be constructed is addressed in [HMP 94]. In this work, proof methodologies based on temporal logic reasoning are discussed for timed transition systems. Two different specification styles are identified and different proof techniques are presented for each. In the first specification style, real-time properties are expressed using time-bounded temporal operators. Proof rules are provided for proving bounded-invariance and bounded-response properties. In the second specification style, real-time properties are expressed using an explicit clock variable. Since the clock variable can be thought of as just an ordinary variable, untimed proof techniques are

used to discharge proofs in these systems. This work suffers the opposite problem of [AH 97]. Namely, guidance is given for constructing proofs, but is not supported with adequate tools.

### 4.2.5. New Analysis Techniques

In the end, current tools and analysis techniques do not provide enough assistance to make the verification of real-world real-time systems practical. What is needed is comprehensive support during all phases of verification. In a practical analysis system, all of the approaches discussed above would be components but the system would also include tools and analysis techniques applicable in the areas where the other methods have inherent weaknesses. The bulk of these additional tools and techniques would be used to develop proofs for those properties that could not be automatically verified. In addition to tools and analysis techniques, it is valuable to provide a step-by-step procedure for discharging proof obligations, which describes how to use the available approaches most effectively based on previous experience gained from the proofs of other systems. The focus of this research is to develop such a set of tools and analysis techniques and the step-by-step methodology for reasoning about the behavior of real-time systems, which is applicable to verifying large and complex real-world specifications. The desired end result is an analysis system that reduces the verification process from an *ad hoc*, error-prone procedure requiring significant technical expertise to one in which the user can follow a set of steps and perform fairly straightforward reasoning when required.

In order to meet this goal, a number of areas of research were investigated. The tasks that were to be completed included

- Defining classification schemes and developing analysis techniques for each
- Developing formula splitting methods
- Developing proof obligations based on formula type
- Developing querying mechanisms for retrieving useful information
- Designing a transition sequence generator
- Defining the requirements for a symbolic executor
- Determining the requirements for and integrating an interactive theorem prover
- Developing methods for combining the approaches
- Designing and implementing a specification manager

By developing classification schemes, analysis can be guided by classifying the elements of the current specification and choosing the appropriate analysis technique for each. Formula splitting methods allow a formula to be broken down into a collection of simpler formulas that may be amenable to different types of analysis. The different types of analysis may include proof obligations based on formula type, where simpler formulas would have simpler proof obligations. In order to

retrieve various information such as the classification types of formulas and other elements, a set of querying mechanisms can be used. These mechanisms include a transition sequence generator, which generates the sequences of transitions that are possible in a process. Such a tool is a first step towards a symbolic executor, which allows more general scenarios to be simulated. A theorem prover provides semi-automated support for analysis. It is not enough, however, to just develop these tools and techniques. They must also be combined into a cohesive methodology for reasoning about real-time systems. This methodology should be used in a specification manager that directs the user as to which analysis step should be performed next.

All of the above tasks were completed. Chapter five describes the formula splitter and the specification manager. Chapter eight presents the classification schemes and querying mechanisms, including the transition sequence generator and how it is a first step towards a symbolic executor. The analysis techniques developed for the classification schemes are discussed in chapters nine and ten. The encoding of ASTRAL into the language of a theorem prover is discussed in chapter six and the use of the theorem prover is shown in chapter ten. Chapter six and ten also mention the proof obligations used for different formula types. Finally, the methods used to combine the approaches are discussed in chapters five, nine, and ten.

# Chapter 5

# Software Development Environment

The success of any language, be it for implementations or specifications, is very often directly related to the availability and quality of tools that support it. Without quality supporting tools, the time and expense of wading through multiple technical papers and reference manuals to grasp the power and subtleties of a language may cause developers to be unwilling to use it. However, with the availability of tools, the payoff becomes much greater, since a large portion of the information contained in the documents can be directly incorporated into the tools making the language more intuitive and easier to use. More importantly, the development process becomes much less susceptible to human error by significantly reducing the amount of work the user needs to perform manually. Since the goal of formal methods is to help implementers prevent errors in system design, it is only appropriate that formal specifiers be supported by tools designed along the same theme, which help them develop specifications without error. This is particularly relevant when working with large systems where the amount of work may overwhelm even the most polished formal specifier. In addition, many specification languages that feel relatively intuitive when working with small examples may quickly become unwieldy when applied to larger systems. For this reason, it is very desirable to provide the specifier with a set of tools that eliminates as much of the burden of specifying and verifying large systems as possible.

Integrated development environments, which combine tools such as syntax-directed editors, verification condition generators (VCGs), and specification processors, offer increases in efficiency and correctness over stand-alone versions of these tools used together. For instance, an integrated environment can eliminate the time and expense of switching between the editing and processing of a specification. Instead of saving the specification, loading it into the specification processor, saving the results of processing, and finally using the editor to manually search for the resultant errors, the process can be streamlined into a click of the mouse to process the specification and another click to switch to the editor and jump directly to the error. This ease of use promotes checking for errors

early and often rather than waiting until the entire specification is written, which is usually more costly and susceptible to major design flaws.

In addition to reducing errors and facilitating the use of specification languages, integrated environments provide an opportunity for language designers to incorporate additions and updates to the language that may not yet have been published, and they provide a standard for all previous work. This might include incorporating subtleties of the language or proofs that may have been discovered only after extensive use and experience with the language. If the designers can enforce these items in the environment, they free the users from having to discover the subtleties for themselves.

The ASTRAL Software Development Environment (SDE) is a tool for the ASTRAL language, which assists in the design, analysis, and reuse of ASTRAL specifications. This chapter discusses the design portion of the SDE, which forms a foundation for the analysis components that are discussed in later chapters.

## 5.1. SDE Overview

The original interface of the SDE was developed by Marco Mussini [Mus 93] and Richard Lee, who worked at Politecnico di Milano and UCSB, respectively. This work defined the look of the SDE interface but did not include any of the current functionality of the SDE other than a hierarchical navigator and a basic syntax-directed editor that did not have all of the features of the editor discussed below. The majority of the work in [Mus 93] was essentially an unusable skeleton of the system that needed to be filled out with actual functionality.

Figure 5.1 shows the user interface to the ASTRAL Software Development Environment. The hub of the SDE is the navigation window located in the upper left portion. The navigation window displays the current specification and allows the user to hierarchically traverse it. By double clicking on a line of the displayed specification, a user can move “up” or “down” in the specification hierarchy of figure 2.2. For instance, figure 5.1 shows the top level of the Gate process in the railroad crossing specification, which was displayed by double clicking on the “top level” line at the process level. The same effect can be achieved by highlighting a line of the specification and using the up and down arrows. By moving up and down in the navigation window, the corresponding portion of the ASTRAL hierarchy for the current specification is displayed and various functions, such as edit, insert, and remove, can be invoked on the highlighted line of the navigation window. For example, if the “Edit” button was pressed in figure 5.1, the editor window would pop up with the schedule text

loaded. Most of the operations of the SDE are linked in some fashion to the navigation window, either as a form of input or as a form of output.

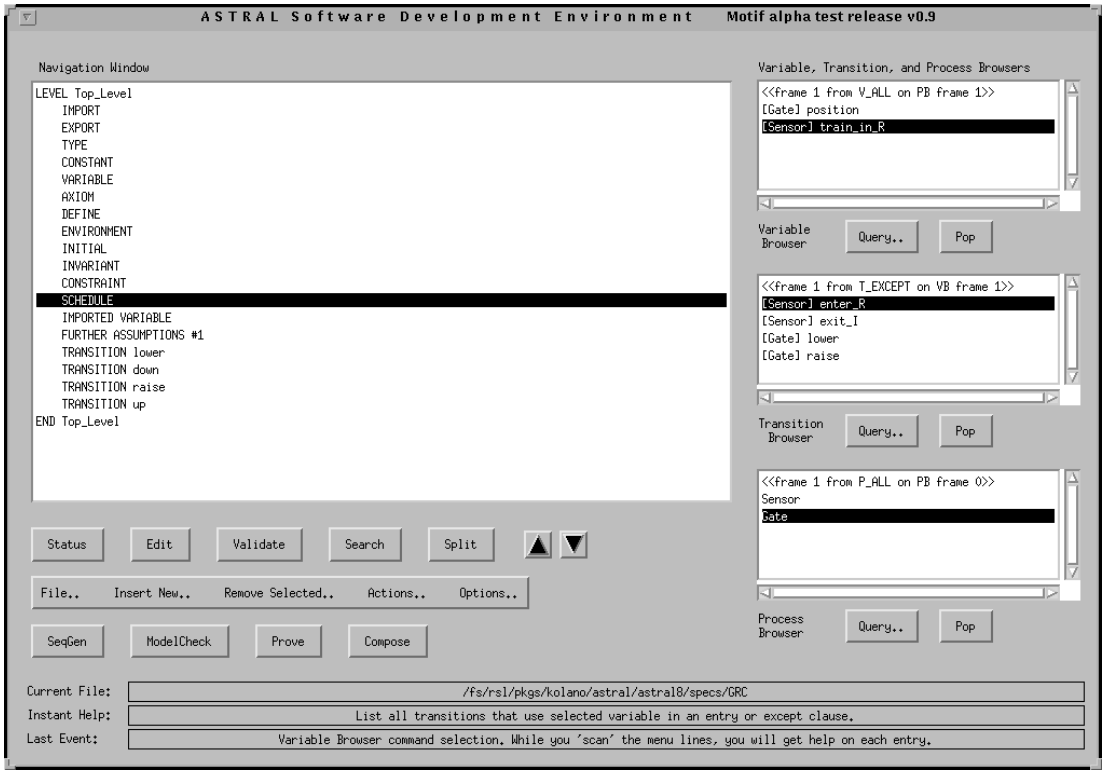



Figure 5.1: The ASTRAL SDE

The top row of buttons in the middle of the SDE are for the most commonly invoked operations in the design phase. The "Status" button brings up the specification manager, which keeps track of changes made to the specification and the current status of proofs. The "Edit" button brings up the syntax-directed editor on the section highlighted in the navigation window. The "Validate" button invokes the specification processing component of the SDE, which brings up a window to report the errors and warnings that result from checking the current specification. By clicking on a result in the error window, the user can move the navigation window to the relevant part of the specification. The "Search" button brings up the search window, which can be used to search and replace expressions throughout the current specification. Finally, the "Split" button invokes the formula splitter, which splits a boolean clause in the navigation window into conjunctive normal form and allows queries to be performed on each split. The lower row of buttons are for commonly invoked operations in the analysis and reuse phases. The "SeqGen" button invokes the transition sequence generator, which displays sequences of transitions that are possible based on a user query. The "ModelCheck" button

brings up the ASTRAL model checker, which can prove or disprove system requirements over finite time intervals for a given set of system constants. The "Prove" button generates the PVS translation of the specification and invokes the PVS theorem prover. Finally, clicking on "Compose" allows the user to work with multiple specifications and composition specific clauses.

The browsers in the right portion of the SDE allow the user to execute a number of predefined queries regarding processes, transitions, and variables and their relationships to each other in the current specification. The result of a query can be clicked on to move the navigation window directly to the appropriate location. The status lines at the bottom of the SDE display various items concerning the SDE, such as the specification currently being displayed in the navigation window, the last event of significance in the SDE, and help lines for different items.

Finally, the menu bar between the rows of buttons contains pull-down menus for various operations, including loading and saving specifications, generating proof obligations, setting SDE options (e.g. read-only to assure that a specification does not get modified), and inserting and removing various objects (e.g. processes, transitions, etc.). The separate components of the SDE are discussed in more detail in the following subsections.

## 5.2. Editor

The SDE editor provides only the most basic functionality of common general-purpose editors such as vi or emacs; however, it is rarely the case that an ASTRAL section is so large as to require the additional functionality provided by general-purpose editors. More importantly, the syntax checking, automatic formatting, and on-line syntax documentation of the SDE editor more than compensate for this absence of additional functionality.

### 5.2.1. Syntax-Directed Editing

All editable items in the SDE are associated with a specific grammar, ranging from the simple alphanumeric constraint on names to the complex grammars of well-formed formulas. Through the use of these grammars, the editor is able to parse its current text and indicate the presence or absence of syntactic errors. Figure 5.2.1 shows the popup window that appears when the edit function is invoked on the section highlighted in the navigation window of figure 5.1. When editing, if the user is unaware of the exact syntax of a section, the "Help" button displays the corresponding grammar and other pertinent information about the section being edited. When the text is correct with respect to its grammar, the "OK" button is displayed at the bottom of the editor. However, when a syntax

error is found, the “Parse Error” button is displayed instead, which is the case in figure 5.2.1. In addition, the line that is believed to contain the error is underlined, to allow the user to quickly locate and correct syntactic errors. Figure 5.2.1 shows the editor with a parsing error present in the text. The error in this case is a parentheses imbalance. The last line is underlined since that is where the missing right parenthesis causes a syntax error.

The grammars are not just important for syntax-directed editing. Having a complete and accurate grammar for the ASTRAL language is essential for the SDE to correctly parse different sections of a specification and create the appropriate parse tree, which is necessary for the other components of the SDE, such as the querying mechanisms and analysis components, to correctly interpret the specification and produce the right results. Many portions of the ASTRAL grammar did not exist or were flawed so they had to be defined from scratch or rewritten, respectively. For example, both the grammar for the implementation clauses and the grammar for transition selection clauses were mentioned informally in [CKM 95] and [CKM 94], but suffered from ambiguities and incompleteness, and needed to be precisely defined in the SDE grammars. The complete grammar for the ASTRAL language is given in appendix E.

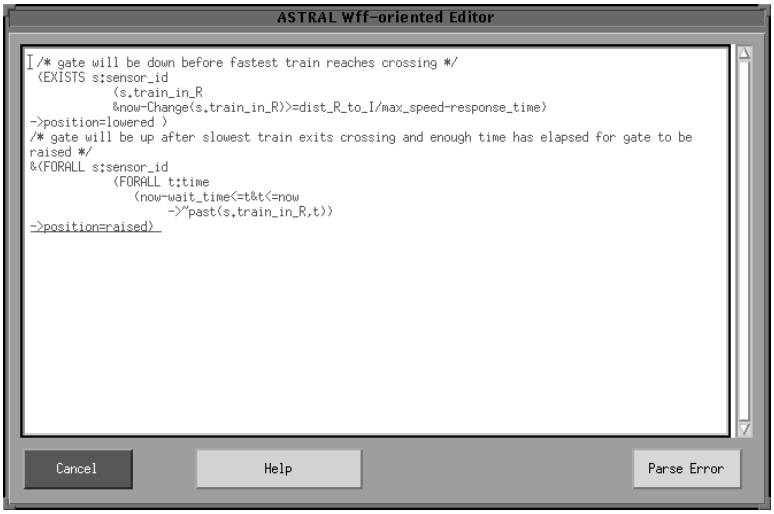



Figure 5.2.1: Editor window for the Gate schedule

Revising the grammars also gave the opportunity to add more expressiveness to some of the grammars. For example, a transition selection statement was informally defined in [CKM 94] as $\{OpSet_i\}<Condition_i>\{ROpSet_i\}$, where $OpSet_i$ is a set of transitions, $ROpSet_i$ is a nonempty subset of $OpSet_i$, and $Condition_i$ is a boolean condition on the state of the process. Any time $OpSet_i$ is enabled in the process and $Condition_i$ holds, then the only transitions that can fire are the transitions

in $ROpSet_i$. In the informal grammar for transition selection clauses, $OpSet_i$ and $ROpSet_i$ were defined as sets of identifiers with a "*" allowed as a wildcard to indicate any set of transitions. For example, in the phone system, "{Give_Dial_Tone, *} TRUE {Give_Dial_Tone}" indicates that Give_Dial_Tone has priority over every other transition. This grammar, however, does not allow much flexibility in defining transition selection statements. For example, suppose a process has n transitions, tr1, ..., trn. A reasonable transition selection statement might be that tr1 should have priority over any other transition, as long as tr2 is not enabled. This statement can only be expressed with the informal grammar by specifying all possible combinations of tr1 with tr3, tr4, ..., trn.

The updated grammar consists of two transition selection forms. The first form is

ET_DECL_LIST & wff → eligible_transitions = TRANS_SET

Here, ET_DECL_LIST is a sequence of constraints about the enabled transitions that are connected by boolean operators, where each constraint is in the form

enabled_transitions SET_OP TRANS_SET

where SET_OP is an ASTRAL set operator and TRANS_SET is a set of transition identifiers. An additional form is allowed for the SUPERSET and CONTAINS operators in which "any_subset(TRANS_SET)" can be used in place of TRANS_SET, where any_subset specifies that any nonempty subset of the given set of transitions satisfies the constraint.

The first transition selection form specifies that whenever the set of enabled transitions satisfies the constraints of ET_DECL_LIST and wff holds, then the transitions that are eligible to fire are those in TRANS_SET. This form is sufficient to specify the above example as shown below.

```
    enabled_transitions CONTAINS {tr1}
&   enabled_transitions ~CONTAINS {tr2}
&   TRUE
→   eligible_transitions = {tr1}
```

Consider the case in which any subset of {tr1, tr2, tr3} has priority over all other transitions. This statement takes seven separate transition selection statements of the first form to specify because each combination of tr1, tr2, and tr3 has to be considered. The second transition selection form allows this transition selection to be specified in a single statement.

```
    ET_DECL_LIST
&   wff
→   eligible_transitions = TRANS_SET INTERSECT enabled_transitions
```

The difference in this form is that the eligible transitions are those in the TRANS_SET intersected with the enabled_transitions. This is useful when an any_subset expression is used in ET_DECL_LIST. The above example can be specified as shown below.

```
        enabled_transitions CONTAINS any_subset({tr1, tr2, tr3})
    &   TRUE
→       eligible_transitions = {tr1, tr2, tr3} INTERSECT enabled_transitions.
```

In this case, {tr1, tr2, tr3} must be intersected with enabled_transitions because the eligible transitions must always be a subset of the enabled transitions and it is not known which subset of {tr1, tr2, tr3} will be enabled.

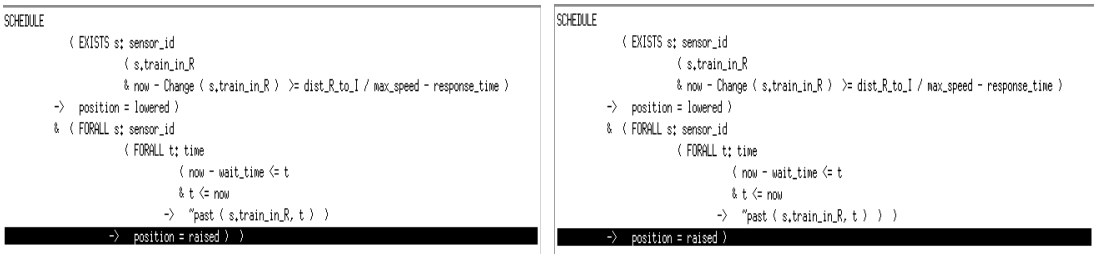
### 5.2.2. Formatting

When first writing specifications, it is more important to concentrate on the content rather than the readability of the specification. This does not mean, however, that readability is unimportant. In fact, an unreadable specification is likely to contribute unnecessary confusion, errors, and additional time to development. Manual formatting is also likely to impact development time, especially during the initial design stages when changes are more likely to occur. In the same way that word wrap (which is also turned on in the editor for this very reason) in word processors allows writers to concentrate on their words instead of where to place carriage returns, so automatic formatting in the SDE allows specifiers to focus not on how the specification is entered but on what it says. When the OK button is pressed during an edit session, the text in the editor replaces the text the editor was originally invoked on. Before the text is replaced, however, the new text is automatically reformatted into a fixed format.

An advantage of automatic formatting, which is not immediately obvious, is that it allows the user to catch semantic errors that might otherwise go undetected in the specification analyzer. As an example, consider the missing parenthesis in the Gate schedule clause that is shown in the editor window of figure 5.2.1. This parenthesis can be placed in a number of different locations to syntactically fix the problem. Figure 5.2.2-1(a) shows the result of formatting in the edit window of figure 5.2.1 (after adding the missing parenthesis to the end). As can be seen from figure 5.2.2-1(a), by adding a parenthesis to the end of the text the highlighted implication is placed in the wrong scope. Figure 5.2.2-1(b) shows the same formula with the parenthesis correctly placed.

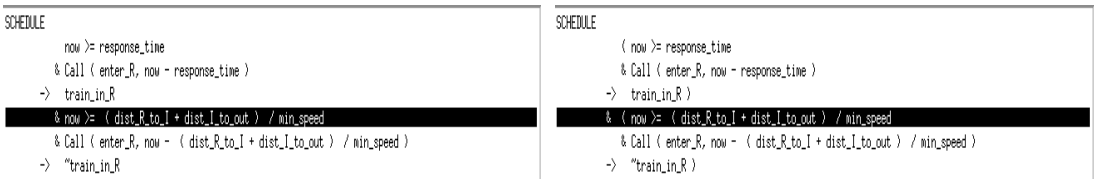
Mistaken operator precedence is another type of error that is usually not detectable in the specification analyzer. Therefore, the formatter indicates the precedence of boolean operators by the distance between the operator and its adjoining text. That is, the closer the operator is to the text, the higher its precedence. In figure 5.2.2-2(a), the highlighted conjunction incorrectly binds more tightly than the two implications surrounding it. In figure 5.2.2-2(b), however, parentheses have corrected the situation and the conjunction now has a lower precedence than the parenthesized expressions.

Both types of errors would most likely go undetected with manual formatting because the user would format the text as s/he assumed it was written, which would be wrong in this case, even though the text was both type correct and syntactically correct.




a) b)

Figure 5.2.2-1: Formatted forms of Gate schedule with misplaced and correctly placed parenthesis




a) b)

Figure 5.2.2-2: Formatted forms of Sensor invariant with scoping error and correction

### 5.2.3. Search and Replace

Although the search function is not directly part of the editor, it shares the two features described above. The search button in the main window brings up the search and replace window, which allows the user to search for and replace regular expressions throughout the entire specification or in a specific portion. While there is nothing revolutionary in its behavior, what is important is that even this procedure has been designed to reduce the possibility of error. To assure that the benefits of syntax-directed editing and formatting are not lost, the replace procedure aborts if the replacement text would cause a syntax error within the section where the replacement is to occur. In addition, the text is reformatted appropriately when any replacements are made.

## 5.3. Validation

One of the most valuable tools that the SDE has to offer is the validation mechanism, which checks for various static errors in the specification such as type errors. When a specification validates without error, it indicates that the specification is ready for the next stage in its development, namely

formal proofs. Similarly, when a composition of system specifications validates without error, it is ready for the construction of the new composite specification.

The bulk of the validation function involves checking that any identifiers used in the specification have been defined in the correct scope and that all operands to both built-in operators and user-defined transitions, defines, etc. have the correct types. Validation also performs other functions such as checking for scope conflicts and warning of missing parameters, which while still well defined in the case of transitions, may not be what the user desired. Figure 5.3 shows a sample validation results window, which demonstrates some of the different types of errors that are reported when the "Validate" button is used.

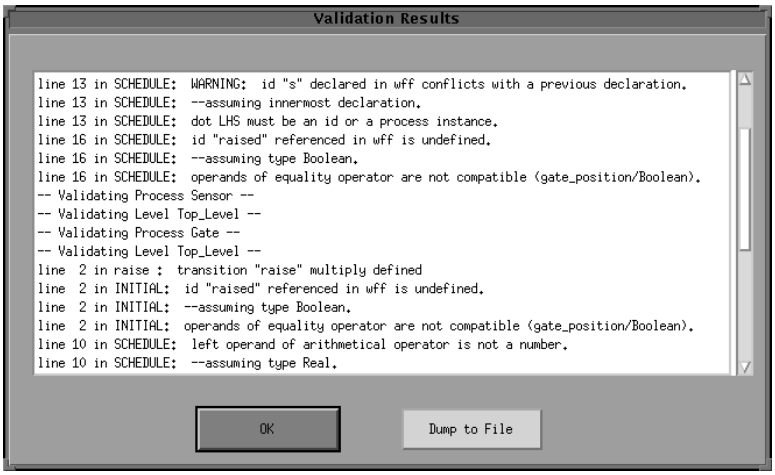



Figure 5.3: The validation results window

In the spirit of "ease of use," entries in the validation window are linked to the navigation window. That is, any error appearing in the validation window can be double clicked, which causes the navigation window to display the corresponding section of the specification and highlight the line at which the error occurred. This is useful for rapidly locating and correcting errors.

## 5.4. Formula Splitter

The formula splitter converts any ASTRAL well-formed formula into conjunctive normal form and then displays each conjunct separately to the user. The splitter can be invoked on any section of an ASTRAL specification that resolves to a boolean expression using the "Split" button. When the splitter is invoked on a section, the splitter window pops up with the splits of that section. For example, consider the splits of the invariant of the P_Robot process in the production cell specification of appendix A. Figure 5.4 shows the window that appears when the splitter is invoked

on this formula. The window shows the 16th split of 17. The other splits can be displayed using the up and down arrows at the bottom of the window. Each split displayed is classified according to the property classification criteria of section 8.3. The split in figure 5.4 is a forward liveness property.

The formula splitter also serves as a "formula browser" that allows queries about formulas in the specification to be performed. For example, the user can display the variables used in the current split by performing the "variables..used in Selected Formula". This functionality is described in section 8.4. Besides being available to the user, the formula splitter is also used by other components of the SDE to perform various formula transformations. For example, during the construction of the composite specification, which is discussed in the next section, the splitter is used to determine which portions of the environment clause should be moved to the imported variable clause. Other components of the SDE also use the splitter to perform various tasks as will be discussed in later chapters.

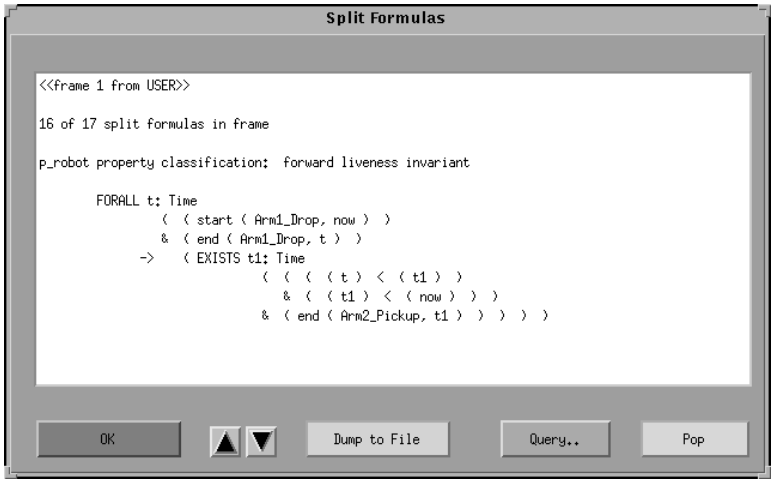



Figure 5.4: The formula splitter window

The splitter algorithm is recursive on the parse tree. The algorithm takes a parse tree node, a negation value, and a time expression and returns a formula in the form ($A_1$ & ... & $A_n$), where each $A_i$ is in the form ($a_i$ | ... | $a_m$) and each $a_i$ is an unsplittable expression. The negation value states whether or not the node is negated. The time expression states the time in the past that the node is to be evaluated at.

The algorithm examines the current node and proceeds based on the type of the node. If the node is a "not" node, the algorithm is recursively invoked on the not operand with a negated negation value. Similarly, if the node is a "past" node, the algorithm is invoked on its first operand with the time

expression set to the time expression of the past node evaluated at the time expression given as a parameter to the algorithm.

If the current node is a boolean operator, the algorithm produces the splits of the left and right operands and then either and-merges them or or-merges them depending on the operator type and the negation value. In an and-merge, two terms ($A_1$ & ... & $A_n$) and ($B_1$ & ... & $B_k$) are combined into a term ($A_1$ & ... & $A_n$ & $B_1$ & ... & $B_k$). For example, and-merge(($a_{11}$ | $a_{12}$) & ($a_{21}$ | $a_{22}$), ($b_{11}$ | $b_{12}$) & ($b_{21}$ | $b_{22}$)) = ($a_{11}$ | $a_{12}$) & ($a_{21}$ | $a_{22}$) & ($b_{11}$ | $b_{12}$) & ($b_{21}$ | $b_{22}$). In an or-merge, two terms ($A_1$ & ... & $A_n$) and ($B_1$ & ... & $B_k$) are combined into a term ($A_1B_1$ & ... & $A_nB_k$), where each term $A_iB_j$ is ($a_{i1}$ | ... | $a_{im}$ | $b_{j1}$ | ... $b_{jl}$). For example, or-merge(($a_{11}$ | $a_{12}$) & ($a_{21}$ | $a_{22}$), ($b_{11}$ | $b_{12}$) & ($b_{21}$ | $b_{22}$)) = ($a_{11}$ | $a_{12}$ | $b_{11}$ | $b_{12}$) & ($a_{11}$ | $a_{12}$ | $b_{21}$ | $b_{22}$) & ($a_{21}$ | $a_{22}$ | $b_{11}$ | $b_{12}$) & ($a_{21}$ | $a_{22}$ | $b_{21}$ | $b_{22}$). Thus, for instance, if the current node is an and operator, the algorithm returns the and-merge of the left and right splits if the current node is not negated and returns the or-merge if the current node is negated.

If the current node is an unnegated universal quantifier or a negated existential quantifier, the quantifier is dropped (i.e. skolemized) and the quantified variables are added to a list of skolem constants. The algorithm then splits the remaining unquantified formula. After all the splits of the root of the parse tree have been produced, the splits are "deskolemized" by universally quantifying over the list of skolem constants at the outermost quantification level of each split. For example, in the split of the following formula

```
FORALL t1: time
    (   start(tr1, t1)
→       FORALL t2: time
            (   t2 < t1
            →       ~start(tr2, t2)))
```

the second quantifier is "lifted" out of the first quantifier as shown below.

```
FORALL t1: time, t2: time
    (   start(tr1, t1)
    &   t2 < t1
→       ~start(tr2, t2)))
```

If the current node is a negated universal quantifier or an unnegated existential quantifier, the formula is unsplittable and the algorithm stops traversing the parse tree.

## 5.5. Composing ASTRAL Specifications

To facilitate reuse and to simplify the construction of large and complex real-time systems, ASTRAL provides the developer with a composition capability. The ASTRAL *compose clause* contains the

necessary information to combine two or more ASTRAL system specifications (i.e. a global specification and its associated collection of process specifications) into a single specification of the combined system. If $S_1$ and $S_2$ denote two ASTRAL top level specifications, then the interaction between the processes of $S_1$ and $S_2$ is described by specifying which exported transitions of the processes of $S_1$ and $S_2$ are no longer exported to the external environment. That is, the stimuli needed to fire these transitions in $S_1$ are produced by processes of the sibling system $S_2$ and vice-versa, rather than by the external environment.

Figure 5.5-1(a) shows two systems, $S_1$ and $S_2$. $S_1$ exports transitions $T_1$ and $T_2$ and state variables $x_1$, $x_2$, and $x_3$, while $S_2$ exports transition $T_3$ and state variables $y_1$ and $y_2$. When $S_1$ and $S_2$ are composed, some transitions of $S_1$ will not require an external call, since $S_2$ is now providing part of the environment in which $S_1$ works. This works similarly for some transitions of $S_2$. For instance, in figure 5.5-1(b), transitions $T_1$ and $T_3$ are no longer exported, since the events that trigger them are now represented by particular values of $y_2$ and $x_1$, $x_3$, respectively. Thus, the composed system, C, will export only transition $T_2$. That is, the external environment of C can call only transition $T_2$ (see figure 5.5-1(c)).

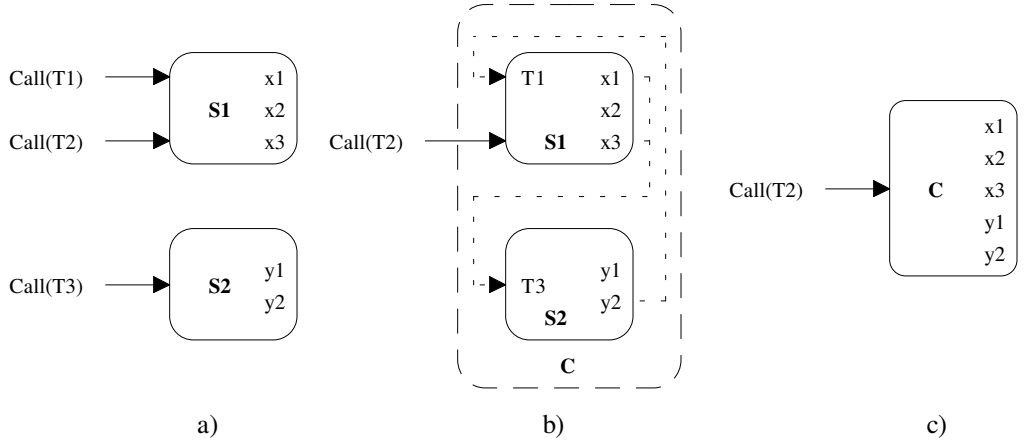



Figure 5.5-1: The composition of $S_1$ and $S_2$ into C

The most important part of the compose section is the *call generation clause*, which describes how exported transitions of $S_1$ processes are triggered by events occurring in $S_2$ processes and vice-versa. These events are described by formulas of the following form:

FORALL t: Time,… $(P(S_1) \leftrightarrow Call(T, t))$,

where $P(S_1)$ is an ASTRAL well-formed formula describing the occurrence of the events in $S_1$ that are equivalent to calling the exported transition T of $S_2$. An example call generation clause is

presented later. The details of the composition clause and the process of automatically composing ASTRAL specifications are presented in [CK 93].

Although ASTRAL allows individual specifications to be composed into a new composite specification, the extensive amount of work required to build the new specification (as described in [CK 93]) may cause users to hesitate before taking advantage of this feature. Also, when constructing the composite specification there are numerous opportunities for errors and omissions. By using the SDE, however, the user is completely relieved of the burden of constructing the new specification.

The "Compose" button sets the SDE into composition mode and allows the manipulation of multiple system specifications in the same fashion as individual specifications by adding a new topmost level to the ASTRAL hierarchy, which contains specifications as its components along with composition specific clauses. Figure 5.5-2 shows the additional composition hierarchy. When validating a composition, the validation procedure examines the declarations in all specifications and warns the user of possible name conflicts. Those declarations that have the same name but different meanings between specifications can be changed by the user using the search and replace feature. Declarations with the same meaning can be left as is and duplicates will be removed automatically during the construction of the composite specification.

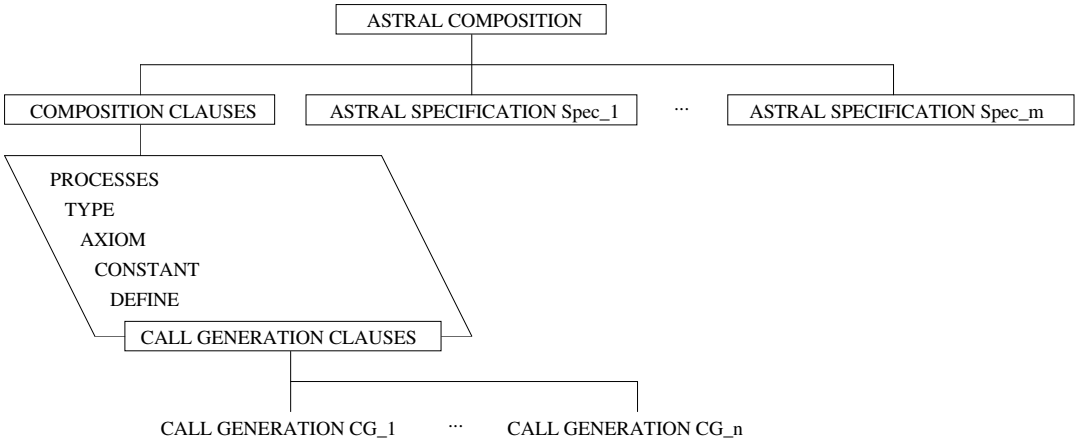



Figure 5.5-2: The composition hierarchy

When the SDE is in composition mode, the compose button is replaced by a "Build" button. The build procedure performs a number of transformations to construct the new composite specification. For example, call statements involving exported transitions that have been made internal must be replaced by an expression derived from the corresponding call generation clause describing how the

transition is invoked internally. However, the call generation clause cannot simply be used as it is declared because it is written for all calls in the system whereas the calls being replaced are specific instances with a particular process id, time, and parameters as well as possibly referencing a number of calls into the past (i.e. using $Call_n$ where $Call_n(T, t)$ holds if the nth call in the past to transition T occurred at time t). The specific changes that must be made to a call generation clause before it can be used were not elaborated in [CK 93]. However, it was necessary to develop algorithms for making these changes when implementing the build functionality in the SDE. For example, suppose the composition section contains the following call generation clause (taken from [CGK 97]):

```
FORALL t: Time, C: Central_Control_ID, L: Line
    (   Change(LD_Unit(C).LocStatus(L), t)
    &   LD_Unit(C).LocStatus(L) = In_Progress
↔       C.Call(Receive_LD(LD_Unit(C).LocOut(L)),t))
```

This means that whenever the variable LD_Unit(C).LocStatus(L) changes to In_Progress, an "internal call" is made to Receive_LD. Furthermore, suppose the following formula is in the schedule clause of one of the original specifications being composed:

$pid.Call_2$(Receive_LD(arg1), time1)

Since Receive_LD is no longer exported in the composite specification, the formula needs to be transformed to an expression in which the external call is replaced by the combination of values described by the call generation clause. An internal call to Receive_LD in process pid with argument arg1 occurs whenever the following formula (call it cg′) holds:

```
EXISTS t: Time, C: Central_Control_ID, L: Line
    (   C = pid
    &   LD_Unit(C).LocOut(L) = arg1
    &   Change(LD_Unit(C).LocStatus(L), t)
    &   LD_Unit(C).LocStatus(L) = In_Progress)
```

To complete the transformation, it must be checked that cg′ holds at time1 and that time1 was the second time in the past that cg′ changed to true. The following abbreviated formula shows the fully modified form:

IF cg′
THEN $Change_3$(cg′, time1)
ELSE $Change_4$(cg′, time1)
FI

If cg′ holds at the current instant, then for $pid.Call_2$(Receive_LD(arg1), time1) to hold, cg′ must have changed three times: once to true at time1, once to false between time1 and the current instant, and finally to true at the current instant. Similarly, cg′ must have changed four times if cg′ does not hold at the current instant. Given the complexity of this simple example, it can be seen that performing

such transformations by hand for complete specifications would take a significant amount of time and effort, not to mention the almost inevitable possibility for error. Even though the exact details of the call generation clause transformations were not discussed in previous work, the update to the ASTRAL language has been incorporated into the SDE. This is an example of how designers using the SDE can have access to the most recent features of the language.

Similar changes need to be made for the environment clauses. That is, the environment is required to satisfy certain conditions to guarantee correct behavior of the system, but when an exported transition becomes internal, assumptions about calls to that transition must now be satisfied by the behavior of other processes in the system rather than by the external environment. Thus, those assumptions must be moved from the environment section to the imported variable clause. This process is performed with the assistance of the formula splitter. The environment clause is first modified according to the call generation clauses. It is then split with the formula splitter and each split is checked for calls to exported transitions. If no such calls are found in the split, then the split is conjoined to the imported variable clause. If a call is found, then the split remains in the environment clause.

Consider the environment clause of the phone system of [CGK 97] as shown below.

```
    FORALL t: Time, L: Connection
        (   Call(Terminate_LD_2(L), t)
    →       EXISTS t1: Time
                (   t1 < t
                &   (   Call(Receive_LD(L), t1)
                    |   Call(Start_LD(L), t1))))
&   FORALL t: Time, L: Connection
        (   Call(Start_Talk_2(L), t)
    →       EXISTS t1: Time
                (   t1 < t
                &   Call(Start_LD(L), t1)))
&   FORALL t: Time, L: Connection
        (   Call(Start_LD(L), t)
    →       EXISTS t1: Time, P: Area_Phone
                (   t1 < t
                &   past(Phone_State(P), t1) = Calling
                &   past(Plug(LDOut_Line(P), L), t)))
&   FORALL t: Time
        (   Call2 (Receive_LD, t)
    →       Call(Receive_LD) - t > LD_Timeout)
```

After the calls to transitions that are no longer exported are transformed to the appropriate call generation expressions and the formula splitter is invoked, four split formulas are generated. The third split formula is shown below.

```
FORALL t: Time, L: Connection
   (   IF      EXISTS t: Time, C: Central_Control_ID, L: Line
                  (   C = self
                  &   LD_Unit(C).LocOut(L) = L
                  &   change(LD_Unit(C).LocStatus(L), t)
                  &   past(LD_Unit(C).LocStatus(L), t) = Connected
                  &   ~change(LD_Unit(C).NetOut(L), t)
                  &   (   LD_Unit(C).NetStatus(L) = Connected
                      |   LD_Unit(C).NetStatus(L) = Available))
       THEN
          change(EXISTS t: Time, C: Central_Control_ID, L: Line
                        (   C = self
                        &   LD_Unit(C).LocOut(L) = L
                        &   change(LD_Unit(C).LocStatus(L), t)
                        &   past(LD_Unit(C).LocStatus(L), t) = Connected
                        &   ~change(LD_Unit(C).NetOut(L), t)
                        &   (   LD_Unit(C).NetStatus(L) = Connected\
                            |   LD_Unit(C).NetStatus(L) = Available)), t)
       ELSE
          change2(EXISTS t: Time, C: Central_Control_ID, L: Line
                        (   C = self
                        &   LD_Unit(C).LocOut(L) = L
                        &   change(LD_Unit(C).LocStatus(L), t)
                        &   past(LD_Unit(C).LocStatus(L), t) = Connected
                        &   ~change(LD_Unit(C).NetOut(L), t)
                        &   (   LD_Unit(C).NetStatus(L) = Connected\
                            |   LD_Unit(C).NetStatus(L) = Available)), t)
       FI
→      EXISTS t1: Time, P: Area_Phone
          (   t1 < t
          &   past(Phone_State(P), t1) = Calling
          &   past(Plug(LDOut_Line(P), L), t)))
```

None of the splits contain any calls to exported transitions so all the splits are rejoined and the entire reformed formula is conjoined to the imported variable clause.

Besides replacing calls with equivalent call generation expressions and moving environmental assumptions to the imported variable clause, the build procedure also performs other transformations. These include removing the no longer exported transitions from the export clause, importing any variables, types, etc. used in the modified clauses, and updating transition entry/exit assertions. In fact, the build procedure performs all of the transformations discussed in [CK 93]. Thus, the user is completely relieved of the burden of producing the composite specification by the build function of the SDE.

## 5.6. Verification Condition Generator

In order to assure that an ASTRAL specification satisfies its requirements, it is necessary to generate and prove the appropriate proof obligations. ASTRAL proofs are divided into three categories: *intra-level* proofs, *inter-level* proofs, and *composition* proofs. The intra-level proof obligations deal with proving that each process level satisfies its stated critical requirements and that the top level specification is consistent and satisfies the global requirements. The inter-level proof obligations deal with proving for each process type that the specification of level i+1 is consistent with the specification of level i. The composition proof obligations deal with proving that the assumptions of each of the components of the composite system are satisfied by the other components in the system that replace what was previously the external environment. Details of the three types of proof obligations can be found in [CKM 94], [CKM 95], and [CK 93], respectively.

The proof obligations for ASTRAL are relatively straightforward, but in many cases are rather lengthy, which means they are prone to error. By generating the appropriate proof obligations automatically, not only is the user relieved of the time involved, but also the proof obligations are guaranteed to be accurate. The SDE generates all three types of ASTRAL proof obligations. The intra-level proof obligations have also been encoded in the theorem prover. The inter-level and composition proof obligations, however, have not yet been defined in the theorem prover portion of the SDE, which is discussed later. Thus, until the theorem prover includes these definitions, the user can still obtain the necessary proof obligations by using the verification condition generator (VCG).

## 5.7. Specification Manager

The “Status” button invokes the specification manager of the SDE. This tool displays various information about the current specification that is stored whenever a specification is saved. The two main types of information are dates and statuses. The dates indicate when various activities have been performed on the specification. These include the date that the specification has been validated, the dates that different clauses have been changed, the dates that the requirements have been model checked, etc. The statuses indicate what the results of the various activities were. These include whether the specification is valid, whether errors have been found using the model checker, how many proof obligations have been discharged with the theorem prover, etc. The proof manager also displays other information such as the context directory associated with the specification, which

indicates where the theorem prover specifications are located. Figure 5.7-1 shows the window of the specification manager.

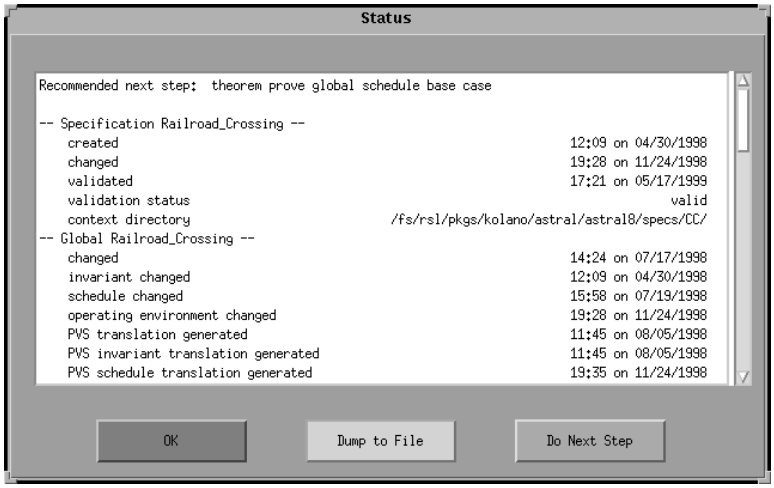



Figure 5.7-1: The specification manager

The information of the specification manager is used by other components of the SDE to perform various functions when needed. For example, most of the advanced components of the SDE, such as the model checker and theorem prover, require that a specification is valid when they are invoked. When a specification is loaded, its validation status is checked. If the specification is valid and has not changed since the time it was validated, the validation procedure is run on the specification so that these advanced components can be invoked immediately after loading. Another component that uses the specification manager information is the transition sequence generator. The sequence generator invokes the theorem prover to attempt the proofs of transition successor obligations, which can be an expensive procedure. The sequence generator uses the transition change dates to determine which transitions have changed since the obligations were last proved. Thus, if only a few transitions have been changed, it is not necessary to rerun the bulk of the transition successor obligations. The change dates are also used when the theorem prover is invoked to determine which portions of the specification need to be translated into the language of the prover. The theorem prover saves specifications that have already been typechecked in a binary form that can be loaded quickly. Since only the portions of the current specification that have changed since the last translation are generated, more of the prover's binary files can be used.

Most of the components of the SDE have been written from scratch, thus they could be tightly integrated. This integration allows components to be easily invoked by each other and allows the

statuses associated with each component to be easily retrievable by the specification manager. The theorem proving component of the SDE, however, is a stand-alone tool that was developed at SRI International. Since the theorem prover was developed elsewhere and its source code is not freely available, invoking it and retrieving its associated status information is more difficult. These statuses consist of the completion status of each proof obligation. In PVS, the ".pvscontext" file stores this and other information about the proof obligations in the PVS specifications of a given directory (i.e. context). The format of this file, however, is proprietary and like the source code is not freely available, thus cannot be used to directly retrieve the proof status. Instead, it is necessary to retrieve the information using the standard PVS interface. It is undesirable for the user to have a part in retrieving this information, thus a suitable degree of integration between the SDE and PVS was necessary.

In essence, there are three operations that were considered to be necessary in order to achieve this level of integration. First, it is necessary to be able to retrieve the completion status of all proofs so that the specification manager can direct the user appropriately as to the step to perform next. Second, it is necessary to be able to change the current context of PVS so that the same PVS session can be used for any sequence of specifications that is loaded in the SDE. This is desirable since PVS is a resource intensive program and it is expensive to invoke and exit PVS multiple times or to have multiple PVS sessions running. Finally, it is necessary to be able to attempt the proofs of all the sequence generator successor obligations and retrieve their status upon completion in order to make the sequence generation process completely automatic.

The first step in achieving this functionality is to allow both the user and the SDE to interact with the same PVS session. Expect [Lib 97] is a scripting language that allows interactive programs to be controlled noninteractively. An additional feature of expect is that it allows two users to control the same program. In this case, the same PVS session can be controlled by two users, where one "user" is the SDE while the other user is the human. The SDE also utilizes expect to issue commands to PVS. Each of the three operations discussed above was added to the PVS interface via the appropriate emacs extensions that are invoked by the SDE when necessary. Upon the completion of each command, an appropriate file is created as a flag to signal the SDE that it has completed the operation. With this interface in place, the specification manager can retrieve the appropriate proof status information, which is done whenever the human quits the SDE, changes context (e.g. loads a new specification into the SDE), or finishes a proof session.

Besides displaying various information about the current specification, the specification manager also directs the user as to which design or analysis step should be performed next. The recommended next step is displayed at the top of the specification manager window as shown in figure 5.7-1. The recommendation is updated according to the operations that the user has performed on the current specification. The user can invoke the SDE operation that is associated with the recommended step directly using the "Do Next Step" button. Figure 5.7-2 shows the hierarchy of steps that is used to recommend the next step for a specification with n processes where each process $P_i$ has $k_i$ levels. Higher steps in the hierarchy are performed before lower steps and steps on the same level are performed left to right. Processes are numbered according to a dependency graph, where the graph is constructed by adding a node for each process and an edge from a process P to a process Q if P imports a variable or transition from Q. If $P_i$ and $P_j$ are two processes in the hierarchy with $i < j$, then there is either a path from $P_i$ to $P_j$ in the dependency graph or there is not a path from $P_j$ to $P_i$.

Some of the orderings in the hierarchy are based on the natural ordering of operations in the SDE. For example, the current specification must always be validated before any of the analysis components can be invoked on it. In the composition portion of the hierarchy, compose must always be invoked before build since build can only be invoked on compositions. The orderings that did not naturally fall in the hierarchy were chosen to achieve the most efficient analysis of the specification possible. The process orderings, discussed in section 9.2.1, and intra-level/inter-level obligation orderings minimize the number of proofs that must be redone when an error is found. Note that the ordering of the individual proof obligations in each process is not shown in the hierarchy, but is taken into account by the specification manager. The intra-level proofs of the top level of each process depend on each other, as discussed in section 9.2.1, so they are performed before the proofs of any other level. The lower level proofs of each process are essentially independent of each other so can potentially be interleaved arbitrarily although the hierarchy shows that the proofs of each process are completed before moving on to the next process.

In the proof chain of a single process, the intra-level and inter-level proofs are performed in alternating fashion from the top level to the bottom level. The inter-level obligations between levels i and i+1 may require the behavior of level i+1 to be modified or additional invariants in level i+1 to complete the proof, which would invalidate the intra-level proofs of level i+1. Thus, the inter-level proofs between levels i and i+1 should be performed before the intra-level proofs of level i+1. Similarly, the intra-level obligations of level i may require the behavior of level i to be modified, which would invalidate any inter-level proofs between levels i and i+1. Thus, the intra-level proofs

of level i should be performed before the inter-level proofs between levels i and i+1. For the same reason, the intra-level proofs of level i should be performed before the intra-level proofs of level i+1.

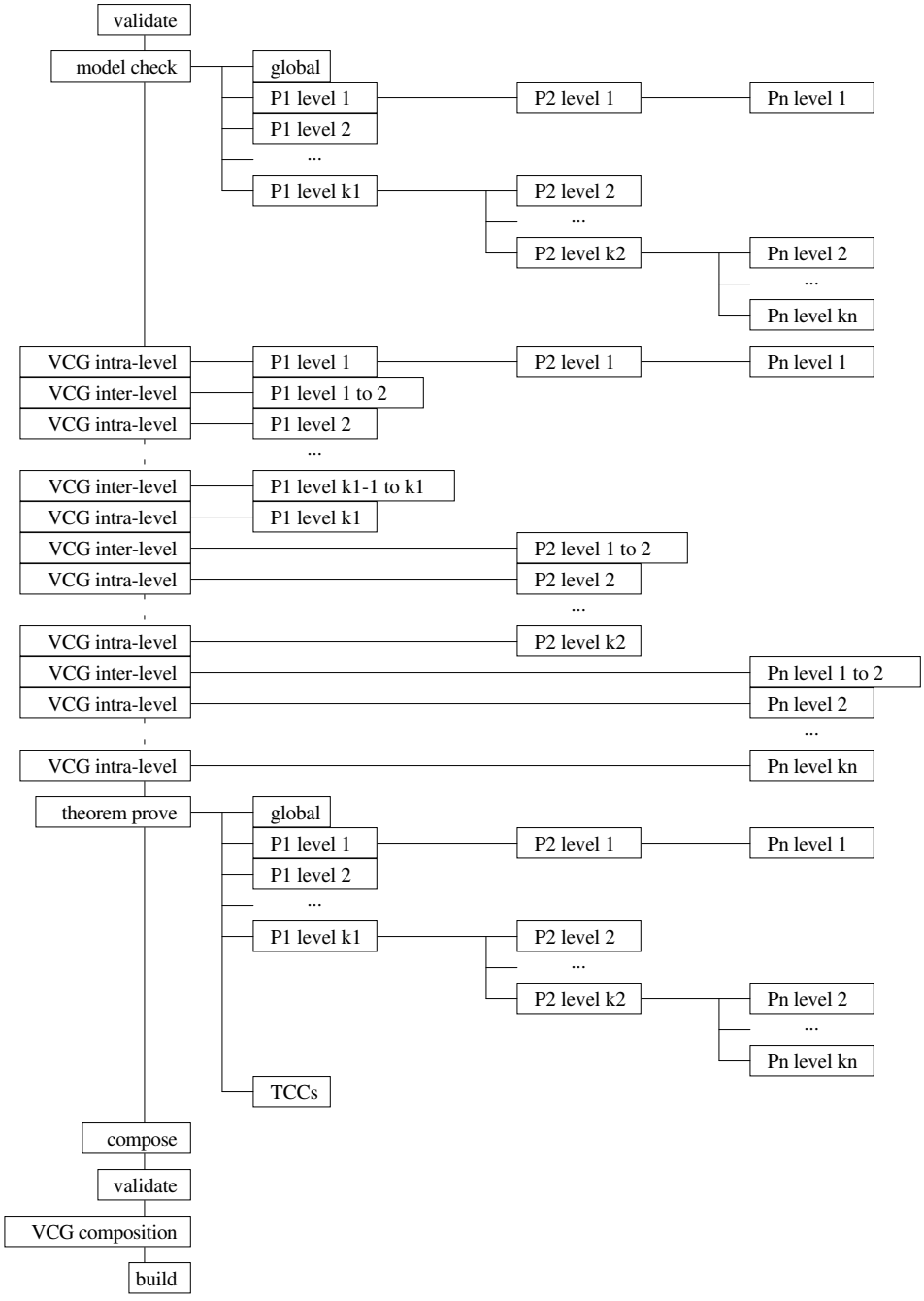



Figure 5.7-2: The do next step hierarchy

The model checker stage, discussed in section 9.1, is higher in the hierarchy than the proof sketch stage (represented by the VCG boxes) and the theorem prover stage because it is an almost fully automated procedure that can find errors effectively with minimal effort. The proof sketch stage, discussed in section 9.2, is higher than the theorem prover stage because hand proofs can be performed more quickly than theorem prover proofs and can serve as the basis of a plan of attack in the theorem prover stage. The theorem prover stage, discussed in chapter ten, is the last stage before composition because it is the final phase of analysis and represents the maximal level of assurance upon completion.

Note that the hierarchy of figure 5.7-2 only shows the steps that are currently available in the SDE and does not represent all possible steps. For example, the model checker and theorem prover do not yet support the proofs of inter-level proof obligations. When this functionality is added to these components, the hierarchy can be expanded.

# Chapter 6

# ASTRAL Semantics in PVS

To perform formal analysis in a specification language, the semantics of the language must be formally defined. If the semantics are defined in a "pencil and paper" fashion, many semantic details may only be mentioned implicitly, which leaves room for ambiguous interpretations and defeats the purpose of defining the semantics. By defining the semantics within the language of a mechanical theorem prover, however, each aspect of the semantics must be completely defined. In addition, every time the theorem prover is used for analysis, the semantics are rigorously tested since they must be used explicitly throughout every proof attempt. In contrast, in proofs by hand with a pencil and paper semantics, the semantics are usually applied implicitly. That is, a user familiar with the language does not refer to the semantics definition explicitly but instead uses his/her own understanding of the semantics. In this case, there is almost no possibility for uncovering errors and omissions within the semantics definition.

The original pencil and paper semantics of ASTRAL had a number of soundness and completeness errors and left room for ambiguous interpretations of certain portions of the language. These semantics were revised and expanded and were encoded within the language of a mechanical theorem prover. After investigating a number of general-purpose theorem provers, PVS [COR 95] was considered ideal for ASTRAL given its powerful typing system, higher-order facilities, heavily automated decision procedures, and ease of use. Other theorem provers that were considered included HOL [GM 93] and ACL2 [KM 96]. HOL does not have the usability of PVS and its decision procedures are not as powerful [Gor 95]. ACL2 is also not as usable as PVS and has limited or no support for arbitrary quantification and real numbers [You 96].

A number of other formal specification languages have been encoded into theorem provers. The temporal logics TRIO [AGM 97] and DC [SS 94] have been encoded into PVS as will be discussed in section 6.3.1. TRIO is a lower-level formalism than ASTRAL and DC is not as expressive. Several real-time state machine languages have also been encoded into theorem provers. The Timed Automata Model has been encoded into PVS [AH 96] and Timed Transition Systems into HOL [HCH

93]. These languages are based on interleaved concurrency, however, which makes their semantics simpler than those of ASTRAL. Additionally, Timed Transition Systems are not defined in terms of arbitrary first-order logic expressions and do not have the complex subtyping mechanisms that are available in ASTRAL.

An encoding of ASTRAL into PVS was reported in [Bun 96] and [Bun 97], but this encoding is based on a definition of ASTRAL that has been developed independently at Delft University based on earlier ASTRAL work in [GK 91a] and [GK 91b]. The ASTRAL definition in these papers did not include the notion of an external environment, thus did not include the call operator, environmental assumptions, or schedules. The Delft definition has diverged from the work reported in [CGK 97] and [CKM 94] and has essentially become a different language. It includes only a small subset of the full set of ASTRAL operators and typing options, does not include all of the sections of an ASTRAL specification, and defines only a small fraction of the ASTRAL axiomatization. In addition, it is based on a discrete time domain and proofs are performed with a global view of the system rather than using a modular approach.

## 6.1. PVS

The Prototype Verification System (PVS) [OSR 98b] is a powerful interactive theorem prover based on typed higher-order logic. A PVS specification [OSR 98a] consists of a modular collection of *theories*. A theory may be parameterized to support polymorphism. Declarations in one theory can be referenced in another theory by using an *importing clause*. Parameterized theories can be imported either with explicit parameters or without parameters. If left without parameters, PVS attempts to instantiate the theory based on the use of its declarations within the importing theory. Most single parameter theories can be instantiated automatically by PVS, but theories with complex or multiple parameters often need to be instantiated explicitly in the referring theory.

A PVS theory declaration consists of a set of types, constants, axioms, and theorems. PVS has a very expressive typing language, which includes functions, arrays, sets, tuples, enumerated types, and predicate subtypes. Types may be *interpreted* or *uninterpreted*. Interpreted types are defined based on existing types, while uninterpreted types must be defined axiomatically. Predicate subtypes allow the expression of complex types that must satisfy a given constraint. For example, the even numbers can be defined as shown below.

```
even_int: TYPE = {i: int | (EXISTS (j: int): 2 * j = i)}
```

For any assignment or substitution that involves a predicate subtype, PVS generates *type correctness conditions* (TCCs), which are obligations that must be proved in order for the rest of the proof to be valid. For example, consider the following declaration.

```
e_plus_2(e: even_int): even_int = e + 2
```

PVS generates the following TCC for the definition of e_plus_2.

```
% Subtype TCC generated (line 7) for  e + 2
        e_plus_2_TCC1: OBLIGATION
            (FORALL (e: even_int): (EXISTS (j: int): 2 * j = e + 2))
```

That is, it must be shown that adding two to an even number is still an even number. Otherwise, the definition of e_plus_2 violates its stated type.

Like types, constants can either be interpreted or uninterpreted. The value of an interpreted constant is stated explicitly, whereas the value of an uninterpreted constant is defined axiomatically. For example, the definition of push in

```
stack: TYPE = list[T];
push(e: T, s: stack): stack = cons(e, s);
```

is an interpreted constant, because the exact effect of a push statement can be determined by expanding its definition. The definition of push in

```
stack: TYPE;
push: [[T, stack] → stack];
```

is uninterpreted because all that is known about push is that applying it to a tuple of type [T, stack] returns a stack of unknown content. In the former definition, the exact consequence of the push operation is given in terms of list operations. To express properties about an uninterpreted constant, however, axioms must be used. For example, in the previous declaration, the following would be appropriate.

```
top_of_push: AXIOM
    top(push(e, s)) = e
```

This states that no matter how stack, push, and top are implemented, applying top to the stack resulting from a push operation will result in the element just pushed. In general, axioms describe anything that is considered to be a "truth" in a theory. Besides types, constants, and axioms, the other basic component of a theory are theorems, which are hypotheses that are thought to be true, but that need to be proven with the help of the prover.

When the PVS prover [SOR 98] is invoked on a theorem, the theorem is displayed in the form of a *sequent*. A sequent consists of a set of *antecedents* and a set of *consequents*, where if $A_1$, ..., $A_n$ are antecedents and $C_1$, ..., $C_n$ are consequents in the current sequent, then the current goal is ($A_1$ & ... &

$A_n) \rightarrow (C_1 \mid ... \mid C_n)$. It is the user's job to direct PVS with prover commands such as instantiating quantifiers and introducing lemmas to show that either (1) there exists an i such that $A_i$ is false, (2) there exists an i such that $C_i$ is true, or (3) there exists a pair (i, j) such that $A_i = C_j$. PVS maintains a proof tree, which consists of all of the subgoals generated during a proof. Initially, when the prover is invoked on a theorem, the proof tree contains only the sequent form of that theorem. As the proof proceeds, subgoals may be generated and proved. To prove that a particular goal in the proof tree holds, all of its subgoals must be proved true. PVS allows the user to define *strategies*, which are collections of prover commands that can be used to automate frequently occurring proof patterns.

## 6.2. Problems with Original ASTRAL Semantics

Before ASTRAL could be encoded into PVS, it was necessary to determine its precise formal semantics. Since the semantics of ASTRAL had been addressed previously in [CKM 94] and [CSK 94], the first attempt was to simply translate these definitions. Unfortunately, both sets of semantics were determined to be inadequate for translation into PVS.

### 6.2.1. Soundness Problems

In [CKM 94], an axiom system is introduced for the ASTRAL abstract machine. While investigating its definition for use with PVS, however, the axiom system given was determined to be neither sound nor complete. For the invariant proofs, three axioms are given stating (1) that the end of a transition occurs at its given duration after the last start, (2) that when a process is idle and some transition is enabled, then some transition fires, and (3) that transitions are nonoverlapping on a single process instance. For the schedule proofs, four axioms are given, which include the three axioms of the invariant proofs, with the definition of "enabled" in (2) modified to include requiring a call to have been issued, along with a fourth axiom describing what it means for a call to have been issued. Consider the following process.

```
PROCESS SPECIFICATION    P
   EXPORT
      T
   TRANSITION   T
      ENTRY     [TIME:   1]
         TRUE
      EXIT
         TRUE
```

Under the axiom system in [CKM 94], it is possible to prove the following invariant.

$$Start(T, now - 1) \rightarrow Start(T, now)$$

That is, that T fires cyclically every time unit. This is provable because by the invariant axioms, if T starts one time unit in the past, then P is idle at the current time. Since the invariant axioms do not mention calls at all, T is enabled at the current instant, since its entry assertion is true and there is only one transition in P, so T must fire. Thus, the invariant holds. The invariant should not hold, however, because T is an exported transition and thus can only start after it has been called from the environment. An invariant must hold regardless of the operating environment, and since one possible environment is that T is not called between now - 1 and now, the invariant is false. Since the invariant could be proven from the axioms, however, the axiom system is unsound.

An additional soundness problem is present in the imported variable obligation. This obligation states that to prove an imported variable clause of a process P, the global schedule and the schedules of all processes besides P can be used. Since the imported variable clause can be used to prove local schedules and local schedules can be used to prove global schedules, however, this creates two types of circular dependencies and permits unsound proofs. For example, the following two process definitions illustrate the first type of dependency.

```
PROCESS SPECIFICATION    P1          PROCESS SPECIFICATION    P2
   IMPORTED VARIABLE CLAUSE             IMPORTED VARIABLE CLAUSE
      TRUE = FALSE                         TRUE = FALSE
   SCHEDULE                             SCHEDULE
      TRUE = FALSE                         TRUE = FALSE
```

The schedule obligations of P1 and P2 hold by their respective imported variable clauses. The imported variable obligations of P1 and P2 hold by the schedule of P2 and the schedule of P1, respectively. Therefore, it is possible to derive that true and false are equal. Thus, the imported variable assumption should not allow the schedules of other processes to be used as assumptions.

The second type of dependency is illustrated by the following specification.

```
GLOBAL                               PROCESS SPECIFICATION    P
   SCHEDULE                             IMPORTED VARIABLE CLAUSE
      TRUE = FALSE                         TRUE = FALSE
                                        SCHEDULE
                                           TRUE = FALSE
```

In this case, the global schedule obligation holds by the schedule of P. The schedule obligation of P holds by its imported variable clause. The imported variable obligation of P holds by the global schedule. Thus, the imported variable obligation should also not allow the global schedule to be used as an assumption.

### 6.2.2. Completeness Problems

The axiom system of [CKM 94] is also incomplete in several ways. For example, there is no way to derive that if a transition starts, then its entry assertion held at that time. This is because the firing axiom is an implication instead of an if and only if, so it can only be derived that if a transition is enabled, it might fire and not vice-versa. It also lacks any axioms about transitions imported from other processes. Even though processes are essentially independent entities, there are still some facts that may be derived about imported transitions in all cases. For example, it is known that imported transitions are nonoverlapping on the same process instance. In [CKM 94], only local transitions were formalized in this manner. Before ASTRAL could be encoded into the PVS logic, the definition of the ASTRAL semantics needed to be revised.

### 6.2.3. Encoding Problems

In [CSK 94], axiomatic and model-theoretic semantics for ASTRAL are given. Most of the axioms and inference rules defined, however, are for the *base logic* of ASTRAL. That is, they provide a framework for interpreting the well-formed formulas of ASTRAL. For example, the domain of time is addressed as is the meaning of temporal formulas that reference variable values at times beyond the current instant (i.e. now) and the corresponding three-valued logic that is necessary to interpret formulas containing these "undefined" values. The base logic, however, does not include the abstract machine of ASTRAL, which is also addressed in [CSK 94], but suffers from many of the same problems as those in [CKM 94].

The axiom system defined for the base logic has been proved sound and complete. For this reason, it was desirable to encode these semantics straight into PVS to take advantage of the already existing proofs. Unfortunately, it was not obvious how some of the axioms and inference rules could be encoded. For example, some of the axioms are context dependent. That is, there must be some examination of the formulas involved in the axioms before they can be applied. One such axiom is that "past(FORALL x A, v)" is equivalent to "FORALL x past(A, v)" *if x is not free in v*. Another example is that a formula is defined *if it does not contain any occurrences of past(w, v)*. For these axioms, it is clear when they can be applied during proofs by hand, but it is not clear how the underlined portions can be encoded directly into the language of a mechanical theorem prover without severely compromising the readability of the encoding. In addition, it was also somewhat undesirable to have to encode all of the axioms and inference rules for the ASTRAL base logic, since most of them are essentially just those of first-order logic. PVS already has first-order logic axioms

and inference rules defined internally so it was preferable to take advantage of the already available PVS framework.

## 6.3. Encoding Issues

While encoding ASTRAL within PVS, a number of issues arose that needed to be handled. Several of these issues are not exclusive to ASTRAL and occur in many different real-time specification languages. The following sections discuss some of these issues and how they were handled in the ASTRAL encoding.

### 6.3.1. Formulas as Types

In many real-time specification languages, a single formula may have multiple values depending on the temporal context in which it is evaluated. Depending on the language, the temporal context may be an explicit clock variable, or implicitly derivable from the formula. To encode such languages into a theorem prover, it is necessary to define formulas as types that can be evaluated in different contexts.

Two different approaches have been used to encode formulas as types in PVS. In the TRIO to PVS encoding [AGM 97], an uninterpreted "TRIO_formula" type is introduced to handle this issue. In TRIO, which is discussed in section 3.1.1.2, the current time is always implicit, but the values of formulas in the past and future can be obtained relative to the current time using the *dist* operator, dist(A, t), which takes a formula A and a relative time t and gives the value of A at t time units from the current time. In the TRIO encoding, the dist operator is defined as a function of type [[TRIO_formula, time] $\rightarrow$ TRIO_formula]. Axioms are defined to transform elements of type TRIO_formula to other elements of type TRIO_formula. Eventually, there must be a valuation from TRIO_formulas to real-world values (i.e. booleans, integers, etc.) so that the decision procedures of PVS can be invoked. Hence a valuation function is defined that takes a TRIO_formula and produces the corresponding boolean value assuming an initial context of the current time instant.

The Duration Calculus (DC) [ZHR 91] is another real-time language that has been encoded into PVS [SS 94]. DC is an implicit-time interval temporal logic in which the current interval is not explicitly known. Rather than using uninterpreted types to define formulas, however, the DC encoding takes advantage of the higher-order capabilities of PVS and defines formulas as functions of type [Interval $\rightarrow$ bool]. DC operators are defined as Curried functions, which when given their original operands, return a function from an Interval to the original range of the operator. For example, the disjunction operator "\/" is defined as "\/(A, B)(i): bool = A(i) OR B(i)", where A and B are of the type [Interval

→ bool] and i is of type Interval. Using this technique, the resulting functions can be combined normally, while still delaying the evaluation of the whole expression until a temporal context is given. Eventually, when a specific interval is given, an actual boolean value is obtained.

For ASTRAL, the DC approach was chosen for several reasons. Since TRIO is an implicit-time temporal logic, one of the main motivations of the TRIO encoding was to keep the actual current time hidden. In ASTRAL, the current time can be explicitly referenced using the variable now, thus it was unnecessary to keep the time hidden. Another disadvantage of the TRIO encoding is that all of the axioms of first-order logic needed to be explicitly encoded into PVS to manipulate the TRIO_formula type. Using the DC encoding style, however, the built-in PVS framework could be utilized, which includes all first-order logic axioms.

All ASTRAL operators have been defined as Curried functions from their operand domains to the type [time → range]. For example, the ASTRAL operator Start(btr1, t1) takes a transition btr1 and a time t1 and returns true if and only if the last start of btr1 was at t1. Its PVS counterpart, Start1(btr1, at1) takes a transition btr1 and an operand at1 of type [time → time] and returns a function of type [time → bool] such that when an evaluation time t1 is given will return true if and only if the last start of btr1 at time t1 was at time at1(t1). In the Start1 definition, shown below, as well as the definitions of all ASTRAL operators that take a time operand, the time operand is itself of type [time → time] and is only evaluated after an evaluation context is provided.

```
Start1(btr1: {tr1: transition | Base_Trans(tr1) = tr1}, at1: [time → time])
      (t1: {t1: time | at1(t1) ≤ t1}): bool =
   (EXISTS (tr1: transition):
      Base_Trans(tr1) = btr1 AND
      Fired(tr1, at1(t1))) AND
   (FORALL (t2: time):
      at1(t1) < t2 AND t2 ≤ t1 IMPLIES
         (FORALL (tr1: transition):
            Base_Trans(tr1) = btr1 IMPLIES
               NOT Fired(tr1, t2)))
```

The definition of Start1 is complicated by the need to handle the definition of transition exceptions. In ASTRAL, it is not possible to assert anything about the start times of an exception. That is, assertions can only be made about the start times of *base transitions*, where the base transition for a given entry or except clause is the transition that the clause is defined in. Thus, in ASTRAL, Start(btr1, t1) actually states that btr1 fired because of its entry assertion *or any one of its exceptions*. It is convenient to think of exceptions as separate transitions in the encoding, however, because just like a transition with a single entry assertion, they each have a precondition, a postcondition, and a

duration, and must be mutually exclusive within the same process instance. The definition of Start1 takes this into account. It says that for at1(t1) to be the last time btr1 started, its entry or one of its exceptions must have fired and no entry or exception has fired from that time up until the evaluation time t1.

With the operators defined in this manner, it is possible to combine ASTRAL operators in standard ways and yet still produce an expression that will only be evaluated once its temporal context is given. The explicit operator definitions also allow all expressions translated from ASTRAL to PVS to be easily expanded and reduced via the built-in mechanisms of PVS. The resulting encoding is very close to the ASTRAL base logic with only slight syntactic differences and allows a specifier who is familiar with the ASTRAL language to easily read the PVS expressions of ASTRAL formulas.

### 6.3.2. Partial Functions

Some specification languages such as Z [Spi 90] allow the definition of partial functions (i.e. functions that are only well defined at certain points) within specifications. Unlike some other theorem provers, PVS does not support the use of partial functions directly. To encode languages that allow the definition of partial functions or whose operators themselves may be partial functions into PVS, alternative approaches must be used. In lieu of partial functions, PVS has a very powerful predicate subtyping system that allows functions to be declared with domains composed of only those elements satisfying a given predicate, such as only those elements for which a function is well defined. The user then proves TCC obligations that the operand of each function satisfies the given predicate. For a specific class of functions, such as boolean functions, an alternative to predicate subtyping is to define a new domain that contains an additional undefined element and then modify the operators for that class of functions to use the new domain. For example, for boolean partial functions, a three-valued domain of {true, false, undefined} can be defined in PVS with boolean operators modified to work with the new domain.

The partial functions in ASTRAL are the operators that take a time as an argument. In ASTRAL, only times in the past may be referenced, thus any formula that references a time beyond the value of now is undefined. In encoding these operators into PVS, the choice was made to use the subtyping mechanism of PVS for similar reasons as the choice to use the DC encoding style. Namely, it was preferable to rely on the existing PVS framework as much as possible. There were also a number of disadvantages to explicitly adding an undefined value and then modifying the appropriate operators. For instance, many additional axioms would need to be added to derive and manipulate expressions containing the undefined element. The main drawback, however, is that the ASTRAL *past* operator,

past(A, t), which takes an expression A and a time t and returns the value of A at t, is a polymorphic function. That is, the past operator can have multiple types depending on the type of A. Since past takes a time, it is undefined when t is greater than now. Since A can be of any type, essentially every type in the specification and hence every operator in the language would need to be redefined using an undefined element. This was highly undesirable and would have unnecessarily complicated both the encoding and the resulting proofs.

Instead, by using the PVS subtyping mechanism, the user must prove TCCs showing that the time operand of any timed operator used in a specification is less than or equal to the temporal context given to the operator. Most of these obligations will be trivial given that the time operands are usually based on now directly or on a quantified time variable that was appropriately limited.

The definition of the Start1 operator in the previous section demonstrates the use of the subtyping mechanism. The time operand of the Start1 function, at1, is of type [time → time] and is only evaluated after an evaluation context is provided. Since it is not known whether at1(t1) will be a valid operand or not (i.e. will cause the expression to be undefined), t1 is limited by the PVS typing system to be greater than or equal to at1(t1). It is then the user's job to show via a TCC obligation that any evaluation times of a Start1 expression occurring in a specification are permissible. The other timed operators of ASTRAL are defined in a similar manner.

### 6.3.3. Noninterleaved Concurrency

Concurrency in real-time systems can be represented by either an interleaved or a noninterleaved model. In an interleaved model, concurrent events occur sequentially between changes to time, while in a noninterleaved model, concurrent events occur simultaneously without an implied ordering. Timed state-machine languages that use an interleaved model of concurrency use an explicit "tick" transition to change time. The combination of the implied ordering of interleaved concurrency and the use of a tick transition allows the semantics of interleaved timed state-machine languages to be simplified significantly over their noninterleaved counterparts because a system execution can be represented as a sequence of transitions rather than an interval of time in which one or more events occur or do not occur at each time. The proof obligations for such languages are also correspondingly simplified since they can be inductive on the nth transition to fire rather than a full induction on a possibly dense time domain.

In ASTRAL, the proof obligations are carried out modularly by proving the properties of each process individually and then proving global properties based on the collection of process properties.

Although the sequence of transitions that fire in a particular process can be represented by an interleaved model since transition execution is nonoverlapping, this sequence is not enough to discharge the proof obligations of the process. Transition entry assertions and process properties can reference calls from the external environment, changes to the values of imported variables, and call/start/end times of imported transitions. These events can occur at any time with respect to the sequence of transitions in a particular process. Thus, the semantic representation of ASTRAL needs to handle multiple concurrent events as well as gaps in time in which no events occur, which requires a noninterleaved model of concurrency.

The semantics of ASTRAL are based on the predicates *Called* and *Fired*, shown below.

```
Called:   [[{tr1: transition | Base_Trans(tr1) = tr1 AND Exported(tr1)}, time] → bool]
Fired:    [[transition, time] → bool]
```

Called(etr1, t1) is true if and only if an exported base transition etr1 was called from the external environment at time t1. Fired(tr1, t1) is true if and only if tr1 fired at t1. Since a different transition may be executing on each process instance, each process instance has a separate Fired and Called predicate. In ASTRAL, a given process instance "knows" its own execution history completely, but only knows the portion of the execution history of other process instances that pertains to the exported transitions of those instances. In the semantics, for a given process instance, the Fired and Called predicates of the process can be used to derive information about the state variables of the process and vice-versa. The predicates of other process instances, however, can only be used to derive a limited amount of information about those processes. Namely, if an imported transition ended, then it is known there was a corresponding start and similarly, if an imported transition started, then it was called.

Since ASTRAL is based on noninterleaved concurrency, the intra-level proof obligations [CKM 94] (i.e. the proof obligations necessary to show that the invariant and schedule of a level hold) are inductive on ASTRAL's time domain. Since the time domain of ASTRAL is the nonnegative real numbers, however, and simple induction on that domain is not valid, the induction must be performed on nonempty intervals of the nonnegative reals. That is, the induction hypothesis is assumed up to some arbitrary time T0 and the user must show that it holds for a constant length of time $\Delta > 0$ afterwards. The induction case of the invariant proof obligation is shown below.

```
invariant_induct: THEOREM
   (FORALL (T1: time): T1 ≤ T0 IMPLIES Invariant(T1)) IMPLIES
      (FORALL (T1: time): T0 < T1 AND T1 < T0 + Δ IMPLIES Invariant(T1))
```

For the induction to be reasonable, Δ must be bounded because the bigger Δ becomes, the more difficult it is to prove that the property holds at the times close to the upper bound T0 + Δ. This is because at those times, more and more time has elapsed since the last known state of the system (i.e. when the inductive hypothesis held). In translating the proof obligations into PVS, it was not possible to say that Δ is "as small as possible". Instead, an explicit upper bound needed to be chosen to restrict Δ. The upper bound chosen for the ASTRAL encoding was a value less than the smallest transition duration. That is, the conjunct "(FORALL (tr1: transition): Δ < Duration(tr1))" was added to the proof obligation above.

This bound is satisfactory for a number of reasons. The main justification is that with Δ bounded by the smallest duration, only a single transition can fire or complete execution within the proof interval. This is advantageous because if only a single transition can end, then the state variables can only change once within the interval. Additionally, if a transition did end within the interval, then the inductive hypothesis held when the transition began firing. These qualities are useful for automating the proofs of certain types of properties as will be shown in section 10.7.1.1.

### 6.3.4. Irregular Operators

In some specification languages, there are operators whose type signatures cannot be described in a regular fashion. One example is the ASTRAL Start operator. For unparameterized transitions, the signature of the Start operator is regular and can be written as "[transition, time] → boolean". For parameterized transitions, however, the transition operand can also be a transition name with a parameter list. For a transition tr1 with n parameters of arbitrary type ($p_1$, ..., $p_n$), all of the following are legal ASTRAL expressions: Start(tr1, t1), Start(tr1($p_1$), t1), Start(tr1($p_1$, $p_2$), t1), ..., Start(tr1($p_1$, ..., $p_n$), t1). Since the parameters are of arbitrary and possibly differing types, there is no type signature that can adequately describe the Start operator. In order to encode the parameterized version of the Start operator, it was necessary to "regularize" its definition.

This was done by first introducing a new "parameter type" using a record declaration, which contains the parameter names and types of all transitions in the process. For example, the definition of parameter in the Central_Control process of the phone system, which is described in section 2.1.5, is shown below

```
parameter: TYPE = [#
    p_area_phone__1: area_phone, p_connection__1: connection,
    p_connection__connection_status__1: connection,
    p_connection__connection_status__2: connection_status #]
```

The parameter elements are named based on the type signatures found in the transition definitions. In the above definition, there are three type signatures, [area_phone], [connection], and [connection, connection_status]. The trailing number indicates the position of the element of the given type in its signature. The idea of this scheme is that all entry/exit assertions and transition operator definitions can reference the same type (i.e. parameter) and use only those parts of a parameter instance appropriate in the given situation. The parts of a parameter that are not used in an expression for all intents and purposes do not exist for that expression. For example, an entry assertion may reference parameters that are passed to it when called from the external environment. The entry assertion only references its own declarations within the parameter type, thus only constrains those portions of the parameter. The unreferenced elements of the parameter type can have any value, thus they do not affect the reasoning.

In addition to the parameter definition, it was necessary to provide a semantic foundation for parameterized transitions. In ASTRAL, any transition may have parameters that are used in the entry and exit assertions to describe the conditions of enablement and the effects of execution, respectively. Exported transitions are enabled if there is some set of parameters that has been provided by the external environment at a time when the transition was called that satisfies the entry assertion and has not yet been serviced by a previous execution of the transition. Transitions that are not exported are enabled if there is any set of parameters of the appropriate types that satisfy the entry assertion. When a parameterized transition fires, one set of the possible sets of parameters is chosen nondeterministically. In the semantics, the functions *Call_Parms* and *Fire_Parms*, shown below, are defined to record the history of transition parameters.

Call_Parms: [[etr1: {tr1: transition | Base_Trans(tr1) = tr1 AND Exported(tr1) AND Has_Parms(tr1)}, {t1: time | Called(etr1, t1)}] → set[parameter]]
Fire_Parms: [[btr1: {tr1: transition | Base_Trans(tr1) = tr1 AND Has_Parms(tr1)}, {t1: time | (EXISTS (tr1: transition): Base_Trans(tr1) = btr1 AND Fired(tr1, t1))}] → parameter]

Call_Parms is only valid at times when an exported transition has been called and holds the parameters supplied by the external environment. Fire_Parms is only valid at times when a parameterized transition has fired and holds the instance of the parameters for which the transition fired.

Start(tr1($p_1$, ..., $p_i$), t1) is true if and only if the last time tr1 fired with its first i parameters equal to $p_1$, ..., $p_i$ was at time t1. The last component necessary to regularize the definition of the Start operator was a function to determine the equality of the first i parameters of a given transition in two instances of the parameter type. For each process specification, an *Eval_Parms* function is

constructed with the required functionality. The Eval_Parms function of the Elevator_Button_Panel process of the elevator control system, which is described in section 2.1.3, is shown below. Eval_Parms is defined recursively on the number of parameters to check. Depending on the transition given, a different set of components of the parameter record are checked. In the definition below, the "p_floor__1" component is checked in the request_floor transition. The *measure* at the end of the Eval_Parms definition must be given in every PVS recursive function definition. It has the same signature as the associated function and defines an expression that decreases in each recursive iteration. It is used to prove the termination of the function.

```
Eval_Parms(BTR1: {TR1: transition | Base_Trans(TR1) = TR1 AND
        Has_Parms(TR1)}, N1: nat, P1: parameter, P2: parameter):
            RECURSIVE bool =
    (IF N1 = 0 THEN TRUE
    ELSE
        CASES BTR1 OF
            request_floor:
                IF N1 = 1 THEN p_floor__1(P1) = p_floor__1(P2)
                ELSE FALSE
                ENDIF
            ELSE FALSE
        ENDCASES AND
        Eval_Parms(BTR1, N1 - 1, P1, P2)
    ENDIF)
    MEASURE (LAMBDA (BTR1: {TR1: transition | Base_Trans(TR1) = TR1}, N1: nat,
        P1: parameter, P2: parameter): N1)
```

With the above definitions, it is possible to provide a regular definition of the Start operator. The Start1 definition shown below is similar to the Start1 definition in section 6.3.1 except that it takes a natural number n1 and a Curried parameter ap1. This definition requires that one of the exceptions associated with btr1 has fired and that the first n1 parameters of btr1 in ap1(t1) match the first n1 parameters of the actual fire parameters at that time. In addition, any time after the given time (at1(t1)) at which an exception associated with btr1 fired, the first n1 parameters must not match.

```
Start1(btr1: {tr1: transition | Base_Trans(tr1) = tr1 AND Has_Parms(tr1)}, n1: nat,
        ap1: [time → parameter], at1: [time → time])(t1: {t1: time | at1(t1) ≤ t1}): bool =
    (EXISTS (tr1: transition):
        Base_Trans(tr1) = btr1 AND
        Fired(tr1, at1(t1)) AND
        Eval_Parms(btr1, n1, ap1(t1), Fire_Parms(btr1, at1(t1)))) AND
        (FORALL (t2: time):
            at1(t1) < t2 AND t2 ≤ t1 IMPLIES
                (FORALL (tr1: transition):
                    Base_Trans(tr1) = btr1 AND
                    Fired(tr1, t2) IMPLIES
                        NOT Eval_Parms(btr1, n1, ap1(t1), Fire_Parms(btr1, t2))))
```

## 6.4. ASTRAL Axiomatization

The ASTRAL axiomatization is defined by three types of axioms. The abstract machine axioms describe the execution of a single process. The imported transition axioms describe information that can be derived about the execution of other processes. Finally, the specification-dependent axioms are axioms that can only be constructed after a specification is given.

### 6.4.1. Abstract Machine Axioms

The execution history of a process is represented by the predicates Called and Fired, discussed in section 6.3.3, and the functions Call_Parms and Fire_Parms, discussed in section 6.3.4. There are eight core axioms based on these four functions that describe the ASTRAL abstract machine. The *call_fire_parms* axiom describes the relationship between Call_Parms and Fire_Parms. Namely, if an exported parameterized transition tr1 fires at t1, the parameters for which tr1 fired must come from the set of tr1 call parameters that have not yet been serviced at t1.

```
call_fire_parms: AXIOM
    (FORALL (tr1: transition, t3: time):
        Base_Trans(tr1) = tr1 AND
        Exported(tr1) AND
        Has_Parms(tr1) AND
        (EXISTS (tr2: transition):
            Base_Trans(tr2) = tr1 AND
            Fired(tr2, t3)) IMPLIES
            (EXISTS (t1: time):
                t1 ≤ t3 AND
                Called(tr1, t1) AND
                member(Fire_Parms(tr1, t3), Call_Parms(tr1, t1)) AND
                (FORALL (tr2: transition, t2: time):
                    t1 ≤ t2 AND t2 < t3 AND
                    Base_Trans(tr2) = tr1 AND
                    Fired(tr2, t2) IMPLIES
                        Fire_Parms(tr1, t2) ≠ Fire_Parms(tr1, t3))))
```

The *trans_fire* axiom is the only way to directly derive that a transition fired. It states that if some transition is enabled and the process is idle (i.e. no other transition is in the middle of execution), then some transition will fire. Note that Enabled requires that the transition's entry assertion holds and that if the transition is exported, then it has been called as shown in section 6.5.3.1.

```
trans_fire: AXIOM
    (FORALL (t1: time):
        (EXISTS (tr1: transition): Enabled(tr1, t1)) AND
        (FORALL (tr2: transition, t2: time):
            t1 - Duration(tr2) < t2 AND t2 < t1 IMPLIES
                NOT Fired(tr2, t2)) IMPLIES
            (EXISTS (tr1: transition): Fired(tr1, t1)))
```

The trans_fire axiom by itself is not sufficient to describe what occurs when a transition fires. A number of other axioms make assertions that further describe the behavior of a process. The *trans_entry* axiom states that whenever a transition fires, its entry assertion held at that time.

```
trans_entry: AXIOM
    (FORALL (tr1: transition, t1: time):
        Fired(tr1, t1) IMPLIES
            Entry(tr1, t1))
```

The *trans_exit* axiom states that whenever a transition fires, its exit assertion holds at a time duration later. Note that in this case, the user must guarantee that the exit assertion will not evaluate to false for the axiom to be sound. In the case of trans_entry, this requirement is not necessary because it is not possible to derive Fired(tr1, t1) if Entry(tr1, t1) does not hold. In the trans_exit case, however, it is possible to derive Fired(tr1, t1), regardless of the value of Exit(tr1, t1 + Duration(tr1)).

```
trans_exit: AXIOM
    (FORALL (tr1: transition, t1: time):
        t1 ≥ Duration(tr1) AND
        Fired(tr1, t1 - Duration(tr1)) IMPLIES
            Exit(tr1, t1))
```

The *trans_called* axiom states that whenever an exported transition fires, it must have been called since the last time the transition fired. Note that it was not possible to deduce this in the axioms of [CKM 94] and [CSK 94].

```
trans_called: AXIOM
    (FORALL (tr1: transition, t1: time):
        Fired(tr1, t1) AND
        Exported(Base_Trans(tr1)) IMPLIES
            Issued_Call(Base_Trans(tr1), t1))
```

The *trans_mutex* axiom states that whenever a transition fires, no other transition can fire until duration later (i.e. until the transition ends). This axiom combined with trans_fire is sufficient to show that a single unique transition fires when some transition is enabled and the process is idle.

```
trans_mutex: AXIOM
    (FORALL (tr1: transition, t1: time):
        Fired(tr1, t1) IMPLIES
            (FORALL (tr2: transition):
                tr2 ≠ tr1 IMPLIES
                    NOT Fired(tr2, t1)) AND
            (FORALL (tr2: transition, t2: time):
                t1 < t2 AND t2 < t1 + Duration(tr1) IMPLIES
                    NOT Fired(tr2, t2)))
```

These six axioms describe the dynamic execution of transitions. Besides the start, end, and call times of transitions, the other time-dependent entities are variables. The axioms so far only describe variables implicitly in the Entry, Exit, and Enabled functions used in them. Thus, the value of a

variable is only known at the time a transition starts and when it ends. In ASTRAL, however, it is also known that a variable only changes value when a transition ends. Thus, the *vars_no_change* axiom states this fact. Specifically, it states that for any interval in which a transition has not ended, all variables keep a single value throughout the interval. This axiom was missing from [CKM 94]. The Vars_No_Change function is process-dependent and is constructed by the translator based on the variables declared in each process. Vars_No_Change(t1, t2) states that the value of all variables of the process have the same value at t1 as they do at t2.

```
vars_no_change: AXIOM
    (FORALL (t1: time, t3: time):
        t1 ≤ t3 AND
        (FORALL (tr2: transition, t2: time):
            t1 < t2 + Duration(tr2) AND
            t2 + Duration(tr2) ≤ t3 IMPLIES
                NOT Fired(tr2, t2)) IMPLIES
            (FORALL (t2: time):
                t1 ≤ t2 AND t2 ≤ t3 IMPLIES
                    Vars_No_Change(t1, t2)))
```

Finally, the *initial_state* axiom states that the initial condition holds at time zero. In [CKM 94], this did not appear as an axiom, but instead appeared in the base case proofs. That is, the initial condition appears in the proof obligations as "initial & now = 0 → invariant (or schedule)". When the initial condition appears like this, however, nothing can be inferred about the initial state of the system. Thus, if the system depends on the initial configuration, nothing can be proved about its operation. As was the case in trans_exit with Exit, Initial is required to be true at time zero, or else the soundness of the axiom cannot be guaranteed.

```
initial_state: AXIOM
    Initial(0)
```

### 6.4.2. Imported Transition Axioms

In addition to the abstract machine axioms, there are three axioms dealing with imported transitions. Most of the information that can be derived about local transitions cannot be derived about imported variables and transitions. For example, it is not known when imported variables will change, nor what the duration of an imported transition is, nor what held when an imported transition started or ended, etc. If any of these items are required to hold to prove a schedule, they must be explicitly stated in an imported variable clause. There are, however, a few things that can be deduced about all imported transitions, regardless of context. None of these axioms were dealt with in [CKM 94] or [CSK 94].

The imported axioms are expressed in terms of *i_Called*, *i_Started*, and *i_Ended* and their associated parameter functions *i_Call_Parms*, *i_Start_Parms*, and *i_End_Parms*, which are defined as shown below. These functions correspond to the local definitions of Called, Fired, Call_Parms, and Fire_Parms, but refer to information about transitions imported from other processes. The exact duration between a start and an end of an imported transition is not known globally or in other processes because the duration is implementation dependent. In addition, the duration may have different values depending on the number and durations of exceptions of each transition. Thus, i_Started and i_Ended had to be defined separately, rather than the single Fired of local process definitions.

```
i_Started: [[id, i_transition, time] → bool]
i_Ended: [[id, i_transition, time] → bool]
i_Called: [[id, i_transition, time] → bool]

i_Start_Parms: [[id1: id, itr1: i_transition,
    {t1: time | Started(id1, itr1, t1)}] → i_parameter]
i_End_Parms: [[id1: id, itr1: i_transition,
    {t1: time | Ended(id1, itr1, t1)}] → i_parameter]
i_Call_Parms: [[id1: id, itr1: i_transition,
    {t1: time | Called(id1, itr1, t1)}] → set[i_parameter]]
```

The *i_trans_mutex* axiom states that for any process id and in any interval such that an imported transition started at the beginning of the interval and has not yet ended, no imported transition can have started or ended on the process associated with that process id within the interval (excluding the first instant).

```
i_trans_mutex: AXIOM
    (FORALL (id1: id, itr1: i_transition, t1: time, t3: time):
        t1 < t3 AND
        i_Started(id1, itr1, t1) AND
        (FORALL (t2: time):
            t1 < t2 AND t2 ≤ t3 IMPLIES
                NOT i_Ended(id1, itr1, t2)) IMPLIES
        (FORALL (itr2: i_transition, t2: time):
            t1 < t2 AND t2 ≤ t3 IMPLIES
                NOT i_Started(id1, itr2, t2) AND
                NOT i_Ended(id1, itr2, t2)))
```

The *i_trans_end* axiom states that for any process id, if an imported transition has ended on that process, no other imported transition ended on the same process at the same time and there was a start with the same parameters that has occurred since the last time the transition ended.

```
i_trans_end: AXIOM
    (FORALL (id1: id, itr1: i_transition, t3: time):
        i_Ended(id1, itr1, t3) IMPLIES
            (FORALL (itr2: i_transition):
                itr2 ≠ itr1 IMPLIES
                    NOT i_Ended(id1, itr2, t3)) AND
            (EXISTS (t1: time):
                t1 < t3 AND
                i_Started(id1, itr1, t1) AND
                i_Start_Parms(id1, itr1, t1) = i_End_Parms(id1, itr1, t3) AND
                (FORALL (t2: time):
                    t1 < t2 AND t2 < t3 IMPLIES
                        NOT i_Ended(id1, itr1, t2))))
```

The *i_trans_start* axiom is similar to i_trans_end, except that it states that if an imported transition starts, then no other imported transition started on the same process at the same time and that the transition has been called but not yet serviced with the parameters used at the start.

```
i_trans_start: AXIOM
    (FORALL (id1: id, itr1: i_transition, t3: time):
        i_Started(id1, itr1, t3) IMPLIES
            (FORALL (itr2: i_transition):
                itr2 ≠ itr1 IMPLIES
                    NOT i_Started(id1, itr2, t3)) AND
            (EXISTS (t1: time):
                t1 ≤ t3 AND
                i_Called(id1, itr1, t1) AND
                member(i_Start_Parms(id1, itr1, t3), i_Call_Parms(id1, itr1, t1)) AND
                (FORALL (t2: time):
                    t1 ≤ t2 AND t2 < t3 AND
                    i_Started(id1, itr1, t2) IMPLIES
                        i_Start_Parms(id1, itr1, t2) ≠ i_Start_Parms(id1, itr1, t3))))
```

### 6.4.3. Specification-Dependent Axioms

There are some axioms that are specification-dependent and must be constructed during translation. In the local and global cases, the axiom section from the specification is translated as an axiom. The axiom section is a time-independent clause, but rather than implementing a separate translation procedure for it, the standard formula translation is used and then the formula is evaluated at time 0. If the axiom clause is written correctly and is time-independent, the 0 will drop out and only assertions about constants will remain.

There are three additional axioms constructed in the global specification. The type "id" is declared as a NONEMPTY_TYPE, thus nothing is known about it by PVS, except that id has at least one item in its domain. The *id_domain* axiom further refines this domain by stating that every id must correspond to some process instance declared in the processes section. For example, the id_domain

axiom of the elevator control system, shown below, states that every id is either the_elevator, the_elevator_buttons, or one of the set of n_floors the_floor_buttons.

```
id_domain: AXIOM
    (FORALL (PID1: id):
        PID1 = the_elevator OR
        PID1 = the_elevator_buttons OR
        (EXISTS (I1: {K1: int | K1 ≥ 1 AND K1 ≤ n_floors}):
            PID1 = the_floor_buttons(I1)))
```

The *id_unique* axiom states that every process instance corresponds to a unique id. Thus, the type id is defined to be exactly the set of process instances. The id_unique axiom of the elevator control system, shown below, states that the_elevator, the_elevator_buttons, and the set of n_floors the_floor_buttons are all different.

```
id_unique: AXIOM
    the_elevator ≠ the_elevator_buttons AND
    (FORALL (I1: {K1: int | K1 ≥ 1 AND K1 ≤ n_floors}):
        the_elevator ≠ the_floor_buttons(I1)) AND
    (FORALL (I1: {K1: int | K1 ≥ 1 AND K1 ≤ n_floors}):
        the_elevator_buttons ≠ the_floor_buttons(I1)) AND
    (FORALL (I1, J1: {K1: int | K1 ≥ 1 AND K1 ≤ n_floors}):
        the_floor_buttons(I1) = the_floor_buttons(J1) IMPLIES
            I1 = J1)
```

The *i_initial_state* axiom asserts that the exported portion of the initial state of each process holds at time zero. In order to construct this axiom, the initial clause of each process is split using the formula splitter and each split that does not reference any local variables or constants is conjoined to an expression for that process. The i_initial_state axiom of the elevator control system is shown below. Note that the expression constructed for each process type is quantified to cover all instances declared of that type.

```
i_initial_state: AXIOM
    (FORALL (PID1: id):
        Id_Type(PID1) = const(elevator) IMPLIES
            i_elevator__position(PID1)(0) = 1 AND
            i_elevator__going_up(PID1)(0) AND
            NOT i_elevator__door_open(PID1)(0) AND
            NOT i_elevator__moving(PID1)(0) AND
            NOT i_elevator__door_moving(PID1)(0)) AND
    (FORALL (PID1: id):
        Id_Type(PID1) = elevator_button_panel IMPLIES
            (FORALL (f: floor):
                NOT i_elevator_button_panel__floor_requested(PID1)(f)(0)) AND
    (FORALL (PID1: id):
        Id_Type(PID1) = floor_button_panel IMPLIES
            NOT i_floor_button_panel__up_requested(PID1)(0) AND
            NOT i_floor_button_panel__down_requested(PID1)(0))
```

There is also an additional axiom in each process definition. The *self_imports* axiom states the relationship between exported, but locally named variables and their global counterparts of the form i_var(id). In addition to stating that var = i_var(self), self_imports also relates the local values of Called, Fired, Call_Parms, and Fire_Parms for exported transitions to the global definitions of i_Called, i_Started, i_Ended, and their associated parameter functions. The self_imports axiom for the Elevator_Button_Panel process is given below. Note that this axiom is large for even this simple process and can get extremely large for complex processes.

```
self_imports: AXIOM
    i_elevator_button_panel__floor_requested(self) = floor_requested AND
    (FORALL (T1: time):
        (i_Called(self, i_elevator_button_panel__request_floor, T1) IFF
            Called(request_floor, T1)) AND
        (i_Called(self, i_elevator_button_panel__request_floor, T1) IMPLIES (
            (FORALL (P1: parameter):
                member(P1, Call_Parms(request_floor, T1)) IMPLIES
                    (EXISTS (IP1: i_parameter):
                        member(IP1, i_Call_Parms(self,
                            i_elevator_button_panel__request_floor, T1)) AND
                        p_floor__1(P1) = p_floor__1(IP1))) AND
            (FORALL (IP1: i_parameter):
                member(IP1, i_Call_Parms(self,
                    i_elevator_button_panel__request_floor, T1)) IMPLIES
                    (EXISTS (P1: parameter):
                        member(P1, Call_Parms(request_floor, T1)) AND
                        p_floor__1(P1) = p_floor__1(IP1))))) AND
        (i_Started(self, i_elevator_button_panel__request_floor, T1) IFF
            (EXISTS (TR1: transition):
                Base_Trans(TR1) = request_floor AND
                Fired(TR1, T1))) AND
        (i_Started(self, i_elevator_button_panel__request_floor, T1) IMPLIES (
            p_floor__1(Fire_Parms(request_floor, T1)) =
                p_floor__1(i_Start_Parms(self,
                    i_elevator_button_panel__request_floor, T1)))) AND
        (i_Ended(self, i_elevator_button_panel__request_floor, T1) IFF
            (EXISTS (TR1: transition):
                T1 ≥ Duration(TR1) AND
                Base_Trans(TR1) = request_floor AND
                Fired(TR1, T1 - Duration(TR1))))))
```

## 6.5. ASTRAL-PVS Library and Translator

The axiomatization and operator definitions discussed in sections 6.3 and 6.4 have been incorporated into an ASTRAL-PVS library. This library contains the specification-independent core of the ASTRAL language. In the axiomatization and operator definitions, some of the theories are parameterized by type and function constants. For example, to define the trans_fire axiom, the type

"transition" and the function "Duration" need to be supplied to the axiomatization. In order to use the axiomatization, the appropriate types and functions must be defined based on the specification to be verified. An ASTRAL to PVS translator has been developed to automatically construct all of the appropriate definitions given an ASTRAL specification. The full PVS translation of the bakery algorithm specification is given in appendix F as an example.

### 6.5.1. Additional Timed Operator Forms

Besides the definitions of the Start operator presented in sections 6.3.1 and 6.3.4, there are two additional forms of note that the timed operators can take. The first form is for the nth operation in the past. For example, the definition of the nth start in the past is given by the Startn function shown below.

```
Startn_0(i1: posint, btr1: {tr1: transition | Base_Trans(tr1) = tr1}, at1: [time → time],
        t1: time): RECURSIVE bool =
    (IF at1(t1) > t1 THEN FALSE
    ELSIF i1 = 1 THEN Start1(btr1, at1)(t1)
    ELSE (EXISTS (t2: time, t3: time):
                t3 < t2 AND t2 ≤ t1 AND
                Start1(btr1, const(t2))(t1) AND
                Start1(btr1, const(t3))(t3) AND
                (FORALL (t4: time):
                    t3 < t4 AND t4 < t2 IMPLIES
                        NOT Start1(btr1, const(t4))(t4)) AND
                Startn_0(i1 - 1, btr1, at1, t3))
    ENDIF)
    MEASURE (LAMBDA (i1, btr1, at1, t1): i1)

Startn(ai1: [time → posint], btr1: {tr1: transition | Base_Trans(tr1) = tr1},
        at1: [time → time])(t1: {t1: time | at1(t1) ≤ t1}): bool =
    Startn_0(ai1(t1), btr1, const(at1(t1)), t1)
```

The definition of Startn is split into two functions, Startn and its internally invoked counterpart, Startn_0. This separation is so that a TCC is only generated for the Startn reference occurring in the original ASTRAL specification and not for the subsequent recursive references in the internal definition. That is, the user only has to prove that the time argument in the original specification is between zero and the evaluation time so that the result is well defined. In the subsequent recursive calls, however, it may be that the evaluation time (t3) may actually be before the time the user gave (at1(t1)) and yet still be a well defined result. In that case, it would mean that the nth start in the past did not occur at the given time since that time was "passed up" by the recursive calls before the nth start occurred and hence the expression is false and not undefined. Startn(i1, btr1, at1)(t1) holds if

there were two starts in the past at t2 and at t3 such that the last start was at t2 and the second to last start was at t3, and the (i1-1)th start of btr1 occurred at at1(t3).

The other form is for timed operator expressions without the time argument, which return the last time at which the specified operation occurred. For example, the definition of the last time a start occurred is given by the Start1 function shown below.

```
Start1(btr1: {tr1: transition | Base_Trans(tr1) = tr1})(t1: {t1: time |
        (EXISTS (t2: time): t2 ≤ t1 AND Start1(btr1, const(t2))(t1))}): time =
    choose! (t2):
        t2 ≤ t1 AND
        Start1(btr1, const(t2))(t1)
```

This definition illustrates the use of the PVS choose function, which is used to define all of the timed operators without the time argument. Choose is a "choice function", which given a predicate P of type [T → bool], where T is an arbitrary type, returns a element e of type T such that P(e) holds. In the Start1 definition above, it is used to select the time such that the given transition last started.

### 6.5.2. Well-Formed Formula Translations

All well-formed formula clauses of ASTRAL, such as invariants, entry assertions, defines, etc. are translated identically. All of the operators of the ASTRAL language have been encoded as interpreted functions, so given their operands, they evaluate to specific values. The parse tree of the ASTRAL formula is traversed and the appropriate PVS definition is substituted for each ASTRAL operator. The major obstacle in translating well-formed formulas is translating identifiers with types involving lists and structures. In ASTRAL, it is possible to define arbitrary combinations of structures and lists as types, thus references to variables of these types can become quite complex. For example, consider the following type declarations.

```
list1:  list of integer,
struct1:  structure of (l_one: list1)
```

If s1 is a variable of type struct1, valid uses of s1 would include s1 by itself, s1[l_one(5)], and s1[l_one(5)][9]. The translation of expressions such as these must result in a Curried time function, so that it can be used with the definitions of the Curried boolean and arithmetic operators. The expression in each bracket can be time-dependent, so it is necessary to define the translation such that an evaluation context (i.e. time) given to the expression as a whole is propagated to all expressions in brackets.

In the translation of this example, s1 is a function of type [time → struct1] and struct1 is a record [# l_one: [integer → list1] #]. The expression "s1[l_one(5)][9]" becomes

(λ(T1: time): nth(((λ(T1: time): l_one((s1)(T1)) ((const(5))(T1))))(T1), (const(9))(T1)))

The lambdas are added to propagate the temporal context given to the formula as a whole. Although the lambda expression generated for s1 looks very difficult to decipher, translated expressions will never actually be used in this "raw" form. In the proof obligations, a translated expression is always evaluated in some context before being used. Once this evaluation occurs, all the lambdas drop out and the expression is simplified to a combination of variables and predicates. For example, the expression above evaluated at time t becomes

nth(l_one((s2)(t))(5), 9)

First, the value of the variable s1 is evaluated at time t. Then, the record member l_one is obtained from the resulting record. This member is parameterized, so it is given a parameter of 5. Finally, element 9 of the resulting list is obtained.

Well-formed formulas in transition exit clauses require special handling. In ASTRAL, variables that are not referenced or only referenced in "primed form" in exit assertions are assumed to have not changed. If the exit clauses are asserted to hold as they are written, then nothing can be deduced about variables not referenced (i.e. it cannot be shown whether the variables change or do not change value). Thus, *implied nochange* expressions are automatically "added" to the exit clause in the PVS translation. Essentially, for each variable v not mentioned in an exit clause of a transition tr1, the expression "v = v′" must be conjoined to the exit assertion. For example, the exit assertion of the door_stop transition of the Elevator process is generated as:

```
((NOT (door_moving)) AND
((door_open) = (NOT ((λ(T1: time): door_open(T1 - Duration(door_stop))))))) AND
(λ(T1: time): position(T1) = position(T1 - Duration(door_stop))) AND
(λ(T1: time): going_up(T1) = going_up(T1 - Duration(door_stop))) AND
(λ(T1: time): moving(T1) = moving(T1 - Duration(door_stop)))
```

even though the actual exit assertion is:

```
    ~door_moving
&   door_open = ~door_open′
```

Additionally, there are special nochange semantics associated with the IF-THEN-ELSE and ALT operators of ASTRAL. For a full treatment of implied nochanges, see [AK 86a].

### 6.5.3. Process Translations

Each ASTRAL process specification is defined as a set of four PVS theories. One of the theories contains all of the basic definitions of the process such as transitions, types, constants, and variables. Each of the other three theories contain the declarations of the constraint, invariant, and schedule

clauses, respectively, as well as their corresponding proof obligations. The PVS translation of the Proc process of the bakery algorithm specification is given in appendix F.

*6.5.3.1. Transition Translations*

The transitions of a process are defined by a set of six declarations. The "transition" type is an enumerated type that consists of all transition names as well as an identifier "trans__i" for each ith exception of all transitions "trans". For example, the transition declaration for the Central_Control process of the phone system is shown below.

```
transition: TYPE = {
    give_dial_tone, process_digit, process_local_call, process_local_call__1,
    connect_long_distance, connect_long_distance__1, enable_ring,
    disable_ring_pulse, enable_ringback, disable_ringback_pulse, receive_ld,
    start_talk_1, start_talk_2, start_ld, start_ld__1, terminate_ld_1, generate_alarm,
    terminate_local_call, terminate_ld_2, terminate_ld_2__1}
```

In this definition, the transitions that have exceptions are process_local_call, connect_long_distance, start_ld, and terminate_ld_2. The function "Base_Trans(TR1: transition)" is defined to be "trans" for TR1 = trans__i and TR1 otherwise. Each process has a function "Duration" that defines the duration of each transition in the transition type. For each base transition, the function "Exported" returns true or false if that transition is exported or not, respectively. Similarly, "Has_Parms" returns true or false if the given base transition has parameters or not. The function "Num_Parms" returns the number of parameters taken by the given base transition.

The last components of the transition declarations are the entry and exit clauses. These are split into Entry_Parms, Entry_No_Parms, Exit_Parms, Exit_No_Parms, depending on if the transition has parameters or not. The behavior of parameterized transitions differs depending on if the transition is exported or not. For an exported transition with parameters, the parameters are provided by the external environment. Thus, when such a transition is called, parameters are associated with the call and the value of the entry assertion can be determined based on those. A parameterized transition that is not exported, however, is enabled if there exists a set of parameter values that makes the entry assertion evaluate to true. Since the definition of enablement is somewhat complex, it was also incorporated into the ASTRAL-PVS library. This necessitated the split of the entry and exit assertions into "_Parms/_No_Parms" versions so that a standardized definition could be provided. The generic definition of Enabled is shown below.

```
Enabled(tr1: transition, t1: time): bool =
    (Exported(Base_Trans(tr1)) IMPLIES
        Issued_Call(Base_Trans(tr1), t1)) AND
    IF Has_Parms(Base_Trans(tr1)) THEN
        IF Exported(Base_Trans(tr1)) THEN
            (EXISTS (p1: parameter, t3: time):
                t3 ≤ t1 AND
                Called(Base_Trans(tr1), t3) AND
                member(p1, Call_Parms(Base_Trans(tr1), t3)) AND
                (FORALL (tr2: transition, t2: time):
                    t3 ≤ t2 AND t2 < t1 AND
                    Base_Trans(tr2) = Base_Trans(tr1) AND
                    Fired(tr2, t2) IMPLIES
                        Fire_Parms(Base_Trans(tr1), t2) ≠ p) AND
                Entry_Parms(tr1, p1)(t1))
        ELSE
            (EXISTS (p1: parameter):
                Entry_Parms(tr1, p1)(t1))
        ENDIF
    ELSE Entry_No_Parms(tr1)(t1)
    ENDIF
```

*6.5.3.2. Type, Constant, Variable, and Define Translations*

Besides the transition declarations, a process theory also consists of declarations of types, constants, variables, and defines. Since PVS has the ability to declare predicate subtypes as described in section 6.1, the translation of types was very straightforward. For example, the definition of the "floor" type of the elevator control system, which is a typedef, is shown below.

```
floor: TYPE = {i: pos_integer | ((const(i)) ≤ (const(n_floors)))(0)};
```

An evaluation time of "0" at the end of the expression is added because the formula translation mechanism produces a function of type [time → T]. Since type definitions cannot depend on time-dependent entities, the 0 will drop out of all legal type expressions when evaluated. The "const" function was introduced to declare functions constant over time. This is used whenever a constant occurs within an expression, but drops out when the expression is evaluated at a specific time.

Unlike constants, variables have different values at different times and since the history of values that a variable v may take can be referenced explicitly using "past(v, t)", variables are declared as uninterpreted functions from time to their declared domain. Thus, "v(t)" in the encoding holds the value of "past(v, t)" in ASTRAL. A parameterized variable is declared as a function from its parameter domain to a function from type time to the original range. The parameter must always be given in ASTRAL expressions (i.e. functions are not allowed as values), so it was not necessary to

have time as the first operand. For example, the definition of the "floor_requested" variable of the Elevator_Button_Panel process is shown below.

floor_requested: [[floor] → [time → boolean]]

For the most part, these translations were straightforward. One difference between ASTRAL and PVS, however, is that in ASTRAL, there is no ordering implied in the declarations of the specification. That is, it is not necessary (in fact, not possible by the structure of ASTRAL specifications) to declare constants, variables, etc. before they are used. Thus, declarations in the type section may refer to declarations in the constant and definition sections and possibly vice-versa, without producing a typecheck error. In PVS, however, an item can only be referenced after it has been declared. Thus, in order to translate the declarations in an ASTRAL specification correctly into a corresponding PVS specification, the ordering of declarations needed to be explicitly determined. To handle this, the translator first constructs the dependencies of all type declarations. As the dependency lists are created, any constants, defines, and process instances encountered are added to that type's dependency list and their dependencies constructed. If a time-dependent expression, such as a variable or a past expression, is encountered during the construction of the dependencies, an error is reported, since a type cannot be time-dependent. Eventually, when all dependencies lists have been created, the necessary ordering can be determined by declaring the items without dependencies, removing those dependencies from the remaining items, and repeating until all items have been declared. Any circular dependencies encountered result in an error.

### 6.5.4. Global Translations

In addition to the theories for each ASTRAL process type specification, there are three additional PVS theories generated for the global specification. These theories contain the basic global definitions (the global theory), the global invariant translation and invariant proof obligations (the global_INV theory), and the global schedule translation and schedule proof obligations (the global_SCH theory), respectively. The PVS translation of the global portion of the bakery algorithm specification is given in appendix F. The basic global definitions of the global theory are constructed similarly to those of the process theories. The global theory also defines the process instances in the system as well as all exported variables and transitions that are used in the system. Each process definition may contain references to imported variables. The imported variables must be declared before they are referenced, so it is not possible to import a variable from an instantiated process theory because that process may in turn import a variable or transition from the importing process, resulting in a circular dependency between processes. Thus, rather than declaring them separately in

each process that imports them, all exported variables are declared once at the global level for each process instance. In the process type theories, a tradeoff between readability and usability had to be considered. The formulas are much more readable and intuitive using the variable names declared in the process type specification. It is more difficult to use the local proofs as lemmas in the global proofs, however, because the global theory can only reference the global exported variables declared globally. The choices to solve this problem were to exclusively use "i_variable(self)" instead of "variable" in the local formulas and not declare the variable locally in the corresponding process theory, or to declare and use "variable" at the local level and axiomatize the relationship between "variable" and "i_variable(self)". In the encoding, the second method was chosen using the self_imports axiom described in section 6.4.3.

The global obligations look very similar to the local obligations but are different in nature. In the global obligations, it is not possible to reason about properties in quite the same way as in the local case, because the global proofs cannot use any information that is not exported by each process. That is, at the global level, nothing is known about the implementation details of processes such as transition entry and exit assertions or the values of local variables. Thus, the axioms of the ASTRAL abstract machine in section 6.4.1 cannot be used in global proofs. Instead, the global proofs must be performed by using the local invariants and schedules as lemmas to prove properties of the system as a whole. Local proof obligations, however, contain references to local variables that are not visible at the global level. Thus, instead of using the translations of the process invariants and schedules directly, a separate i_Invariant and i_Schedule clause is generated that contains only the exported portions of the corresponding clause from each process. The exported portions are determined in a similar manner as the exported portions of the initial clause discussed in section 6.4.3.

# Chapter 7

# Parallel Refinement Mechanisms

Whether in programming languages or formal specification languages, *refinement* is the process of moving from an abstract design level to a concrete implementation by describing how the components in each upper level are implemented in the lower level. The left side of figure 7-1 shows the process of refinement, where each abstraction layer is depicted as a box. Each lower level box describes the implementation of the box above. Eventually, the complete system description is reached in the right side of the figure. Refinement allows designers to describe a system from the top down in more and more detail. That is, the desired behavior of each individual component is assumed and then the interactions between the components are specified. This allows designers to look at the components that make up the system and their interactions without looking at how each component is implemented. In this way, the design of a system can be modularized into different layers of abstraction.

In program refinement, an abstract program written in the upper level may have several different implementations in the lower level that preserve the properties of the upper level program. Three of the most commonly occurring program refinement techniques are examined in the context of the Unity language in [Sin 93]. In *data refinement*, an abstract data type defined in the upper level program is implemented by a concrete data type in the lower level to match the data types available on a particular machine. In *atomicity refinement*, a program using a coarse grain of atomicity in the upper level is implemented at the lower level with a finer grain of atomicity to allow more concurrency. Finally, in *guard strengthening*, guards in the upper level program become more restrictive in the lower level to limit the possible executions of the program. Conditions are given in [Sin 93] for which these three refinements can be shown to preserve program properties such as safety and progress as well as the fixed point of the program.

In formal methods, refinement allows the analysis of each abstraction layer to be proved without knowledge of how components in that layer are implemented. Each lower level component is then shown to implement the behavior that was assumed in the upper level. This allows the analysis that

was performed in the upper level to be preserved in the lower level. Proofs of properties in the upper level hold for all lower level implementations that meet the assumed behavior. It also simplifies the analysis of the upper level since the upper level does not need to be specified in as much detail as the lower level, so the state space is smaller in size than would be the case when reasoning about a complete implementation. This means that automated analysis techniques have a greater chance of success in the upper level. In real-time systems, refinement is more difficult than in untimed systems because not only do functional requirements need to be preserved, but also timing requirements. ASTRAL is well-suited for refinement since each process has a well-defined interface of what it relies on and what it guarantees, namely the environmental assumptions and imported variable clauses, and the invariants and schedules. This means that the lower level must preserve the invariants and schedules.

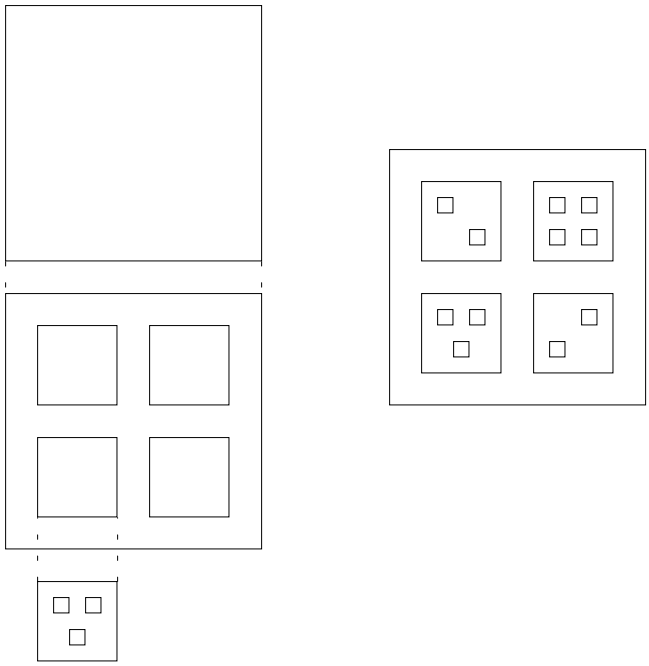


Figure 7-1: Refinement

Refinement has often challenged the formal methods community. In most cases, mathematical elegance and proof manageability have been chosen over flexibility and freedom, which are often needed in practice to deal with unexpected or critical situations. A typical example is provided by algebraic approaches that exploit some notion of homomorphism between algebraic structures. When applied to parallel systems such approaches led to the notion of observational equivalence of processes [HM 85] (i.e. the ability of the lower level process to exhibit all and only the observable

behaviors of the higher level one). Observational equivalence, however, has proved to be too restrictive to deal with general cases and more flexible notions of inter-layer relations have been advocated [DiM 99].

The issue of refinement becomes even more critical when dealing with real-time systems where time analysis is a crucial factor. In this case, the literature contains only a few, fairly limited proposals. [FGM 98] addresses the issue within the context of timed Petri nets and the TRIO language. In this approach, a system is modeled as a timed Petri net and its properties are described as TRIO formulas. Then, mechanisms are given that refine the original net into a more detailed one that preserves the original properties. The approach is limited, however, by the expressive power of pure Petri nets, which do not allow one to deal with functional data dependencies. In [Ost 99], a system is modeled by an extension of finite state machines and its properties are expressed in a real-time logic language. Refinement follows a fairly typical algebraic approach by mapping upper level entities into lower level ones and pursuing observational equivalence between the two layers. In this case, observable variables (i.e. variables that are in the process interface), must be identical in the two levels. This leads to a lack of flexibility, as pointed out above, that is even more evident in time-dependent systems where refined layers must also guarantee consistency between the occurrence times of the events.

In this chapter, general refinement mechanisms are proposed that allow several types of implementation strategies to be specified in a fairly natural way. In particular, processes can be implemented both sequentially, by refining a single complex transition as a sequence or selection of more elementary transitions, and in a parallel way, by mapping one process into several concurrent ones. This allows one to increase the amount of parallelism through refinement whenever needed or desired.

Also, *asynchronous implementation policies* are allowed in which lower level actions can have durations unrelated to upper level ones, provided that their effects are made visible at the lower level exactly at the times specified by the upper level. For instance, in a phone system, many calls must be served simultaneously, possibly by exploiting concurrent service by many processors. Such services, however, are asynchronous since calls occur in an unpredictable fashion at any instant. Therefore, it is not easy to provide a high level description of a call service, which manages a set of calls within a given time interval, in an abstract way that can be naturally refined as a collection of many independent and individual services of single calls, possibly even allowing a dynamic allocation of servers to the phones issuing the calls.

Not surprisingly, generality has a price in terms of complexity. In the approach presented in this chapter, however, this price is paid only when necessary. Simple implementation policies yield simple ASTRAL specifications, whereas complex ASTRAL specifications are needed only for sophisticated implementation policies. The same holds for the proof system, which is built hand-in-hand with the implementation mechanisms.

## 7.1. Sequential Refinement Mechanism

A refinement mechanism for ASTRAL was defined in [CKM 95]. In this definition, an ASTRAL process specification consists of a sequence of levels where the behavior of each level is implemented by the next lower level in the sequence. Given two ASTRAL process level specifications $P_U$ and $P_L$, where $P_L$ is a refinement of $P_U$, the implementation statement IMPL defines a mapping from all the types, constants, variables, and transitions of $P_U$ into their corresponding terms in $P_L$, which are referred to as *mapped* types, constants, variables, or transitions. $P_L$ can also introduce types, constants and/or variables that are not mapped. These are referred to as the *new* types, constants, or variables of $P_L$. Note that $P_L$ cannot introduce any new transitions (i.e. each transition of $P_L$ must be a mapped transition). A transition of $P_U$ can be mapped into a sequence of transitions, a selection of transitions, or any combinations thereof.

A selection mapping of the form $T_U == A_1 \ \& \ T_{L.1} \mid A_2 \ \& \ T_{L.2} \mid ... \mid A_n \ \& \ T_{L.n}$, is defined such that when the upper level transition $T_U$ fires, one and only one lower level transition $T_{L.j}$ fires, where $T_{L.j}$ can only fire when both its entry assertion and its associated "guard" $A_j$ are true. The left side of figure 7.1 depicts a selection of transitions.

A sequence mapping of the form $T_U$ == WHEN $Entry_L$ DO $T_{L.1}$ BEFORE $T_{L.2}$ BEFORE ... BEFORE $T_{L.n}$ OD, defines a mapping such that the sequence of transitions $T_{L.1}$; ...; $T_{L.n}$ is enabled (i.e. can start) whenever $Entry_L$ evaluates to true. Once the sequence has started, it cannot be interrupted until all of its transitions have been executed in order. The starting time of the upper level transition $T_U$ corresponds to the starting time of the sequence (which is not necessarily equal to the starting time of $T_{L.1}$ because of a possible delay between the time when the sequence starts and the time when $T_{L.1}$ becomes enabled), while the ending time of $T_U$ corresponds to the ending time of the last transition in the sequence, $T_{L.n}$. Note that the only transition that can modify the value of a mapped variable is the last transition in the sequence. This further constraint is a consequence of the ASTRAL communication model. That is, in the upper level, the new values of the variables affected by $T_U$ are

broadcast when $T_U$ terminates. Thus, mapped variables of $P_L$ can be modified only when the sequence implementing $T_U$ ends. The right side of figure 7.1 depicts a sequence of transitions.

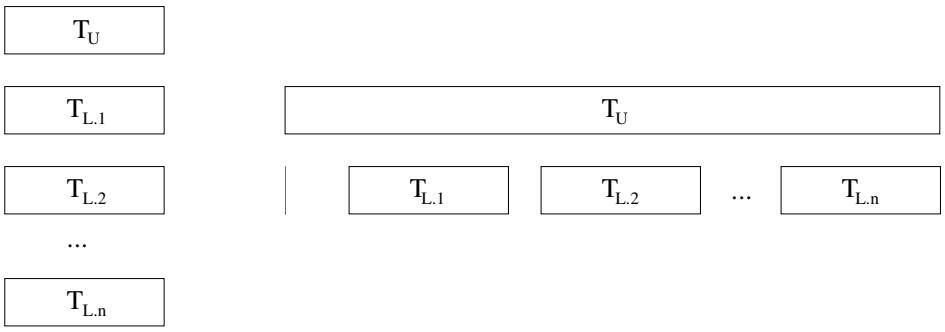



Figure 7.1: Selection and sequence mappings

## 7.2. Proof Obligations for Sequential Refinement Mechanism

The inter-level proofs consist of showing that each upper level transition is correctly implemented by the corresponding sequence, selection, or combination thereof in the next lower level. For selections, it must be shown that whenever the upper level transition $T_U$ fires, one of the lower level transitions $T_{L.j}$ fires, that the effect of each $T_{L.j}$ is equivalent to the effect of $T_U$, and that the duration of each $T_{L.j}$ is equal to the duration of $T_U$. These obligations are, respectively

($S_0$) $\quad IMPL(Entry_U) \leftrightarrow A_1 \,\&\, Entry_{L.1} \mid ... \mid A_n \,\&\, Entry_{L.n}$
($S_{1.j}$) $\quad A_j' \,\&\, Entry_{L.j}' \,\&\, Exit_{L.j} \rightarrow IMPL(Exit_U)$
($S_2$) $\quad Dur_{L.1} = Dur_{L.2} = ... = Dur_{L.n} = Dur_U$

For sequences, it must be shown that the sequence is enabled if and only if $T_U$ is enabled, that the effect of the sequence is equivalent to the effect of $T_U$, and that the duration of the sequence (including any initial delay after $Entry_L$ is true) is equal to the duration of $T_U$. These are shown by the n+2 incremental proof obligations

($P_0$) $\quad IMPL(Entry_U) \leftrightarrow Entry_L$
($P_{j+1}$) $\quad past(Entry_L, Now - Dur_U) \,\&\, Start(T_{L.1}, t_1) \,\&\, ... \,\&\, Start(T_{L.j}, t_j)$
$\rightarrow$ EXISTS $t_{j+1}$: Time
( $t_{j+1} \geq t_j + Dur_{L.j} \,\&\, t_{j+1} + \sum_{k=j+1}^{n} Dur_{L.k} \leq Now$
& $past(Entry_{L.j+1}, t_{j+1})$)
where in obligation $P_n$, "$\leq$ Now" is replaced by "= Now"
($P_{n+1}$) $\quad past(Entry_L, Now - Dur_U) \,\&\, past(Exit_{L.1}, t_1 + Dur_{L.1}) \,\&\, ... \,\&\, Exit_{L.n}$
$\rightarrow IMPL(Exit_U)$

The idea of the selection and sequence obligations is that whenever an upper level transition is enabled, some lower level sequence or selection will be enabled because the entry assertions are equivalent. Similarly, whenever an upper level transition ends, some lower level sequence or

selection will end because the durations are the same. Finally, whenever an upper level transition produces some effect, the lower level transition will produce an equivalent effect because the IMPL of the exit assertion of the upper level transition holds at the end of the lower level sequence or selection. This means that the upper and lower levels will have equivalent executions.

The SDE supports the implementation mechanism of [CKM 95]. Each level below the top level contains an implementation clause that describes the IMPL mapping between that level and the level above it. The validation component of the SDE checks the implementation clause for various errors such as not mapping an item in the upper level, not mapping a transition in the lower level, mapping an item more than once, etc. The VCG component of the SDE uses the information in the implementation clause to construct the above proof obligations. The expressions IMPL(EntryU) and IMPL(ExitU) used in the proof obligations are replaced by the appropriate lower level expressions as defined by the IMPL mapping. The VCG algorithm also includes corrections to two of the problems that have been found in [CKM 95]. The two problems that have been corrected are the proof obligations for arbitrary sequence and selection combinations and the further assumptions algorithm. These problems as well as additional problems are discussed in the next section.

## 7.3. Problems with Sequential Refinement Mechanism

Several problems exist in the work of [CKM 95]. These problems are discussed in the following sections.

### 7.3.1. Arbitrary Sequences and Selections

The exact proof obligations for simple sequences and selections are given in [CKM 95], but the proof obligations that must be generated for an arbitrary combination of sequences and selections are not discussed. While implementing the VCG component of the SDE, these proof obligations were developed. For example, consider the mapping

$$T_U == \text{WHEN } Entry_L \text{ DO } ( \; A_0 \; \& \; ( \; A_1 \; \& \; T_{L.1} \; | \; A_2 \; \& \; (T_{L.2} \text{ BEFORE } T_{L.3})) \; | \; A_4 \; \& \; T_{L.4}) \text{ BEFORE } T_{L.5} \text{ OD.}$$

This mapping consists of nested sequences and selections. For arbitrary combinations of sequences and selections, the proof obligations are generated by constructing a set of simple sequences, such that all possible sequences that can occur in the arbitrary mapping are represented. This is done by "distributing" the selection portions of the mapping until a selection of simple sequences is obtained. Since all the mappings in the set are sequences, the existing sequence proof obligations can then be

generated and proven to show that the behavior of the arbitrary mapping is equivalent to that of the upper level transition. For the above mapping, the set of sequences is

WHEN $Entry_L$ DO $T_{L.1}'$ BEFORE $T_{L.5}$ OD
WHEN $Entry_L$ DO $T_{L.2}'$ BEFORE $T_{L.3}$ BEFORE $T_{L.5}$ OD
WHEN $Entry_L$ DO $T_{L.4}'$ BEFORE $T_{L.5}$ OD,

where $Entry(T_{L.1}') = A_0$ & $A_1$ & $Entry(T_{L.1})$, $Entry(T_{L.2}') = A_0$ & $A_2$ & $Entry(T_{L.2})$, $Entry(T_{L.4}') = A_4$ & $Entry(T_{L.4})$, and $Exit(T_{L.j}') = Exit(T_{L.j})$.

Note that this technique can produce an exponential number of sequences with respect to the number of transitions referenced in the original mapping. This complexity is unavoidable, however, because the user must prove every possible combination of sequences to guarantee the correctness of the mapping. If any combination was not proved, there would be the potential for that combination to violate the critical requirements of the upper level. In order for such complexity to occur, however, a transition mapping must contain a large number of nested selections. In general, the number of nested selections will be small because transitions will rarely need to be implemented by a large number of choices. In the end, the user has the ability to control the number of sequences to be proved by choosing the complexity of the mappings.

### 7.3.2. Soundness of Proof Obligations

The proof obligations presented in [CKM 95] for inter-level refinement are incomplete. First, there is no obligation to prove that the lower level begins execution in a state that is consistent with the initial state of the upper level. This obligation is stated as "$Initial_L \rightarrow IMPL(Initial_U)$". Without this obligation, the other obligations can hold and yet the refinement does not preserve the properties of the upper level. For example, consider the following specification fragments.

```
upper level                 lower level
INITIAL                     IMPL(x) == x
    x = 5                   IMPL(t) == t
INVARIANT                   INITIAL
    x ≥ 0                       x = -2
TRANSITION t                TRANSITION t
    ...                         ...
```

In the initial state, the lower level does not preserve the invariant of the upper level since $x < 0$, but the proof obligations hold, so this allows us to derive $IMPL(Invariant_U) = TRUE$, which is not true, so the proof obligations are unsound.

The obligations for selections are also unsound. Consider an upper level transition $T_U$ with entry assertion $Entry_U$, exit assertion $Exit_U$, and duration $Dur_U$. Suppose $T_U$ is implemented by two transition $T_{L.1}$ and $T_{L.2}$ defined as follows.

| | |
|---|---|
| $A_1 = TRUE$ | $A_2 = FALSE$ |
| $Entry_{L.1} = IMPL(Entry_U)$ | $Entry_{L.2} = TRUE$ |
| $Exit_{L.1} = IMPL(Exit_U)$ | $Exit_{L.2} \rightarrow \neg IMPL(Exit_U)$ |
| $Dur_{L.1} = Dur_U$ | $Dur_{L.2} = Dur_U$ |

This implementation holds by the selection proofs obligations. $S_0$ holds because $IMPL(Entry_U) \leftrightarrow TRUE \ \& \ IMPL(Entry_U) \mid FALSE \ \& \ TRUE$. $S_{1.1}$ holds because $TRUE \ \& \ IMPL(Entry_U') \ \& \ IMPL(Exit_U) \rightarrow IMPL(Exit_U)$. $S_{1.2}$ holds because $FALSE \ \& \ TRUE \ \& \ \neg IMPL(Exit_U) \rightarrow IMPL(Exit_U)$. Finally, $S_2$ holds because $Dur_{L.1} = Dur_{L.2} = Dur_U$. In this implementation, however, $T_{L.2}$ can fire at any time since its entry assertion is true. Its exit assertion does not achieve the effect of $T_U$, however, thus this implementation is not correct even though the proof obligations succeeded.

The obligations for sequences are also unsound. Consider the obligation $P_1$ for sequences as given in [CKM 95].

$$(P_1) \quad past(Entry_L, t_0) \rightarrow EXISTS\ t_1: Time\ (t_1 \geq t_0 \ \& \ t_1 + \sum_{k=j+1}^{n} Dur_{L.k} \leq Now \ \& \ past(Entry_{L.1}, t_1))$$

This obligation states that if $Entry_L$ held at an arbitrary time $t_0$ (i.e. the sequence started at $t_0$), then the entry of the first transition in the sequence (i.e. $T_{L.1}$) must hold such that there is enough time left to complete the sequence. This does not exclude the possibility, however, of another transition being enabled at $t_1$ and nondeterministically firing, thereby delaying or disabling the execution of $T_{L.1}$. Suppose the sequence is a refinement of the upper level transition $T_U$. This means that when $T_U$ fires in the upper level, the sequence for $T_U$ starts to fire in the lower level, but during execution, another sequence or selection begins to fire, corresponding to a transition in the upper level firing while $T_U$ is firing, which is not possible by the trans_mutex axiom.

To correct the obligations, the clause "$past(Entry_{L.n}, t_i)$" is changed to "$past(Start(T_{L.n}, t_i), t_i)$". The change for the $P_1$ obligation is shown below.

$$(P_1') \quad past(Entry_L, t_0) \rightarrow EXISTS\ t_1: Time\ (t_1 \geq t_0 \ \& \ t_1 + \sum_{k=j+1}^{n} Dur_{L.k} \leq Now \ \& \ past(Start(T_{L.1}, t_1), t_1))$$

This forces the first transition of the sequence to occur after the sequence begins execution. The obligations $P_2$ through $P_n$ need to be changed similarly.

### 7.3.3. Further Assumptions and Restrictions Algorithm

In ASTRAL specifications, it is possible to specify implementational restrictions using further assumptions clauses. In these clauses, the domains of constants can be limited using constant refinement clauses and nondeterministic choices between transitions can be restricted using transition selection clauses. To show that a lower level is consistent with an upper level, it is necessary to show that the implementational choices specified in the upper level further assumptions clauses have been implemented correctly in the lower levels. In [CKM 95], the algorithm shown in the left side of figure 7.3.3 is given to construct a new entry assertion $REntry_{Uj}$ for every transition $T_{Uj}$ in a process $P_U$ based on each transition selection rule $R_i$ in the form $\{OpSet_i\}<Condition_i>\{ROpSet_i\}$ of $P_U$.

```
Foreach T_Uj in P_U do
  REntry_Uj := true
  Foreach R_i in TS do
    TMP := true
    if T_Uj ∈ {OpSet_i} ∧ TU_j ∉ {ROpSet_i} then
      Foreach T_Ul in P_U do
        case T_Ul ∈ {OpSet_i} ∧ T_Ul ≠ T_Uj do
          TMP := TMP & Entry_Ul od
        case T_Ul ∉ {OpSet_i} do
          TMP := TMP & ~Entry_Ul od
      od
      TMP := TMP & Condition_i
    fi
    REntry_Uj := REntry_Uj | TMP
  od
  REntry_Uj := Entry_Uj & ~(REntry_Uj)
od
```

```
Foreach T_Uj in P_U do
  REntry_Uj := false
  Foreach R_i in TS do
    if T_Uj ∈ {OpSet_i} ∧ TU_j ∉ {ROpSet_i} then
      TMP := true
      Foreach T_Ul in P_U do
        case T_Ul ∈ {OpSet_i} ∧ T_Ul ≠ T_Uj do
          TMP := TMP & Entry_Ul od
        case T_Ul ∉ {OpSet_i} do
          TMP := TMP & ~Entry_Ul od
      od
      TMP := TMP & Condition_i
    else TMP := false
    fi
    REntry_Uj := REntry_Uj | TMP
  od
  REntry_Uj := Entry_Uj & ~(REntry_Uj)
od
```

Figure 7.3.3: Original and fixed entry assertion construction algorithms

This algorithm builds the new entry assertion for transition $T_{Uj}$ by looking at the rules that contain $T_{Uj}$ in the left hand side but not in the right hand side (i.e. rules that prevent $T_{Uj}$ from firing when its entry assertion evaluates to true). Whenever such a rule is found, a formula representing the conditions that cause the rule to apply is built. Transitions belonging to the left hand side of the rule must be enabled and transitions not belonging to it must be disabled. Furthermore, the boolean expression of the rule has to evaluate to true in order to apply the rule. Each of the resulting formulas defines a state in which transition $T_{Uj}$ must not fire even though it is enabled. Thus, the logical disjunction of all such formulas represent all and only the state in which $T_{Uj}$ must not fire even though $Entry_{Uj}$ evaluates to true. The new entry assertion $REntry_{Uj}$ is obtained as the logical

conjunction of $Entry_{Uj}$ and the negation of the formula representing all and only the states in which $T_{Uj}$ must not fire.

The original algorithm, however, contains an error. Since $REntry_{Uj}$ is initialized to true, the expression "$REntry_{Uj}$ | TMP" will always be true and thus "$Entry_{Uj}$ & ~($REntry_{Uj}$)" will always be false, meaning that no transitions will be enabled. There is also a problem in initializing TMP to true when $T_{Uj}$ is not in any $OpSet_i$ or is in every $ROpSet_i$ when it is in $OpSet_i$. In these cases, TMP will be true, thus "$REntry_{Uj}$ | TMP" will always be true and again "$Entry_{Uj}$ & ~($REntry_{Uj}$)" will always be false. The algorithm can be fixed as shown in the right side of figure 7.3.3.

### 7.3.4. IMPL Mapping

The IMPL mappings for types, constants, and variables are not discussed in [CKM 95], but are assumed to be an extension of the mappings in [AK 85], modified to include ASTRAL constructs. The mappings in [AK 85], however, do not consider any nontrivial type mappings, thus do not allow the IMPL translation of an arbitrary expression to be constructed. For example, consider an upper level with a type "S: set of T" and a variable "v_s: S". In the lower level, the specifier may wish to implement S and v_s as "L: list of T" and "v_l: L", such that if an element of type T is in the set v_s, the element is somewhere on the list v_l. The IMPL mapping can be defined as IMPL(S) == L and IMPL(v_s) == v_l. Suppose an entry assertion $Entry_U$ in a transition of the upper level states that "t ISIN v_s", where t is an element of type T. The proof obligations require IMPL($Entry_U$) be constructed in order to attempt the proofs. There is no mention in [CKM 95] or [AK 85], however, of how to construct the lower level expression for such a type mapping. If only variables are transformed, the entry assertion becomes "IMPL(t) ISIN v_l", but v_l is a list and ISIN is an operator on sets. It is thus necessary to define IMPL mappings in much more detail to be able to attempt the proof obligations. A full discussion of the revised IMPL mapping is presented in section 7.4.3.

### 7.3.5. Expressiveness

Besides errors and omissions in the mechanism itself, the refinement mechanism in [CKM 95] also suffers from a lack of expressiveness and flexibility. That is, for many systems, there are realistic and useful refinements that cannot be expressed using this refinement mechanism. Consider the system shown in figure 7.3.5-1.

This system is a circuit that computes the value of a * b + c * d, given inputs a, b, c, and d. The ASTRAL specification for the circuit is shown below.

```
PROCESS   Mult_Add
   EXPORT
      compute, output
   CONSTANT
      dur1: pos_real
   VARIABLE
      output: integer
   INITIAL
      TRUE
   AXIOM
      TRUE
INVARIANT
   FORALL t1: time, a, b, c, d: integer
      (   Start(compute(a, b, c, d), t1)
   →      FORALL t2: time
             (   t1 + dur1 ≤ t2 & t2 ≤ now
             →      past(output, t2) = a * b + c * d))
TRANSITION   compute(a, b, c, d: integer)
   ENTRY      [TIME:    dur1]
      TRUE
   EXIT
      output = a * b + c * d
```

A reasonable refinement of the Mult_Add circuit is shown in figure 7.3.5-2.

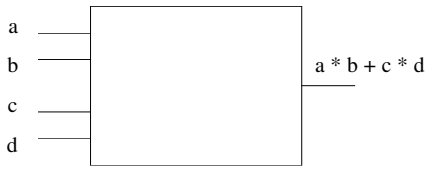



Figure 7.3.5-1: Mult_Add circuit

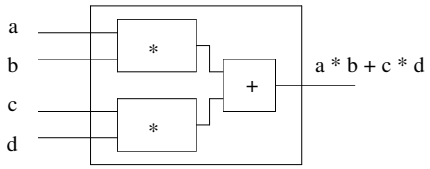



Figure 7.3.5-2: Refined Mult_Add circuit

Here, the refinement of the system consists of two multipliers, which compute a * b and c * d in parallel and then an adder that adds the products together and produces the sum. Although this refinement is a realistic refinement of the system, it cannot be expressed using the [CKM 95] refinement mechanism. This is because there is no notion of parallelism in that mechanism. In the end, the mechanism can only refine a system into a choice between sequences of transitions. Thus, the closest execution that could be expressed is a nondeterministic choice between computing a * b first or c * d first as shown below, which does not capture the essential aspect of the desired refinement. Namely, it does not capture the parallelism between the two multipliers.

```
IMPL(compute(a, b, c, d)) ==
      WHEN TRUE DO  ( multiply(a, b) BEFORE multiply(c, d)
                    | multiply(c, d) BEFORE multiply(a, b))
                       BEFORE add(a * b, c * d) OD
```

The motivation for the development of a parallel refinement mechanism for ASTRAL is to support the expression of any reasonable refinement that a developer may wish to specify, such as the one for the Mult_Add system.

## 7.4. Parallel Refinement Mechanism

In parallel refinement, an upper level transition may be implemented by a dynamic set of lower level transitions. To guarantee that an upper level transition is correctly implemented by the lower level, it is necessary to define what occurs in the lower level when a transition is executed in the upper level, so that it can be shown that it will only occur when an upper level transition fires and that the effect will be equivalent.

### 7.4.1. Parallel Sequences and Selections

The first attempt at defining parallel transition mappings was to extend the sequence and selection mappings into parallel sequence and selection mappings. Thus, a "||" operator could be allowed in transition mappings, such that "P1.tr1 || P2.tr2" indicates that tr1 and tr2 occur in parallel on processes P1 and P2, respectively. With this addition, the compute transition of the Mult_Add circuit could be expressed as the following.

```
IMPL(compute(a, b, c, d)) ==
        WHEN TRUE DO  (   M1.multiply(a, b)
                      ||  M2.multiply(c, d)) BEFORE A1.add(a * b, c * d)
```

where M1 and M2 are the multipliers and A1 is the adder.

Although parallel sequences and selections work well for this example, they do not allow enough flexibility to express many reasonable refinements. For example, consider a production cell that executes a "produce" transition every time unit to indicate the production of an item. In a refinement of this system, the developer may wish to implement produce by defining two "staggered" production cells that each produce an item every two time units, thus effectively producing an item every time unit. The upper level production cell $P_U$ and the lower level production cells $P_{L.1}$ and $P_{L.2}$ are shown in figure 7.4.1. Note that the first transition executed on $P_U$ is an "initialize" transition that is used to represent the "warm-up" time of the production cell in which no output is produced.

This refinement cannot be expressed using parallel sequences and selections because there is no sequence of parallel transitions at the lower level that corresponds directly to produce at the upper level. When produce starts in the upper level, one of the lower level produce's will start and when produce ends in the upper level, one of the lower level produce's will end and achieve the effect of

upper level produce, but the produce that starts is not necessarily the produce that achieves the effect of the corresponding end.

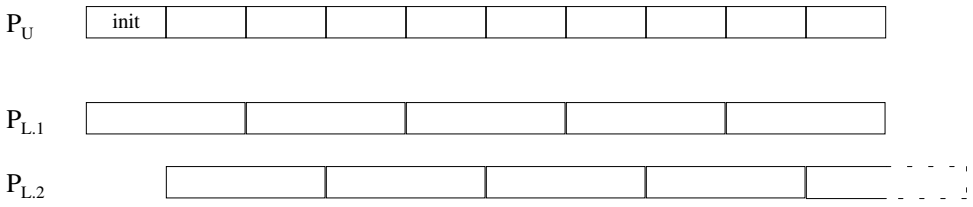



Figure 7.4.1: Production cell refinement

### 7.4.2. Parallel Start and End Mappings

The desired degree of flexibility is obtained by using transition mappings that are based on the start, end, and call of each transition. For each upper level transition $T_U$, a start and end mapping must be defined as follows.

- IMPL(Start($T_U$, now)) == $wff\_start_L$
- IMPL(End($T_U$, now)) == $wff\_end_L$

If $T_U$ is exported, a call mapping must also be defined.

- IMPL(Call($T_U$, now)) == $wff\_call_L$

Here, $wff\_start_L$, $wff\_end_L$, and $wff\_call_L$ are well-formed formulas using lower level transitions and variables. For the most part, the end and call mappings will correspond to the end/call of some transition in the lower level, whereas the start mapping may correspond to the start of some transition or some combination of changes to variables/now/etc. Call mappings are restricted such that for every lower level exported transition $T_L$, Call($T_L$) must be referenced in some upper level exported transition call mapping IMPL(Call($T_U$, now)). This restriction expresses the fact that the interface of the process to the external environment cannot be changed. For parameterized transitions, only the call mapping may reference the parameters given to the transition. Any parameter referenced in a call mapping must be mapped to a call parameter of some lower level transition and the corresponding start mapping must contain the same transitions as the call mapping. For an exported transition, the start and end parameters are the parameters generated at the calls to the transition. For non-exported transitions, the parameters are used to express nondeterminism so cannot be used in the mapping or else it would be necessary to prove that the upper and lower levels always make the same nondeterministic choices in the proof obligations.

With this mapping, the produce transition can be mapped as follows.

```
IMPL(Start(initialize, now)) ==
        now = 0
    &   P_L.1.Start(produce, now)
IMPL(Start(produce, now)) ==
        IF now mod 2 = 0
        THEN P_L.1.Start(produce, now)
        ELSE P_L.2.Start(produce, now)
        FI
```

```
IMPL(End(initialize, now)) ==
        now = 1

IMPL(End(produce, now)) ==
        IF now mod 2 = 0
        THEN P_L.1.End(produce, now)
        ELSE P_L.2.End(produce, now)
        FI
```

It is possible to show that any sequence or selection mapping as defined in [CKM 95] can be described by the start and end mappings. For a selection $T_U$ == $A_1$ & $T_{L.1}$ | $A_2$ & $T_{L.2}$ | ... | $A_n$ & $T_{L.n}$, the start of the upper level transition $T_U$ occurs whenever one of transitions $T_{L.i}$ starts and its associated guard holds. This is described by the start mapping

```
IMPL(Start(T_U, now)) ==
        (   A_1 & P_L.Start(T_L.1, now)
        |   A_2 & P_L.Start(T_L.2, now)
        |   ...
        |   A_n & P_L.Start(T_L.n, now))
```

The end of $T_U$ occurs whenever one of the transitions $T_{L.i}$ ends. This is described by the end mapping

```
IMPL(End(T_U, now)) ==
        (   P_L.End(T_L.1, now)
        |   P_L.End(T_L.2, now)
        |   ...
        |   P_L.End(T_L.n, now))
```

For a sequence $T_U$ == WHEN $Entry_L$ DO $T_{L.1}$ BEFORE $T_{L.2}$ BEFORE ... BEFORE $T_{L.n}$ OD, the start of $T_U$ occurs whenever the entry condition of the sequence $Entry_L$ holds. This is described by the start mapping

```
IMPL(Start(T_U, now)) == Entry_L
```

The end of $T_U$ occurs whenever the last transition $T_{L.n}$ ends. This is described by the end mapping

```
IMPL(End(T_U, now)) == P_L.End(T_L.n, now)
```

Note that nothing is stated about the transitions that occur between when $Entry_L$ holds and $T_{L.n}$ ends. The rest of the sequence will appear in the proof obligations, where it will need to be proven that each transition of the sequence occurs at the proper time to guarantee that the sequence takes $Dur_U$ time and that the effect of $T_U$ is satisfied.

### 7.4.3. Other Mappings

Besides transitions, the user must also define mappings for types, constants, and variables. For the most part, the constant and variable mappings are similar to the mappings used in [CKM 95]. In [CKM 95], however, the lower level only consisted of a single process type, so variable mappings

only referred to the variables of a single process. In the parallel mechanism, variable mappings can refer to variables and transitions of any process in the refinement. For example, the mapping for a variable x can be defined IMPL(x) == (P1.y + P2.z) / 2. That is, x is the average of the y variable of process P1 and the z variable of process P2. An additional consequence of having multiple lower level processes is that information must be provided about the processes that make up the lower level. Thus, the implementation section contains a processes clause similar to the clause in the global specification that describes all the process instances of the new lower level. The upper level process is the parallel composition of these process instances.

As discussed in section 7.3.4, the description of the type mappings in [CKM 95] and [AK 85] are not sufficient to construct the IMPL translation of an arbitrary expression. The example used in section 7.3.4 was a set type being mapped to a list type. This created a problem because the set operators are not valid on lists. In general, there is a problem any time an upper level type T is mapped to a lower level type IMPL(T) that is not "compatible" with T. To be more precise, define types T and IMPL(T) to be compatible if and only if:

- T is an undefined type
- T is identical to IMPL(T)
- T is a list of E and IMPL(T) is a list of IE and E is compatible with IE
- T is a set of E and IMPL(T) is a set of IE and E is compatible with IE
- T is a structure of ID1: E1, ..., IDn: En and IMPL(T) is a structure of ID1: IE1, ..., IDn: IEn and Ei is compatible with IEi
- IMPL(T) is a typedef of E and T is compatible with E

Note that type mappings are restricted such that built-in types cannot be mapped and that any alias or subtype of a given supertype can only be mapped if no other alias or subtype has been mapped. For example, the types T1 and T2: typedef t1: T1 (P(t1)) cannot both be mapped. In this restriction, the built-in types integer and time are assumed to be subtypes of the supertype real.

Examples of compatible types are:

(1) T: (e1, e2), IMPL(T): (e1, e2)
(2) T: list of real, IMPL(T): list of integer

Examples of incompatible types are:

(1) T: (open, closed), IMPL(T): (open, closed, opening, closing)
(2) T: list of integer, IMPL(T): set of integer
(3) T: list of bool, IMPL(T): integer
(4) T: structure of (i1: integer, i2: integer),
IMPL(T): structure of (j1: integer, j2: integer)

A more complete definition of the IMPL mapping is given below. The IMPL mapping describes how items in an upper level are implemented by items in a lower level. The items of the upper level

include variables, constants, types, and transitions. In addition, the IMPL mapping must describe how upper level expressions are transformed into lower level expressions. In many cases, namely when variables and constants are mapped to expressions of compatible types, the basic mappings are sufficient to transform upper level expressions into lower level expressions. When mappings occur between incompatible types, however, the basic mappings must be supplemented with additional mapping information.

For each upper level type T that is mapped to an incompatible lower level type IMPL(T) and for each variable or constant of type T that is mapped to a lower level expression of an incompatible type TL, a mapping must be defined for each operator op in the upper level that is used on an item of type T. For simplicity, assume that all operators are in prefix notation.

IMPL(op(v1: T1, ..., vi: T, ..., vn: Tn)) == f(IMPL(v1), ..., IMPL(vi), ..., IMPL(vn))

The operator mappings are restricted such that none of the timed operators (i.e. start, end, call, change, and past) can be mapped. The start, end, and call operators will always be mapped as a simple replacement mapping as described earlier. The past and change operators will always use the "natural" operator mapping. The natural mapping is defined as follows.

IMPL_0(op(v1: T1, ..., vn: Tn)) == op(IMPL(v1), ..., IMPL(vn))

In other words, the natural mapping for operators passes the IMPL construct through to its operands. For example, IMPL(past(A, t)) == past(IMPL(A), IMPL(t)) and IMPL(change(A, t)) = change(IMPL(A), IMPL(t)). The implementation of any operator that does not have an explicit mapping for its operand types is defined to be the natural operator mapping.

As an example of an operator mapping, consider the mapping from type "S: set of T" to "L: list of T", where the element type T of S and L is integer. Suppose an expression "{1, 2, 3} SUBSET v_s" occurs in the upper level. S is not compatible with L, so the SUBSET operator must be mapped.

```
IMPL(SUBSET(s1: S, s2: S)) ==
        list_len(IMPL(s1)) ≠ list_len(IMPL(s2))
    &   FORALL i: integer
            (   1 ≤ i
            &   i ≤ list_len(IMPL(s1))
        →       EXISTS j: integer
                    (   1 ≤ j
                    &   j ≤ list_len(IMPL(s2))
                    &   IMPL(s2)[j] = IMPL(s1)[i]))
```

In this case, the implementation of subset is defined such that whenever s1 is a proper subset of s2 in the upper level, the lists corresponding to s1 and s2 in the lower level do not have the same length and every element that is on the list IMPL(s1) is on the list IMPL(s2).

There are several things to note about this mapping. First, IMPL is allowed to be recursive on the structure of the parse tree. That is, for an operator op(p1, ..., pn), IMPL(op(...)) is allowed to reference IMPL(p1), ..., IMPL(pn). This allows the operator mappings to be significantly simplified because it is not necessary to describe how each operand is mapped. The operand mappings are described individually in their own mappings that can be reused in each operator mapping. For example, in the mapping for ISIN, it is not necessary to describe how a set of type S is translated into a list of type L. This is described by a separate mapping. As a consequence of allowing recursion, the translation of an upper level expression cannot simply traverse the parse tree of the expression and replace each mapped object by its right hand side. Instead, the replacement algorithm is directed by the IMPL mapping. That is, the replacement algorithm must call itself whenever IMPL is used in the right hand side of a mapping that is currently being used for replacement.

The other thing to note is that operators may take items other than variables as operands. When a variable v is given as an operand, IMPL(v) is well-defined since all variables must be mapped. The operators may also take explicitly valued constants (e.g. 5, {3, 6}, etc.) and imported variables as operands. This means that an IMPL mapping must be defined to map these types of operands to an equivalent lower level expression of the correct type. In the above example, IMPL(s1) is referenced in the definition of SUBSET and the set {1, 2, 3} is used as an operand to SUBSET in the upper level, so IMPL must define how the set {1, 2, 3} is mapped to LISTDEF (1, 2, 3) and in general how any constant or imported set of integers is mapped to a list of integers. Like operators, a natural constant mapping is defined as follows.

IMPL_0(c: T) == c for any type T that has a built-in supertype

List and set constants are mapped using the natural operator mapping.

```
IMPL_0(LISTDEF(e1, ..., en)) ==
        LISTDEF(IMPL(e1), ..., IMPL(en))
IMPL_0({e1, ..., en}) ==
        {IMPL(e1), ..., IMPL(en)}
IMPL_0(SETDEF e: T (P(e))) ==
        SETDEF e: IMPL(T) (IMPL(P(e)))
```

In these mappings, the values of a built-in type are mapped to the same values, a list of elements is mapped to a list of the implementation of each element, and a set of elements is mapped to a set of the implementation of each element. For each operator mapping IMPL(op(p1, ..., pi: T, ..., pn)) that references IMPL(pi) such that IMPL(c: T) has not been defined and IMPL_0(pi) is either undefined or causes a type mismatch when exchanged for IMPL(pi), the user must define a mapping IMPL(c: T). If no such mapping is required, IMPL(c: T) is defined to be IMPL_0(c: T).

In general, an element of type T in the upper level may be mapped to more than one value of type IMPL(T) at the lower level. For example, consider the mapping from type S to type L. In this mapping, a set v_s in the upper level maps to a list v_l in the lower level such that v_l contains exactly the elements that are in v_s. Lists, however, are ordered, so the elements in v_s may occur in v_l in any order. Therefore, v_s maps to set_size(v_s)! different lists in the lower level. In general, it is undesirable to limit the value that can be chosen in the lower level, which in turn would limit implementation possibilities. For example, if type T was totally ordered, v_l could be chosen such that if t1 and t2 were in v_s and t1 < t2, then t1 would occur before t2 in v_l. In some cases, however, it is not possible to choose one particular value at the lower level. If T is an undefined type, there is no way to describe a transformation from v_s to a specific v_l in the ASTRAL base logic because nothing is known about elements of type T.

To facilitate such mappings, the choose operator "choose e: T (P(e))" is introduced into the ASTRAL language, which corresponds to Hilbert's ε-operator [Lei 69]. The value of the expression "choose e: T (P(e))" is an element e of type T, such that the ASTRAL predicate P(e) holds if such an element exists. If more than one such element exists, the operator nondeterministically chooses one of those elements. If no such element exists, the operator nondeterministically chooses some element of T.

With the choose operator in the language, defining element transformations becomes much simpler. For example, consider the mapping from elements of type S to elements of type L.

```
IMPL(v_s: S) ==
        choose v_l: L
            (   list_len(v_l) = set_size(IMPL_0(v_s))
            &   FORALL e: IMPL(T)
                    (   e ISIN IMPL_0(v_s)
                ↔       EXISTS i: integer
                            (   1 ≤ i
                            &   i ≤ list_len(v_l)
                            &   v_l[i] = e)))
```

This mapping states that a constant or imported variable v_s of type S is mapped to a list v_l of type L such that the length of v_l is equal to the cardinality of v_s and every element in v_s is on v_l. Note that in this mapping, the natural mapping IMPL_0 is referenced to avoid any reference to an upper level term (in this case v_s) in the right hand side of the mapping. Although it is possible to avoid referencing upper level terms in most cases, it is impossible to avoid this in all cases. In particular, when mapping constants, it is sometimes necessary to choose a replacement expression based on the actual value of the constant in the upper level. Most notably, when mapping enumerated types, it is necessary to reference upper level enumerated constants in the right hand side

of the IMPL mapping. For example, consider an upper level enumerated type "gate_u: (open, closed)" that is mapped to a lower level type "gate_l: (open, closed, opening, closing)" such that closed maps to closed and open maps to one of open, opening, or closing. In this mapping, there is no way to map an arbitrary constant of type gate_u to a constant of type gate_l without selecting a value based on gate_u. To accommodate such mappings, a single case split is allowed on the upper level constant that is being mapped such that each case corresponds to an explicit constant value. For example, arbitrary gate_u constants can be mapped as follows:

```
IMPL(c: gate_u) ==
        CASE c OF
            open:
                choose e: gate_l
                    (   e = open
                    |   e = opening
                    |   e = closing)
            close:
                closed
        ESAC
```

When c is an actual constant value, the IMPL replacement algorithm uses the case information to choose the correct replacement. When c is an imported variable, the right side of the mapping is substituted as is, which is well-defined since the upper level type must be globally defined and the interface to the process does not change from the top level, so types available at the top level are still available at the lower levels.

## 7.5. The Mult_Add Circuit

The specification of the refinement of the Mult_Add circuit in figure 7.3.5-2 is shown below using the new parallel refinement mechanism. Each multiplier has a single exported transition "multiply" that computes the product of two inputs. The adder has a single transition "add" that computes the sum of the outputs of the two multipliers.

```
PROCESS SPECIFICATION Multiplier
    EXPORT
        multiply, product
    VARIABLE
        product: integer
    TRANSITION    multiply(a, b: integer)
        ENTRY       [TIME:    2]
            EXISTS t: time
                (   End(multiply, t))
            →       now - End(multiply) ≥ 1
        EXIT
            product = a * b
```

```
PROCESS SPECIFICATION Adder
    IMPORT
        M1, M2, M1.product, M2.product,
        M1.multiply, M2.multiply
    EXPORT      sum
    VARIABLE
        sum: integer
    TRANSITION    add
        ENTRY       [TIME:    1]
                M1.End(multiply, now)
            &   M2.End(multiply, now)
        EXIT
                sum = M1.product + M2.product
```

The lower level consists of two instances of the Multiplier process type and one instance of the Adder process type.

```
PROCESSES
        M1, M2: Multiplier
        A1: Adder
```

The output variable of the upper process is mapped to the sum variable of the adder.

```
IMPL(output) == A1.sum
```

The duration of the compute transition is the sum of the multiply transition and the add transition in the lower level.

```
IMPL(dur1) == 3
```

When compute starts in the upper level, then multiply starts on M1 and M2. When compute ends in the upper level, add ends on A1. When compute is called in the upper level with inputs a, b, c, and d, multiply is called on M1 with inputs a and b and multiply is called on M2 with inputs c and d.

```
IMPL(Start(compute, now)) ==
        M1.Start(multiply, now)
    &   M2.Start(multiply, now)
IMPL(End(compute, now)) ==
        A1.End(add, now)
```

```
IMPL(Call(compute(a, b, c, d), now)) ==
        M1.Call(multiply(a, b), now)
    &   M2.Call(multiply(c, d), now)
```

## 7.6. Proof Obligations for Parallel Refinement Mechanism

The goal of the refinement proof obligations is to show that any properties that hold in the upper level hold in the lower level without actually reproving the upper level properties in the lower level. In order to show this, it must be shown that the lower level correctly implements the upper level. ASTRAL properties are interpreted over execution histories, which are described by the values of state variables and the start, end, and call times of transitions at all times in the past back to the initialization of the system. A lower level correctly implements an upper level if the implementation of the execution history of the upper level is equivalent to the execution history of the lower level. This corresponds to proving the following four statements.

(V) Any time a variable has one of a set S of possible values in the upper level, the implementation of the variable has one of a subset of the implementation of S in the lower level.

(C) Any time the implementation of a variable changes in the lower level, a transition ends in the upper level.

(S) Any time a transition starts in the upper level, the implementation of the transition starts in the lower level and vice-versa.

(E) Any time a transition ends in the upper level, the implementation of the transition ends in the lower level and vice-versa.

If these four items can be shown, then any property that holds in the upper level is preserved in the lower level because the structures over which the properties are interpreted is identical over the implementation mapping.

### 7.6.1. Direct Proof Obligations

As a first attempt at defining the proof obligations for the parallel refinement mechanism, the proofs of (V), (C), (S), and (E) are carried out directly. That is, they can each be expressed explicitly using the variable, start, and end mappings and then proved at all times. The equivalence can be proved inductively on the time domain. For simplicity, a discrete time domain will be used in the following discussion. Thus, in the proof obligations, all the mappings can be assumed to hold up until but not including some time T0, and then it must be proved that each mapping holds at T0.

Let UL_state of type [ul_bool_expr → bool] and LL_state of type [ll_bool_expr → bool] be uninterpreted functions such that UL_state(ul_bool_expr) is true if and only if ul_bool_expr holds in the upper level and LL_state(ll_bool_expr) is true if and only if ll_bool_expr holds in the lower level, where ul_bool_expr is a boolean expression involving upper level terms and ll_bool_expr is a boolean expression involving lower level terms. For example, the expression UL_state(Start(compute, t)(t)) is true if and only if in the execution of the upper level process Mult_Add, compute starts at time t.

Two assumptions, A_const and A_call, will be made in the proof obligations. Let const_ul denote the set of constants in the upper level, cval_ul denote the set of possible constant values in the upper level, trans_ul denote the set of transitions in the upper level, and var_ul denote the set of variables in the upper level. A_const states that if a constant c_ul has one of a possible set of values s_cv_ul, then the implementation of c_ul has one of a subset of the values in s_cv_ul in the lower level.

```
(A_const)   FORALL c_ul: const_ul, s_cv_ul: set(type(c_ul))
              (EXISTS s2_cv_ul: set(type(c_ul))
                 (   s2_cv_ul CONTAINED_IN s_cv_ul
                 &  (   UL_state(c_ul ISIN s_cv_ul)
                    ↔ LL_state(IMPL(c_ul ISIN s2_cv_ul)))))
```

A_call states that any time a transition tr_ul is called at the upper level, the implementation of a call to tr_ul occurs at the lower level.

```
(A_call)    FORALL t: time, tr_ul: trans_ul
                (   UL_state(Call(tr_ul, t)(t))
                ↔ LL_state(IMPL(Call(tr_ul, t))(t)))
```

The base case obligations are shown below. For variables, if v_ul has one of a possible set of values in s_cv_ul at time zero in the upper level, then the implementation of v_ul has one of a subset of the values in the implementation of s_cv_ul at time zero in the lower level. For variable changes, no base case obligation is needed because no expression can change at time zero, by the definition of the change operator. For transitions, if tr_ul starts at time zero in the upper level, then the implementation of the start of tr_ul must hold at time zero in the lower level. Note that although tr_ul cannot end at time zero in the upper level, the implementation of the end of tr_ul can be an arbitrary expression that does not necessarily involve transition ends, so it can hold at time zero, thus the base case for ends must be similarly shown.

```
(V_b)   A_call & A_const
    → FORALL v_ul: var_ul, s_cv_ul: set(type(v_ul))
        (EXISTS s2_cv_ul: set(type(v_ul))
            (   s2_cv_ul CONTAINED_IN s_cv_ul
            &   (   UL_state((v_ul ISIN s_cv_ul)(0))
                ↔ LL_state(IMPL(v_ul ISIN s2_cv_ul)(0)))))

(S_b)   A_call & A_const
    → FORALL tr_ul: trans_ul
        (   UL_state(Start(tr_ul, 0)(0))
        ↔ LL_state(IMPL(Start(tr_ul, 0))(0)))

(E_b)   A_call & A_const
    → FORALL tr_ul: trans_ul
        (   UL_state(end(tr_ul, 0)(0))
        ↔ LL_state(IMPL(end(tr_ul, 0))(0)))
```

The inductive assumption A_induct is defined as follows.

```
(A_induct)      FORALL t: time
                    (   t < T0
                →       (   FORALL v_ul: var_ul, s_cv_ul, s2_cv_ul: set(type(v_ul))
                            (EXISTS s2_cv_ul: set(type(v_ul))
                                (   s2_cv_ul CONTAINED_IN s_cv_ul
                                &   (   UL_state((v_ul ISIN s_cv_ul)(t))
                                    ↔ LL_state(IMPL(v_ul ISIN s2_cv_ul)(t)))))
                        &   FORALL tr_ul: trans_ul
                                (   UL_state(Start(tr_ul, t)(t))
                                ↔ LL_state(IMPL(Start(tr_ul, t))(t)))
                        &   FORALL tr_ul: trans_ul
                                (   UL_state(end(tr_ul, t)(t))
                                ↔ LL_state(IMPL(end(tr_ul, t))(t)))))
```

The induction step obligations are similar to the base case obligations, but they must be proved at an arbitrary time T0 instead of at time zero. In addition, the mappings are assumed up until T0. The induction step obligations are shown below.

```
(V_i)    A_call & A_const & A_induct
      → FORALL v_ul: var_ul, s_cv_ul: set(type(v_ul))
           (EXISTS s2_cv_ul: set(type(v_ul))
              (   s2_cv_ul CONTAINED_IN s_cv_ul
              &  (   UL_state((v_ul ISIN s_cv_ul)(T0))
                 ↔ LL_state(IMPL(v_ul ISIN s2_cv_ul)(T0)))))

(C_i)    A_call & A_const & A_induct
      → FORALL v_ul: var_ul, tr_ul: trans_ul
           (   LL_state(change(IMPL(v_ul), T0)(T0))
           →   UL_state(end(tr_ul, T0)(T0)))

(S_i)    A_call & A_const & A_induct
      → FORALL tr_ul: trans_ul
           (   UL_state(Start(tr_ul, T0)(T0))
           ↔ LL_state(IMPL(Start(tr_ul, T0))(T0)))

(E_i)    A_call & A_const & A_induct
      → FORALL tr_ul: trans_ul
           (   UL_state(end(tr_ul, T0)(T0))
           ↔ LL_state(IMPL(end(tr_ul, T0))(T0)))
```

To prove these obligations, axioms must be introduced for the manipulation of expressions involving UL_state and LL_state. In general, new information can be deduced from any UL_state/LL_state expression by using the axiom system utilized for the intra-level proofs in PVS. For example, if UL_state(Start(tr1, t1)(t)) holds, it can be deduced that UL_state(Fired(tr1, t1)) holds and hence that UL_state(Entry(tr1, t1)) holds. In addition, axioms are defined for splitting and combining UL_state/LL_state expressions. For example, if LL_state(A & B) holds, it can be deduced that LL_state(A) & LL_state(B) hold, and vice-versa. There are also axioms to reduce UL_state/LL_state expressions to constants, such as UL_state(FALSE) = FALSE. Finally, there are axioms to deduce that information about the operating environment in the upper and lower levels is the same at all times. For example, for an imported variable P.v, if UL_state(P.v(t)), then it can be deduced that LL_state(P.v(t)). Basically, it can be assumed that all other processes besides the one being refined behave the same in the executions of the upper and lower levels.

These obligations are almost direct translations of the requirements (V), (C), (S), and (E) above. Therefore, they are sufficient to show that a lower level correctly implements an upper level. They suffer, however, from a number of major drawbacks. First, they require the introduction of a new axiom system to manipulate UL_state and LL_state expressions. This means that the user must learn

an additional axiom system to perform the proofs. Additionally, they require the user to reason about both the upper and lower levels in the proofs. Finally, they require additional mechanisms to handle nondeterministic systems as discussed below.

In a specification with nondeterministic behavior such as multiple transitions enabled at the same time or a nondeterministic choice of variable values in an exit assertion, there is a need for some type of "oracle" that relates nondeterministic choices made in the upper and lower levels. That is, if a choice c is made in the upper level, then c is made in the lower level. Normally, the inductive hypothesis subsumes the need for an oracle, but it is necessary at the time the mapping is to be proved (i.e. time 0 in the base case and time T0 in the induction step). For instance, suppose the mapping for Start(tr1, t) is being proved, but whenever tr1 is enabled, tr2 is also enabled. Without an oracle stating that tr1 is chosen at t, there is no way to prove that if tr1 starts in the upper level that the implementation of tr1 will actually start in the lower level because the implementation of tr2 might actually start.

Such an oracle is difficult to define, given that mappings between an upper level and lower level can be complex expressions and the fact that while proving, it is not known if the lower level is actually a refinement of the upper level so that the same things will be enabled, same variable choices will occur, etc. A first attempt at defining an oracle might be:

- For transitions, if at time t, tr1, ..., trn are enabled in the upper level, the process is idle, and tri fires, then if at time t, IMPL(tr1), ..., IMPL(trn) are enabled in the lower level and the appropriate processes are idle, then IMPL(tri) occurs.
- For variables, if at time t, there exists a choice between values v1, ..., vn to give a variable var in an exit assertion in the upper level and vi is chosen, then if at time t, there exists a choice between values IMPL(v1), ..., IMPL(vn) to give an expression IMPL(var) in an exit assertion in the lower level, then IMPL(vi) is chosen.

Even if such an oracle could be defined, it would overly complicate the proofs of any nondeterministic system. In the end, proving (V), (C), (S), and (E) directly requires too much overhead for even simple implementations. Thus, these requirements will be proven indirectly instead.

### 7.6.2. Indirect Proof Obligations

Instead of proving directly that the mappings hold at all times, it will be shown that the mappings hold indirectly by proving that they preserve the axiomatization of the ASTRAL abstract machine, thus preserve any reasoning performed in the upper level. This can be done by proving the implementation of each abstract machine axiom. Note that these obligations do not consider further assumption clauses and are invalid for transitions with exceptions since in that case, Start(tr1, t1)(t1) ≠ Fired(tr1, t1), which is assumed in the obligations. Further assumption clauses can be supported using techniques similar to those in [CKM 95]. Exceptions are a syntactic shorthand, thus can be supported by appropriate specification rewriting or by a slight extension to the transition mappings.

To perform the proofs, the following assumption must be made about calls to transitions in each lower level process.

```
impl_call: ASSUMPTION
    (FORALL (tr_ll: trans_ll, t1: time):
        Exported(tr_ll) AND
        Call(tr_ll, t1)(t1) IMPLIES
            (EXISTS (tr_ul: trans_ul):
                (FORALL (t2: time):
                    IMPL(Call(tr_ul, t2)(t2)) IMPLIES
                        Call(tr_ll, t2)(t2)) AND
                IMPL(Call(tr_ul, t1)(t1))))
```

This assumption states that any time a lower level exported transition is called, there is some call mapping that references a call to the transition that holds at the same time. This means that if one transition of a "conjunctive" mapping is called, then all transitions of the mapping are called. That is, it is not possible for a lower level transition to be called such that the call mapping for some upper level transition does not hold. For example, consider the mapping for the compute transition of the Mult_Add circuit.

```
IMPL(Call(compute(a, b, c, d), now)) ==
        M1.Call(multiply(a, b), now)
    &   M2.Call(multiply(c, d), now)
```

In this case, impl_call states that any time multiply is called on M1, multiply is called on M2 at the same time and vice-versa. Note that impl_call is not an environmental assumption. The call mappings explicitly state how calls at the lower level are generated from calls at the upper level. The impl_call assumption expresses this fact and makes sure that lower level calls do not occur in any other way but as stated.

An assumption is also needed to assure that whenever the parameters of an upper level exported transition are distributed among multiple transitions at the lower level, the collection of parameters for which the lower level transitions execute come from a single set of call parameters. For example, in the Mult_Add circuit, the compute transition in the upper level may be called with two sets of parameters {1, 2, 3, 4} and {5, 6, 7, 8} at the same instant. In the lower level implementation, the multiply transition of each multiplier takes two of the parameters from each upper level call. Thus, in the example, multiply is enabled on M1 for {1, 2} and {5, 6} and on M2 for {3, 4} and {7, 8}. Without an appropriate assumption, M1 may choose {1, 2} and M2 may choose {7, 8}, thus computing the product for {1, 2, 7, 8}, which was not requested at the upper level.

The implementation of the call_fire_parms axiom provides the appropriate restriction. The *impl_call_fire_parms* assumption states that any time the mapped start of an exported parameterized transition occurs, the mapped parameters for which the mapped transition fired came from the set of mapped call parameters that have not yet been serviced at that time.

```
impl_call_fire_parms: ASSUMPTION
    (FORALL (tr1: transition, t3: time):
        Exported(tr1) AND
        Has_Parms(tr1) AND
        IMPL(Start(tr1, t3)(t3)) IMPLIES
            (EXISTS (t1: time):
                t1 ≤ t3 AND
                IMPL(Call(tr1, t1)(t1)) AND
                member(IMPL(Fire_Parms(tr1, t3)), IMPL(Call_Parms(tr1, t1))) AND
                (FORALL (t2: time):
                    t1 ≤ t2 AND t2 < t3 AND
                    IMPL(Start(tr1, t2)(t2)) IMPLIES
                        IMPL(Fire_Parms(tr1, t2)) ≠ IMPL(Fire_Parms(tr1, t3)))))
```

Note that impl_call_fire_parms must be stated as an assumption and not as an obligation because there is no way to specify the lower level processes such that they will collectively make the same nondeterministic choice of which set of call parameters to service. Since this behavior cannot be specified, impl_call_fire_parms cannot be proved as an obligation. The other portions of the impl_call_fire_parms assumption hold by the restriction on call mappings for exported parameterized transitions as mentioned in section 7.4.2.

In the axiomatization of the ASTRAL abstract machine, the predicate “Fired(tr1, t1)” is used to denote that the transition tr1 fired at time t1. If Fired(tr1, t1) holds, then it is derivable that Start(tr1, t1)(t1) and End(tr1, t1 + Duration(tr1))(t1 + Duration(tr1)). Additionally, since End(tr1, t1)(t1) can only be derived when Fired(tr1, t1 - Duration(tr1)) holds and the time parameter of Fired is restricted

to be nonnegative, it is known that an end can only occur at times greater than or equal to the duration of the transition. In the parallel refinement mechanism, the user maps the start and end of upper level transitions, so it is unknown whether these properties of end still hold. Since the axioms rely on these properties, they must be proved explicitly as proof obligations. The *impl_end1* obligation ensures that the mapped end of a transition can only occur after the mapped duration of the transition has elapsed.

```
impl_end1: OBLIGATION
    (FORALL (tr1: transition, t1: time):
        IMPL(End(tr1, t1)(t1)) IMPLIES
            t1 ≥ IMPL(Duration(tr1)))
```

The *impl_end2* obligation ensures that for every mapped start of a transition, there is a corresponding mapped end of the transition, that for every mapped end, there is a corresponding mapped start, and that mapped starts and mapped ends are separated by the mapped duration of the transition.

```
impl_end2: OBLIGATION
    (FORALL (tr1: transition, t1: time, t2: time):
        t1 = t2 - IMPL(Duration(tr1)) IMPLIES
            (IMPL(Start(tr1, t1)(t1)) IFF
                IMPL(End(tr1, t2)(t2))))
```

The following obligations are the mappings of the ASTRAL abstract machine axioms except for call_fire_parms, which is discussed above. The *impl_trans_entry* obligation ensures that any time the mapped start of a transition occurs, the mapped entry assertion of the transition holds.

```
impl_trans_entry: OBLIGATION
    (FORALL (tr1: transition, t1: time):
        IMPL(Start(tr1, t1)(t1)) IMPLIES
            IMPL(Entry(tr1, t1)))
```

The *impl_trans_exit* obligation ensures that any time the mapped end of a transition occurs, the mapped exit assertion of the transition holds.

```
impl_trans_exit: OBLIGATION
    (FORALL (tr1: transition, t1: time):
        IMPL(End(tr1, t1)(t1)) IMPLIES
            IMPL(Exit(tr1, t1)))
```

The *impl_trans_called* obligation ensures that any time the mapped start of an exported transition occurs, a mapped call has been issued to the transition but not yet serviced.

```
impl_trans_called: OBLIGATION
    (FORALL (tr1: transition, t1: time):
        IMPL(Start(tr1, t1)(t1)) AND
        Exported(tr1) IMPLIES
            IMPL(Issued_Call(tr1, t1)))
```

The *impl_trans_mutex* obligation ensures that any time the mapped start of a transition occurs, no other mapped start of a transition can occur until the mapped duration of the transition has elapsed.

```
impl_trans_mutex: OBLIGATION
    (FORALL (tr1: transition, t1: time):
        IMPL(Start(tr1, t1)(t1)) IMPLIES
            (FORALL (tr2: transition):
                tr2 ≠ tr1 IMPLIES
                    NOT IMPL(Start(tr2, t1)(t1))) AND
            (FORALL (tr2: transition, t2: time):
                t1 < t2 AND t2 < t1 + IMPL(Duration(tr1)) IMPLIES
                    NOT IMPL(Start(tr2, t2)(t2))))
```

The *impl_trans_fire* obligation ensures that any time the mapped entry assertion of a transition holds, a mapped call has been issued to the transition but not yet serviced if the transition is exported, and no mapped start of a transition has occurred within its mapped duration of the given time, a mapped start will occur.

```
impl_trans_fire: OBLIGATION
    (FORALL (t1: time):
        (EXISTS (tr1: transition):
            IMPL(Enabled(tr1, t1))) AND
        (FORALL (tr2: transition, t2: time):
            t1 - IMPL(Duration(tr2)) < t2 AND t2 < t1 IMPLIES
                NOT IMPL(Start(tr2, t2)(t2))) IMPLIES
        (EXISTS (tr1: transition): IMPL(Start(tr1, t1)(t1))))
```

The *impl_vars_no_change* obligation ensures that mapped variables only change value when the mapped end of a transition occurs.

```
impl_vars_no_change: OBLIGATION
    (FORALL (t1: time, t3: time):
        t1 ≤ t3 AND
        (FORALL (tr2: transition, t2: time):
            t1 < t2 AND t2 ≤ t3 IMPLIES
                NOT IMPL(End(tr2, t2)(t2))) IMPLIES
            (FORALL (t2: time):
                t1 ≤ t2 AND t2 ≤ t3 IMPLIES
                    IMPL(Vars_No_Change(t1, t2))))
```

The *impl_initial_state* obligation ensures that the mapped initial clause holds at time 0.

```
impl_initial_state: OBLIGATION
    IMPL(Initial(0))
```

Besides the abstract machine axioms, the local proofs of ASTRAL process specifications can also reference the local axiom clause of the process. Since this clause can be used in proofs and the constants referenced in the clause can be implemented at the lower level, the mapping of the local axiom clause of the upper level must be proved as a proof obligation. The *impl_local_axiom*

obligation ensures that the mapped axiom clause holds at all times. In order to prove this obligation, it may be necessary to specify local axioms in the lower level processes that satisfy the implementation of the upper level axiom clause.

```
impl_local_axiom: OBLIGATION
    (FORALL (t1): IMPL(Axiom(t1)))
```

To prove the above obligations, the abstract machine axioms can be used in each lower level process. For example, to prove the impl_initial_state obligation, the initial clause of each lower level process can be asserted with the initial_state axiom.

Unlike the obligations that attempt to prove the mappings directly, the above obligations do not require a new axiom system, do not require an oracle for nondeterministic choices, and only use information about lower level processes.

### 7.6.3. Correctness of Indirect Proof Obligations

The proof obligations for the parallel refinement mechanism as stated above are sufficient to show that for any invariant I that holds in the upper level, IMPL(I) holds in the lower level. Consider the correctness criteria (V), (C), (S), and (E) above. (V) is satisfied because by impl_initial_state, the values of the implementation of the variables in the lower level must be consistent with the values in the upper level. Variables in the upper level only change when a transition ends and at these times, the implementation of the variables in the lower level change consistently by impl_trans_exit. (C) is satisfied because the implementation of the variables in the lower level can only change value when the implementation of a transition ends by impl_vars_no_change. The forward direction of (S) is satisfied because whenever an upper level transition fires, a lower level transition will fire by impl_trans_fire. The reverse direction of (S) is satisfied because whenever the implementation of a transition fires in the lower level, its entry assertion holds by impl_trans_entry, it has been called by impl_trans_called, and no other transition is in the middle of execution by impl_trans_mutex. (E) is satisfied because (S) is satisfied and by impl_end1 and impl_end2, any time a start occurs, a corresponding end occurs and vice-versa.

More formally, any time an invariant I can be derived in the upper level, it is derived by a sequence of transformations from I to TRUE, $I \vdash_{f1/a1} I1 \vdash_{f2/a2} \ldots \vdash_{fn/an} TRUE$, where each transformation fi/ai corresponds to the application of a series fi of first-order logic axioms and a single abstract machine axiom ai. Since the implementation of each axiom of the ASTRAL abstract machine is preserved by the parallel refinement proof obligations, a corresponding proof at the lower level $IMPL(I) \vdash_{f1'/impl\_a1} IMPL(I1) \vdash_{f2'/impl\_a2} \ldots \vdash_{fn'/impl\_an} TRUE$ can be constructed by replacing the application of each

abstract machine axiom ai by impl_ai. Additionally, each series fi of first-order logic axioms is replaced by a series fi′ that takes any changes to the types of variables and constants into consideration.

## 7.7. Proof of Mult_Add Circuit Refinement

This section shows the application of the parallel refinement proof obligations to the Mult_Add circuit.

### 7.7.1. Impl_end1 Obligation

```
FORALL t1: time
    (   past(A1.End(add, t1), t1)
→       t1 ≥ 3)
```

By the entry assertion of add, multiply must end when add starts. The duration of add is 1 and the duration of multiply is 2, so the earliest add can end is at time 3. Thus, impl_end1 holds.

### 7.7.2. Impl_end2 Obligation

```
FORALL t1: time
    (   past(M1.Start(multiply, t1 - 3), t1 - 3)
    &   past(M2.Start(multiply, t1 - 3), t1 - 3)
↔       past(A1.End(add, t1), t1))
```

For forward direction, it must be shown that add starts on A1 at t1 - 1. From the antecedent, multiply ends on both M1 and M2 at t1 - 1 so the entry assertion of add holds on A1 at time t1 - 1. A1 must be idle or else from the entry of add, multiply ended in the interval (t1 - 2, t1 - 1), which is not possible since multiply was still executing on M1 and M2 in that interval. Therefore, add starts at t1 - 1 on A1, thus ends at t1.

For the reverse direction, add starts on A1 at t1 - 1 from the antecedent. From the entry of add, multiply ends on both M1 and M2 at t1 - 1, so starts at t1 - 3. Thus, the reverse direction holds and impl_end2 holds.

### 7.7.3. Impl_trans_entry Obligation

```
FORALL t1: time
    (   past(M1.Start(multiply, t1), t1)
    &   past(M2.Start(multiply, t1), t1)
→       TRUE)
```

This formula trivially holds.

### 7.7.4. Impl_trans_exit Obligation

```
FORALL t1: time
   (   past(A1.End(add, t1), t1)
→      FORALL a, b, c, d: integer
          (   past(M1.Start(multiply(a, b), t1 - 3), t1 - 3)
          &   past(M2.Start(multiply(c, d), t1 - 3), t1 - 3)
          →      past(A1.sum, t1) = a * b + c * d))
```

By the exit assertion of add, past(A1.sum, t1) = past(M1.product, t1 - 1) + past(M2.product, t1 - 1). From the entry of add, multiply ends on both M1 and M2 at t1 - 1. Since multiply ends on M1 and M2 at t1 - 1, it starts on M1 and M2 at t1 - 3 for two pairs of parameters (a, b) and (c, d), respectively, which were provided by the external environment. By the exit assertion of multiply, past(M1.product, t1 - 1) = a * b and past(M2.product, t1 - 1) = c * d, so past(A1.sum, t1) = a * b + c * d. Thus, impl_trans_exit holds.

### 7.7.5. Impl_trans_called Obligation

```
FORALL t1: time
   (   past(M1.Start(multiply, t1), t1)
   &   past(M2.Start(multiply, t1), t1)
→      EXISTS t2: time
          (   t2 ≤ t1
          &   past(M1.Call(multiply, t2), t1)
          &   past(M2.Call(multiply, t2), t1)
          &   FORALL t3: time
                 (   t2 ≤ t3 & t3 < t1
              →      ~   (   past(M1.Start(multiply, t3), t3)
                         &   past(M2.Start(multiply, t3), t3)))))
```

Since multiply started on M1 (M2) at time t1, by trans_called applied on process M1 (M2), multiply was called at some time t2 ≤ t1 and multiply has not started on M1 (M2) in the interval [t2, t1). By impl_call, the time that multiply was called on M1 and M2 must be the same. Thus, impl_trans_called holds.

### 7.7.6. Impl_trans_mutex Obligation

```
FORALL t1: time
   (   past(M1.Start(multiply, t1), t1)
   &   past(M2.Start(multiply, t1), t1))
→      FORALL t2: time
          (   t1 < t2 & t2 < t1 + 3
          →      ~   (   past(M1.Start(multiply, t2), t2)
                     &   past(M2.Start(multiply, t2), t2))))
```

Since multiply started on M1 (M2) at time t1, by trans_mutex applied on process M1 (M2), nothing can fire on M1 (M2) until time t1 + 2. The multiply transition, however, is the only transition of M1 (M2) and multiply is not enabled until 1 time unit after the end of the last multiply, so cannot start until t1 + 3. Thus, impl_trans_mutex holds.

### 7.7.7. Impl_trans_fire Obligation

```
FORALL t1: time
   (   EXISTS t2: time
          (   t2 ≤ t1
          &   past(M1.Call(multiply, t2), t1)
          &   past(M2.Call(multiply, t2), t1)
          &   FORALL t3: time
                 (   t2 ≤ t3 & t3 < t1
              →      ~   (   past(M1.Start(multiply, t3), t3)
                         &   past(M2.Start(multiply, t3), t3))))
   &   FORALL t2: time
          (   t1 - 3 < t2 & t2 < t1
       →      ~   (   past(M1.Start(multiply, t2), t2)
                  &   past(M2.Start(multiply, t2), t2)))
→      past(M1.Start(multiply, t1), t1)
   &   past(M2.Start(multiply, t1), t1))
```

To prove this obligation, it is first necessary to prove that M1.Start(multiply) and M2.Start(multiply) always occur at the same time. This can be proved inductively. At time 0, both M1 and M2 are idle. By impl_call, if multiply is called on either M1 or M2, multiply is called on both M1 and M2. If both are called, then both fire because the entry assertion of multiply is true since at time 0, multiply cannot have ended. If neither is called, then neither can fire. For the inductive case, assume M1.Start(multiply) and M2.Start(multiply) have occurred at the same time up until time T0. Suppose multiply occurs on M1 (M2), then M1 (M2) was idle, multiply has been called since the last start, and it has been at least one time unit since multiply ended on M1 (M2). M2 (M1) cannot be executing multiply at T0 or else M1 (M2) must also be executing multiply by the inductive hypothesis, thus M2 (M1) must be idle. Similarly, it must have been at least one time unit since multiply ended on M2 (M1). By impl_call, multiply must have been called on M2 (M1) since it was called on M1 (M2). Thus, multiply is enabled on M2 (M1), so must fire. Therefore, M1.Start(multiply) and M2.Start(multiply) always occur at the same time. From this fact

```
FORALL t3: time
   (   t2 ≤ t3 & t3 < t1
→      ~   (   past(M1.Start(multiply, t3), t3)
          &   past(M2.Start(multiply, t3), t3)))
```

is equivalent to

```
FORALL t3: time
    (   t2 ≤ t3 & t3 < t1
→       ~past(M1.Start(multiply, t3), t3)
    &   ~past(M2.Start(multiply, t3), t3))
```

Since nothing has started in the interval (t1 - 3, t1), nothing can end in the interval (t1 - 1, t1 + 2), thus the entry assertion of multiply on M1 is satisfied. Since the entry of multiply holds, multiply has been called but not yet serviced, and M1 is idle, multiply starts on M1 by trans_fire. Since multiply always starts on both M1 and M2 at the same time as shown above, impl_trans_fire holds.

### 7.7.8. Impl_vars_no_change Obligation

```
FORALL t1, t3: time
    (   t1 ≤ t3
    &   FORALL t2: time
            (   t1 < t2 & t2 ≤ t3
        →       ~past(A1.End(add, t2), t2))
→       FORALL t2: time
            (   t1 ≤ t2 & t2 ≤ t3
        →       past(A1.sum, t1) = past(A1.sum, t2)))
```

This formula holds by the vars_no_change axiom applied on process A1.

### 7.7.9. Results of Proof Obligations

The impl_initial_state and impl_local_axiom obligations trivially hold as the initial and axiom clauses are both "TRUE". Since the proof obligations hold for the Mult_Add circuit, the lower level is a correct refinement of the upper level and thus the implementation of the upper level invariant, shown below, holds in the lower level.

```
FORALL t1: time, a, b, c, d: integer
    (   M1.Start(multiply(a, b), t1 - 3)
    &   M2.Start(multiply(c, d), t1 - 3)
→       FORALL t2: time
            (   t1 + dur1 ≤ t2
            &   t2 ≤ now
        →       past(A1.sum, t2) = a * b + c * d))
```

The previous example has shown that the parallel refinement mechanism can express the parallel implementation of a simple system in a simple and straightforward manner. More importantly, the proof obligations for a simple implementation were themselves simple. Now, the refinement of a much more complex system will be discussed along with the application of the proof obligations to it. From the following example, it will be shown that the parallel refinement mechanism can be used to express very complex parallel implementations, but at a cost of complicating the proofs of the proof obligations.

## 7.8. Parallel Phone System

This section discusses the parallel refinement of a central control of a phone system, which is based on the phone system discussed in section 2.1.6, but considers only local calls and not long distance calls. The phone system consists of a set of phones that need various services (e.g. getting a dial tone, processing digits entered into the phone, making a connection to the requested phone, etc.) as well as a set of central controls that perform the services.

The specification of the central control, which is the core of the whole system, is articulated into three layers. The goal of the top level is to provide an abstract and global view of the supplied services in such a way that the user can have a complete and precise knowledge of the external behavior of the central control, both in terms of functions performed and in terms of service times, but the designer still has total freedom for implementation policies. In fact, as a result, the description provided in [CGK 97] is just an alternative implementation of the top level description given below, which differs from the present implementation in that services are granted sequentially rather than in parallel. To achieve this goal (i.e. to allow the implementation of services both asynchronously in parallel and strictly sequentially), the top level is specified such that a set of services can start and a set of services can end at every time unit in the system (for simplicity, discrete time is assumed).

### 7.8.1. Top Level of the Central Control

The central control process has two transitions, Begin_Serve and Complete_Serve, each with duration serve_dur, where 2 * serve_dur is a divisor of the duration of every central control service. Begin_Serve and Complete_Serve execute cyclically and indicate the start and end, respectively, of the execution of several different functions, each of which corresponds to a transition of the [CGK 97] central control (excluding the part referring to long distance calls). Since the goal is to allow different functions to be executed in parallel, functions are allowed to begin in parallel in Begin_Serve and complete in parallel in Complete_Serve. Two different functions that begin service at the same time do not necessarily complete service at the same time, thus different functions can have different durations. The functions executed by Begin_Serve and Complete_Serve are:

| | | | |
|---|---|---|---|
| (GDT) | Give_Dial_Tone | (ERB) | Enable_Ringback |
| (PD) | Process_Digit | (DRBP) | Disable_Ringback_Pulse |
| (PC) | Process_Call | (ST) | Start_Talk |
| (ER) | Enable_Ring | (TC) | Terminate_Connection |
| (DRP) | Disable_Ring_Pulse | (GA) | Generate_Alarm |

The functions are specified similarly to those in [CGK 97], but instead of specifying a separate transition for each, the entry assertion of Begin_Serve and the exit assertion of Complete_Serve are the conjunctions of the entry assertions of each function and the exit assertions of each function, respectively. In addition, each function is specified to service a set of phone processes instead of a single phone. Additionally, a variable "serving(phone): bool" is declared, which is true if and only if a phone has begun to be serviced but has not yet completed being serviced and is initially false for all phones. For each function g above, a set of phones W_g is defined, which is the set of phones waiting to be serviced by the function g. These sets are described as follows.

```
(W_GDT)   setdef P: phone (   P.Offhook & Phone_State(P) = Idle)
(W_PD)    setdef P: phone (   P.Offhook
                          &  (   Phone_State(P) = Ready_To_Dial
                             |   Phone_State(P) = Dialing
                             &   Count(P) < 7
                             &   P.End(Enter_Digit) > Change(Count(P))))
(W_PC)    setdef P: phone (   P.Offhook & Count(P) = 7
                          &   Phone_State(P) = Dialing
                          &   ~Get_ID(Number(P)).Offhook
                          &   Phone_State(Get_ID(Number(P)))= Idle)
...
```

In general, for each function g, the set W_g is the set of phones that satisfy the entry assertion of the transition of [CGK 97] associated with g. Let K_W_g be the maximum number of phones that can be served by the function g at any time and K_max be the maximum number of phones that can be served by any function at any time. Additionally, let Dur_g be the duration of the function g and Exit_g(P) be the exit assertion of the [CGK 97] transition associated with g applied to the phone P. In the following definitions of Begin_Serve and Complete_Serve, quantification over the functions g of the central control is used to simplify the presentation. The quantifiers can be expanded out over the 10 functions of the central control. For each function g, let serving_g be defined as setdef P: phone (serving(P) & P ISIN past(W_g, change(serving(P)) - serve_dur). That is, serving_g specifies the set of phones currently being served by g. This definition is necessary since each W_g changes dynamically over time according to the behavior of the phone processes. Additionally, let serving_all be defined as setdef P: phone (serving(P)), which is the set of all phones being served.

Begin_Serve(S) is enabled for a nonempty set S when

- now is a multiple of 2 * serve_dur
- S is the union of the sets S_g, where for each function g,
    1. S_g only contains phones that are in W_g
    2. S_g does not contain any phones currently being served
    3. S_g ∪ serving_g contains at most K_W_g phones

- S ∪ serving_all contains at most K_max phones
- if S ∪ serving_all contains less than K_max phones, then for each function g, either S_g ∪ serving_g is at maximum capacity or all the phones of W_g are in S ∪ serving_all

The exit assertion of Begin_Serve specifies that the set of phones that begin being served is equal to the set parameter S.

```
Begin_Serve(S: nonempty_set_of_phone)
    ENTRY       [TIME:    serve_dur]
            now MOD (2 * serve_dur) = 0
        &   EXISTS S_GDT, S_PD, ..., S_TC: set_of_phone
                (   S_GDT CONTAINED_IN W_GDT
                &   S_GDT SET_DIFF serving_all = S_GDT
                &   set_size(S_GDT UNION serving_GDT) ≤ K_W_GDT
                &   ...
                &   S = S_GDT UNION S_PD UNION ... UNION S_TC
                &   set_size(S UNION serving_all) ≤ K_max
                &   (   set_size(S UNION serving_all) < K_max
                →       FORALL g: central function
                            (   set_size(S_g UNION serving_g) = K_W_g
                            |   W_g CONTAINED_IN S_g UNION serving_all)))
    EXIT
            FORALL P: phone
                (IF P ISIN S
                THEN serving(P)
                ELSE serving(P) ↔ serving′(P)
                FI)
```

Complete_Serve is enabled when

- now + serve_dur is a multiple of 2 * serve_dur
- enough time has elapsed (i.e. the duration of the function) since some phone began being served such that service for that phone can be completed

The exit assertion of Complete_Serve specifies that all phones that have been served for at least the duration of the appropriate function will complete being served with the variables for each phone changed according to the exit assertion of the appropriate function.

```
Complete_Serve
    ENTRY       [TIME:    serve_dur]
            now MOD (2 * serve_dur) = serve_dur
        &   EXISTS P: phone, g: central function
                (   P ISIN serving_g
                &   now - change(serving(P)) + serve_dur ≥ Dur_g - serve_dur)
    EXIT
            FORALL P: phone, g: central function
                (IF     P ISIN past(serving_g, now - serve_dur)
                    &   now - past(change(serving(P)), now - serve_dur) ≥
                                Dur_g - serve_dur
```

```
THEN
        Exit_g(P)
    &   ~serving(P)
ELSE
        serving(P) ↔ serving′(P)
FI)
```

Notice that the above specifications automatically define the updating of the set of phones. That is, each set W_g is updated according to changes in external processes (e.g. phones becoming offhook) and according to the changes made by the exit assertion of Complete_Serve.

### 7.8.2. Sequential Refinement of Top Level Central Control

After redefining the top level specification of the central control, it becomes possible to show (assuming discrete time) that the original central control specification in [CGK 97] without long distance is one possible second level implementation of the top level given in the previous section. Without loss of generality, it is assumed that the transitions of [CGK 97] only begin execution at times that are multiples of 2 * serve_dur. This essentially says that 2 * serve_dur is the fastest the system can recognize external changes. This can be accomplished by assuming the clause now MOD (2 * serve_dur) = 0 is conjoined to every entry assertion. The key to this refinement is mapping K_max to one. This means that only a single function can occur at any given time in the system.

#### *7.8.2.1. IMPL Mapping*

The IMPL mapping for the sequential refinement of the top level central control is shown below. K_max is mapped to one to force a sequential execution. The capacity of each function is mapped to one and the duration of each function is mapped to the duration of the corresponding transition of [CGK 97]. All other constants besides those below are mapped to themselves.

```
IMPL(K_max) == 1
IMPL(K_g) == 1 for all functions g
IMPL(Dur_g) == duration of transition corresponding to function g
```

A phone P being served in the upper level corresponds to the time between serve_dur after the start of some service transition and just before the end of that transition. All other variables besides serving are mapped to themselves.

```
IMPL(serving(P)) ==
        EXISTS tr: transition, t: time
            (   Start(tr(P), t)
            &   t + serve_dur ≤ now
            &   now < t + Duration(tr))
```

A start of Begin_Serve in the upper level occurs if and only if there is a start of a transition in the lower level at the same time. An end of Begin_Serve occurs if and only if there was a start of a transition serve_dur time units earlier.

```
IMPL(Start(Begin_Serve, now)) ==
        EXISTS tr: transition (Start(tr, now))
IMPL(End(Begin_Serve, now)) ==
        now ≥ serve_dur
    &   EXISTS tr: transition (Start(tr, now - serve_dur))
```

A start of Complete_Serve in the upper level occurs if and only if there was a start of a transition in the lower level at a time Duration(tr) - serve_dur time units earlier. An end of Complete_Serve occurs if and only if there is an end of a transition at the same time.

```
IMPL(Start(Complete_Serve, now)) ==
        EXISTS tr: transition
            (   now ≥ Duration(tr) - serve_dur
            &   Start(tr, now - Duration(tr) + serve_dur))
IMPL(End(Complete_Serve, now)) ==
        EXISTS tr: transition (End(tr, now))
```

*7.8.2.2. Proof of Sequential Refinement*

The most interesting proof obligations in the sequential refinement of the top level central control are the impl_trans_entry and impl_trans_exit obligations. In these proof obligations, let W_g, serving_g, and serving_all refer to IMPL(W_g), IMPL(serving_g), and IMPL(serving_all), respectively.

*7.8.2.2.1. Impl_trans_entry Obligation*

In the Begin_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
    (past
        (   EXISTS tr: transition (Start(tr, now))
    →       EXISTS S: nonempty_set_of_phone
                (   now MOD (2 * serve_dur) = 0
                &   EXISTS S_GDT, S_PD, ..., S_TC: set_of_phone
                        (   S_GDT CONTAINED_IN W_GDT
                        &   S_GDT SET_DIFF serving_all = S_GDT
                        &   set_size(S_GDT UNION serving_GDT) ≤ 1
                        &   ...
                        &   S = S_GDT UNION S_PD UNION ... UNION S_TC
                        &   set_size(S UNION serving_all) ≤ 1
                        &   (   set_size(S UNION serving_all) < 1
                        →       FORALL g: central function
                                    (   set_size(S_g UNION serving_g) = 1
                                    |   W_g CONTAINED_IN S_g UNION serving_all)))), t1))
```

By the antecedent, there is some transition tr_g that starts at time t1. Let P be the phone that tr_g is servicing. The existential clause of the consequent is satisfied by the set consisting of only P. By previous assumption, the transitions of [CGK 97] can only start at times that are multiples of 2 * serve_dur, thus the first conjunct of the consequent holds.

The implementation of serving only holds when a transition is in the middle of execution and serve_dur has elapsed since the transition fired. By trans_mutex, there can only be one such transition. The only transition in the middle of execution is tr_g and at t1, serve_dur time has not yet elapsed. Therefore, set_size(serving_all) = 0. The second conjunct of the consequent is satisfied by the collection of sets S_h, where S_h contains only P for h = g and is empty otherwise by the entry assertion of tr_g.

In the Complete_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
  (past
    (  EXISTS tr: transition
         (  now ≥ Duration(tr) - serve_dur
         &  Start(tr, now - Duration(tr) + serve_dur))
→     (  now MOD (2 * serve_dur) = serve_dur
      &  EXISTS P: phone, tr1: transition
           (  P ISIN serving_g
           &  now - change(
                        EXISTS tr: transition, t: time
                          (  Start(tr(P), t)
                          &  t + serve_dur ≤ now
                          &  now < t + Duration(tr))) + serve_dur ≥
                                  Duration(tr1) - serve_dur)), t1))
```

By previous assumption, transitions only start at times that are multiples of 2 * serve_dur and have durations that are multiples of 2 * serve_dur, thus t1 - Duration(tr) is a multiple of 2 * serve_dur and t1 - Duration(tr) + serve_dur MOD (2 * serve_dur) = serve_dur. Therefore, the first conjunct holds.

Let tr_g be the transition that fires at t1 - Duration(tr_g) + serve_dur. Let P be the phone that tr_g is servicing. At t1, tr_g has not yet ended and a serve_dur has elapsed since tr_g began, thus the first part of the existential clause holds.

The implementation of serving changes whenever serve_dur has elapsed since the start of a transition or at the end of a transition. Since tr_g is still executing, the last change is at the start time of tr_g + serve_dur or t1 - Duration(tr_g) + 2 * serve_dur. Thus, t1 - (t1 - Duration(tr_g) + 2 * serve_dur) + serve_dur ≥ Duration(tr_g) - serve_dur since Duration(tr_g) - serve_dur ≥ Duration(tr_g) - serve_dur. Thus, the second part of the existential clause holds.

*7.8.2.2.2. Impl_trans_exit Obligation*

In the Begin_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
  (past
     (   now ≥ serve_dur
     &   EXISTS tr: transition (Start(tr, now - serve_dur))
  →      EXISTS S: nonempty_set_of_phone
           (FORALL P: phone
              (IF P ISIN S
              THEN
                    EXISTS tr: transition, t: time
                       (   Start(tr(P), t)
                       &   t + serve_dur ≤ now
                       &   now < t + Duration(tr))
              ELSE
                    EXISTS tr: transition, t: time
                       (   Start(tr(P), t)
                       &   t + serve_dur ≤ now
                       &   now < t + Duration(tr))
                 ↔ EXISTS tr: transition, t: time
                       (   past(Start(tr(P), t), now - serve_dur)
                       &   t + serve_dur ≤ now - serve_dur
                       &   now - serve_dur < t + Duration(tr))
              FI)), t1))
```

By the antecedent, there is some transition tr_g that starts at time t1 - serve_dur. Let P be the phone that tr_g is servicing. The existential clause of the consequent is satisfied by the set consisting of only P. Only one phone can satisfy the setdef predicate in the consequent. P satisfies the predicate for transition tr_g and time t1 - serve_dur because Start(tr_g(P), t1 - serve_dur) from the antecedent, t1 - serve_dur + serve_dur ≤ t1, and t1 < t1 - serve_dur + Duration(tr_g) since Duration(tr_g) must be a multiple of 2 * serve_dur.

In the Complete_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
  (past
     (   EXISTS tr: transition (End(tr, now))
  →      FORALL P: phone, g: central function
           (IF     P ISIN past(serving_g, now - serve_dur)
              &   now - past(change(
                          EXISTS tr: transition, t: time
                             (   Start(tr(P), t)
                             &   t + serve_dur ≤ now
                             &   now < t + Duration(tr))), now - serve_dur) ≥
                                    Dur_g - serve_dur
```

```
THEN
        Exit_g(P)
    &   ~EXISTS tr: transition, t: time
            (   Start(tr(P), t)
            &   t + serve_dur ≤ now
            &   now < t + Duration(tr))
ELSE
        EXISTS tr: transition, t: time
            (   Start(tr(P), t)
            &   t + serve_dur ≤ now
            &   now < t + Duration(tr))
    ↔   EXISTS tr: transition, t: time
            (   past(Start(tr(P), t), now - serve_dur)
            &   t + serve_dur ≤ now - serve_dur
            &   now - serve_dur < t + Duration(tr))
FI), t1))
```

Let tr_g be the transition that ends at t1 for a phone P. There can only be one phone and transition for which the if condition is satisfied, since only one transition can be in the middle of execution at any given time. The if condition is satisfied for phone P and function g of tr_g.

At t1 - serve_dur, the last change of the implementation of serving is at t1 - Duration(tr_g) + serve_dur, so the first part of the condition holds since past(W_g, t1 - Duration(tr_g) + serve_dur - serve_dur) holds by trans_entry. The second part of the condition holds since t1 - (t1 - Duration(tr_g) + serve_dur) ≥ Duration(tr_g) - serve_dur.

P is the only phone for which the then branch must hold. The exit of tr_g holds for P by trans_exit. Since tr_g ends at t1, the implementation of serving no longer holds at t1, thus the then branch holds. For all other phones, the else branch must hold. Both existential clauses are false because no phone other than P was being serviced at t1 - serve_dur and no phone can be serve_dur into its execution at t1 since tr_g just ended at t1. Thus, the then branch holds.

### 7.8.3. Parallel Refinement of Top Level Central Control

In the parallel refinement of the top level central control, the central control is refined into several parallel processes, each of which is devoted to a single function g of the top level central control. Each one of these processes executes two transitions that correspond to Begin_Serve and Complete_Serve at the top level.

The main issue in this step is mapping the global state of the central control into disjoint components to be assigned to the different lower level parallel processes. Figure 7.8.3-1 shows the relationship

between the functions and the variables of the central control. A function connected to a variable indicates that the exit assertion of the function sets the variable.

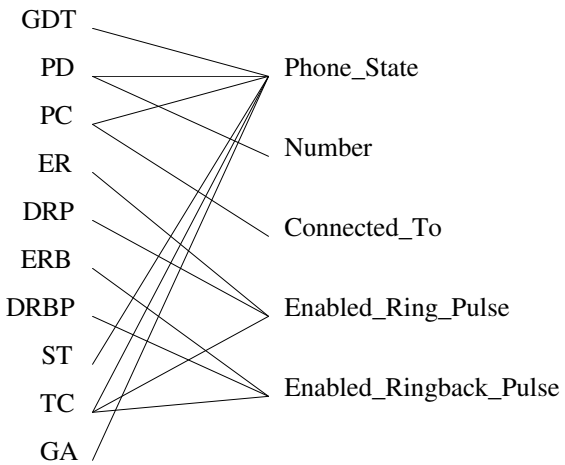



Figure 7.8.3-1: Function and variable relationship in the central control

Note that the exit assertions of the GDT and PD functions have been modified to allow Number to be exclusive to the PD function. The Number(P) reference was removed from the GDT exit assertion and the exit assertion of PD was changed to the following.

```
IF Phone_State′(P) = Ready_To_Dial
THEN
        Number(P) BECOMES LISTDEF(P.Next_Digit′)
    &   Phone_State(P) BECOMES Dialing
ELSE
        Number(P) BECOMES Number′(P) CONCAT LISTDEF(P.Next_Digit′)
FI
```

The most critical variable of the central control is the Phone_State variable, which is set by 6 of the 10 functions. In the following discussion of the parallel refinement of the top level central control, only the implementation of the Phone_State variable and its corresponding type Enabled_State will be described. The other variables can be mapped in a similar fashion.

The type Enabled_State describes all states through which the managing of a call passes during the evolution of the call itself. Such states build a sort of “chain” and the various state transformations move the state of each phone call through it from Idle to Ready_To_Dial, etc. In the implementation of the top level, each step of this path is executed by a different process. This causes a difficulty since all processes must have disjoint states and shared variables (in writing) are not allowed. A possible systematic implementation schema (perhaps not optimal) to split the original state space into disjoint

components is to map the type Enabled_State into a structure of time fields, where each enumerated constant in Enabled_State has a field in the structure as shown below.

```
IMPL(Enabled_State) ==
    STRUCTURE OF (Idle, Ready_To_Dial, Ringing, ...: time)
```

The basic idea of this mapping is that each field of the structure is a timestamp and the field with the most recent timestamp determines the value of variables of the structure type. Since Enabled_State is not compatible with IMPL(Enabled_State), a mapping for constants must be defined as shown below.

```
IMPL(v_es: Enabled_State) ==
    CASE v_es OF
        Idle:
            choose i_v_es: IMPL(Enabled_State)
                (   FORALL f: field (i_v_es[f] = 0)
                |   FORALL f: field
                        (f ≠ Idle → i_v_es[Idle] > i_v_es[f]))
        Ready_To_Dial:
            choose i_v_es: IMPL(Enabled_State)
                (   FORALL f: field
                        (f ≠ Ready_To_Dial → i_v_es[Ready_To_Dial] > i_v_es[f]))
        Ringing:
            choose i_v_es: IMPL(Enabled_State)
                (   FORALL f: field
                        (f ≠ Ringing → i_v_es[Ringing] > i_v_es[f]))
        ...
    ESAC
```

This mapping states that a constant that is Idle in the upper level maps to a structure of type IMPL(Enabled_State) such that all the fields are zero or the Idle field is greater than all the other fields. For values v other than Idle, a constant maps to a structure such that the field associated with v is greater than all the other fields.

In addition to a mapping for constants of type Enabled_State, a mapping must also be defined for the operators with operands of type Enabled_State. The only operator used on operands of type Enabled_State is the = operator. The mapping for the = operator is shown below.

```
IMPL(=(v_es1, v_es2: Enabled_State): bool) ==
    (   (   FORALL f1: field (IMPL(v_es1)[f1] = 0)
    →       FORALL f1: field
                (   IMPL(v_es2)[f1] = 0
                |   (f1 ≠ Idle → IMPL(v_es2)[Idle] > IMPL(v_es2)[f1])))
    |   (   FORALL f1: field (IMPL(v_es2)[f1] = 0)
    →       FORALL f1: field
                (   IMPL(v_es1)[f1] = 0
                |   (f1 ≠ Idle → IMPL(v_es1)[Idle] > IMPL(v_es1)[f1])))
    |   (   EXISTS f1: field
                (   FORALL f2: field
```

```
    (   f1 ≠ f2
→       (   IMPL(v_es1)[f1] > IMPL(v_es1)[f2]
        &   IMPL(v_es2)[f1] > IMPL(v_es2)[f2])))))
```

This mapping states that two constants of type Enabled_State in the upper level are equal if and only if either (1) all the fields in the structure generated from the implementation of one of the constants are zero and the other structure is either all zeroes or the Idle field is greater than all the other fields or (2) there is a field that is greater than all the other fields in both structures in the lower level.

For the implementation of Phone_State, each server has a variable "f(phone): time", for each field f of the IMPL(Enabled_State) structure that the server is responsible for. The mapping for Phone_State is shown below.

```
IMPL(Phone_State(P)) ==
        choose v_es: IMPL(Enabled_State)
            (   v_es[Idle] = TC.Idle(P)
            &   v_es[Ready_To_Dial] = GDT.Ready_To_Dial(P)
            &   ...)
```

This mapping specifies that the state of a phone P is determined by the server that has most recently timestamped a field of P. Thus, it is possible for all the servers to directly affect the state of a phone. Figure 7.8.3-2 shows the mapping from the servers of the lower level to the values of Enabled_State. Note that the value "Calling" is not mapped to any server because Calling is only used for long distance calls. For this mapping to work, it must be guaranteed that no two servers ever give the same timestamp to the same phone. This is a problem, for example, if a phone is offhook and a "slow" server begins to serve the phone and then while this is occurring, the user of the phone hangs up, and the TC server attempts to set the phone to Idle.

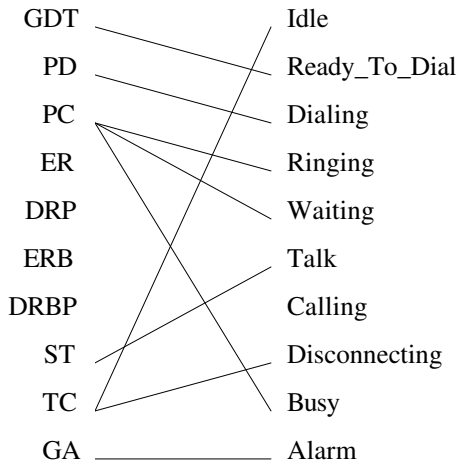



Figure 7.8.3-2: Mapping from servers to Enabled_State values

This problem can be avoided by keeping the Begin_Serve/Complete_Serve mechanism of the top level. Each server will only attempt to serve a phone if no other server is serving that phone. No two servers will ever be able to execute Begin_Serve at the same time for the same phone because at any given time, there is a unique function that a phone needs next. Since discrete time is assumed, it can also be guaranteed that Begin_Serve's cannot overlap on different servers. Thus, no two servers can ever give the same timestamp to the same phone.

For the implementation of serving, each server has a variable "serving_set: set_of_phone". Instead of storing the value of serving for every phone, a server only needs to store those phones that it is currently serving. The mapping for serving is shown below.

IMPL(serving(P)) == EXISTS SP: server (P ISIN SP.serving_set)

The value of K_max in the upper level is intended to limit the amount of parallelism in the system for sequential implementations. In the parallel refinement, it is undesirable to place any such limitation on the number of phones that can be served system wide. Thus, the mapping for K_max is the maximum allowable parallelism sum_K, where sum_K = $\sum_g K\_W\_g$.

IMPL(K_max) == sum_K

All other constants map to themselves.

In the top level, Begin_Serve is enabled when there exists a set of phones S that can be partitioned into 10 disjoint subsets such that there is a subset S_g for each function g that is limited in size by K_W_g - set_size(serving_g) and must contain only the phones that begin being served. In the second level, each server has a Begin_Serve transition that is enabled when there exists a set of phones that corresponds to the disjoint subset S_g for the function g that the server performs. The exit assertion of Begin_Serve on a server SP_g specifies that the set of phones that begin being served by SP_g is equal to the disjoint subset S_g. The definition of Begin_Serve for the PD server is shown below.

```
Begin_Serve(S: nonempty_set_of_phone)
   ENTRY       [TIME:    serve_dur]
          now MOD (2 * serve_dur) = 0
      &   S CONTAINED_IN W_PD
      &   S SET_DIFF serving_all = S
      &   set_size(S UNION serving_set) ≤ K_W_PD
      &   (    set_size(S UNION serving_set) < K_W_PD
      →       W_PD CONTAINED_IN S UNION serving_all)
   EXIT
          serving_set = serving_set′ UNION S
```

A start of Begin_Serve in the upper level corresponds to a start of Begin_Serve on some server in the lower level. The mapping for an end of Begin_Serve is similarly defined.

```
IMPL(Start(Begin_Serve, now)) ==
    EXISTS SP: server
        (SP.Start(Begin_Serve, now))

IMPL(End(Begin_Serve, now)) ==
    EXISTS SP: server
        (SP.End(Begin_Serve, now))
```

The definition of Complete_Serve for servers in the lower level is similar to the definition of Complete_Serve in the upper level except that each server SP_g only checks the phones it is serving when determining which phones were being served for Dur_g and changes the state of these phones according to Exit_g. The definition of Complete_Serve for the PD server is shown below.

```
Complete_Serve
    ENTRY       [TIME:    serve_dur]
            now MOD (2 * serve_dur) = serve_dur
        &   EXISTS P: phone
                (   P ISIN serving_set
                &   now - change(P ISIN serving_set) + serve_dur ≥ Dur_PD - serve_dur)
    EXIT
            FORALL P: phone
                (IF     P ISIN serving_set′
                    &   now - past(change(P ISIN serving_set), now - serve_dur) ≥
                            Dur_PD - serve_dur
                    THEN
                            P ~ISIN serving_set
                        &   IF FORALL SP: server
                                    (   SP ≠ GDT
                                    →       GDT.Ready_To_Dial(P) > SP.ES(P))
                            THEN
                                    Number(P) BECOMES LISTDEF(P.Next_Digit′)
                                &   Dialing(P) = now
                            ELSE
                                    Number(P) BECOMES
                                            Number′(P) CONCAT LISTDEF(P.Next_Digit′)
                            FI
                    ELSE
                            P ISIN serving_set′ ↔ P ISIN serving_set
                    FI)
```

When the state of a phone P was previously Ready_To_Dial, the new state of P is set to Dialing, since the timestamp of Dialing(P) is set to the current time by the expression “Dialing(P) = now”. Note that the expression SP.ES(P) is a convenience notation to refer to the timestamp of phone P for any variables that are components of the Phone_State mapping in the server SP (e.g. TC.Idle(P)).

A start of Complete_Serve in the upper level corresponds to a start of Complete_Serve on some server in the lower level. The mapping for an end of Complete_Serve is similarly defined.

```
IMPL(Start(Complete_Serve, now)) ==          IMPL(End(Complete_Serve, now)) ==
      EXISTS SP: server                            EXISTS SP: server
         (SP.Start(Complete_Serve, now))              (SP.End(Complete_Serve, now))
```

### 7.8.4. Proof of Parallel Refinement of Top Level Central Control

The proof obligations that will be shown for the parallel refinement of the top level of the central control are the impl_trans_entry obligation for Begin_Serve, the impl_trans_exit obligation for Complete_Serve, and the impl_trans_fire obligation. In these proof obligations, let W_g, serving_g, and serving_all refer to IMPL(W_g), IMPL(serving_g), and IMPL(serving_all), respectively.

*7.8.4.1. Impl_trans_entry Obligation for Begin_Serve*

```
FORALL t1: time
   (past
      (   EXISTS SP: server (SP.Start(Begin_Serve, now))
   →      EXISTS S: nonempty_set_of_phone
             (   now MOD (2 * serve_dur) = 0
             &   EXISTS S_GDT, S_PD, ..., S_TC: set_of_phone
                    (   S_GDT CONTAINED_IN W_GDT
                    &   S_GDT SET_DIFF serving_all = S_GDT
                    &   set_size(S_GDT UNION GDT.serving_set) ≤ K_W_GDT
                    &   ...
                    &   S = S_GDT UNION S_PD UNION ... UNION S_TC
                    &   set_size(S UNION serving_all) ≤ sum_K
                    &   (   set_size(S UNION serving_all) < sum_K
                    →       FORALL g: central function
                               (   set_size(S_g UNION SP_g.serving_set) = K_W_g
                               |   W_g CONTAINED_IN S_g UNION serving_all)))), t1))
```

Begin_Serve can only be enabled at times that are multiples of 2 * serve_dur. Complete_Serve can only be enabled at times that are serve_dur plus a multiple of 2 * serve_dur. Begin_Serve and Complete_Serve each have duration serve_dur, thus any time Begin_Serve fires on some server, every other server will either be idle or will fire Begin_Serve. At t1, Begin_Serve fires on some server, thus every other server is either idle or fires Begin_Serve. For each server SP_g on which Begin_Serve fires at t1, let S2_g be the set parameter for which Begin_Serve fired. The consequent is satisfied by the union of all sets S2_g.

The first part of the consequent is satisfied by the entry of Begin_Serve on a server on which it started. The second part is satisfied by S_g = S2_g for the servers SP_g that Begin_Serve started on at t1 and by S_g = ∅ for the other servers. By the entry of Begin_Serve, S2_g CONTAINED_IN W_g, S2_g SET_DIFF serving_all = S2_g, and set_size(S_g UNION SP_g.serving_set) ≤ K_W_g.

The empty set trivially satisfies these constraints. By the definition of S, S = UNION S2_g = UNION S2_g UNION EMPTY. Each S_g UNION serving_g must be ≤ K_W_g by the entry of Begin_Serve, so set_size(S UNION serving_all) must be ≤ sum_K. Suppose set_size(S_g UNION serving_g) < K_W_g & W_g ~CONTAINED_IN S UNION serving_all for some g. Let P be a phone that needs service from g, but is not in S_g UNION serving_g. Suppose SP_g starts Begin_Serve at t1. In this case, S2_g does not satisfy the entry assertion of Begin_Serve because set_size(S2_g) < K_W_g and yet P is in W_g, but not being served by any other server. Therefore, ~SP_g.Start(Begin_Serve(S2_g), t1) holds, which is a contradiction. Suppose SP_g did not start Begin_Serve at t1. In this case, the entry assertion holds because P needs service and SP_g still has capacity left. SP_g must be idle because Begin_Serve and Complete_Serve are mutually exclusive by their entry assertions. Thus, SP_g.Start(Begin_Serve, t1) holds, which is also a contradiction.

*7.8.4.2. Impl_trans_exit Obligation for Complete_Serve*

```
FORALL t1: time
   (past
      (   EXISTS SP: server
             (SP.End(Complete_Serve, now))
   →      FORALL P: phone, g: central function
             (IF     P ISIN past(SP_g.serving_set, now - serve_dur)
                 &   now - past(change(EXISTS SP: server (P ISIN SP.serving_set)),
                                now - serve_dur) ≥ Dur_g - serve_dur
             THEN
                     Exit_g(P)
                 &   ~EXISTS SP: server (P ISIN SP.serving_set)
             ELSE
                     EXISTS SP: server (P ISIN SP.serving_set)
                 ↔   past(EXISTS SP: server (P ISIN SP.serving_set), now - serve_dur)
             FI), t1))
```

Suppose there is some phone P and function g such that the if condition holds, but the then branch does not hold. Complete_Serve must be enabled on SP_g at t1 - serve_dur because now MOD (2 * serve_dur) = serve_dur by the entry assertion of Complete_Serve on a server on which it ended at t1, and P ISIN SP_g.serving_set at t1 - serve_dur and t1 - serve_dur - change(P ISIN serving_set) + serve_dur ≥ Dur_PD - serve_dur by the if condition. SP_g must be idle because Begin_Serve and Complete_Serve are mutually exclusive by their entry assertions. By trans_fire, Complete_Serve starts at t1 - serve_dur on SP_g, thus its exit assertion holds at t1, so the then branch holds.

For the else branch, suppose a phone P and central function g do not satisfy the if condition, but the status of P in SP_g.serving_set changes at t1. Begin_Serve and Complete_Serve are mutually

exclusive, thus Complete_Serve on SP_g changes the status. Complete_Serve on SP_g can only change the status, however, when the if condition holds for P, which is a contradiction.

*7.8.4.3. Impl_trans_fire Obligation*

In the Begin_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
   (  past
         (EXISTS S: nonempty_set_of_phone
            (  now MOD (2 * serve_dur) = 0
            &  EXISTS S_GDT, S_PD, ..., S_TC: set_of_phone
                  (  S_GDT CONTAINED_IN W_GDT
                  &  S_GDT SET_DIFF serving_all = S_GDT
                  &  set_size(S_GDT UNION GDT.serving_set) ≤ K_W_GDT
                  &  ...
                  &  S = S_GDT UNION S_PD UNION ... UNION S_TC
                  &  set_size(S UNION serving_all) ≤ sum_K
                  &  (  set_size(S UNION serving_all) < sum_K
                  →     FORALL g: central function
                           (  set_size(S_g UNION SP_g.serving_set) = K_W_g
                           |  W_g CONTAINED_IN S_g UNION serving_all)))), t1)
   &  FORALL t2: time
         (  t1 - serve_dur < t2 & t2 < t1
      →     ~EXISTS SP: server
               (past(SP.Start(Begin_Serve, t2), t2))
         &  ~EXISTS SP: server
               (past(SP.Start(Complete_Serve, t2), t2)))
→     EXISTS SP: server
         (past(SP.Start(Begin_Serve, t1))))
```

Let S be a set of phones satisfying the existential clause in the antecedent. Let S_g be a nonempty set of the second part of the existential clause. There must be such a set since S is nonempty and S is the union of all such sets. The entry assertion of Begin_Serve is satisfied by the set S_g on SP_g at t1. By the antecedent, no server is executing any transition at t1, thus Begin_Serve will fire on SP_g at t1 by trans_fire.

In the Complete_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
   (  past
         (  now MOD (2 * serve_dur) = serve_dur
         &  EXISTS P: phone, g: central function
               (  P ISIN SP_g.serving_set
               &  now - change(EXISTS SP: server (P ISIN SP.serving_set)) +
                     serve_dur ≥ Dur_g - serve_dur), t1)
   &  FORALL t2: time
         (  t1 - serve_dur < t2 & t2 < t1
      →     ~EXISTS SP: server
```

```
                (past(SP.Start(Begin_Serve, t2), t2))
        &   ~EXISTS SP: server
                (past(SP.Start(Complete_Serve, t2), t2)))
→       EXISTS SP: server
            (past(SP.Start(Complete_Serve, t1))))
```

Let P and g be the phones satisfying the existential clause in the antecedent. Thus, P is in the serving set of SP_g at t1. Also, P has been being served for Dur_g - 2 * serve_dur. Thus, the entry assertion of Complete_Serve on SP_g holds at t1. By the antecedent, no server is executing any transition at t1, thus Complete_Serve will fire on SP_g at t1 by trans_fire.

### 7.8.5. Parallel Refinement of Second Level Process Call Server

This section discusses the parallel refinement of the second level process call server. The other servers of the second level central control can be refined in a similar manner. The PC server is implemented by a parallel array of K_W_PC *microservers*, where each microserver is devoted to processing the calls of a single phone. Each microserver picks a phone from W_PC according to some possibly nondeterministic policy and inserts its identifier into a set of served phones through a sequence of two transitions. The union of the elements of such sets over all the PC microservers implements the serving set of the upper level PC server.

At this refinement level, it is not possible to statically allocate the individual phone timestamps of Ringing, Waiting, and Busy to different microservers or else there would be no way for phones allocated to the same microserver to be serviced at the same time, which is possible at the higher levels. Instead, microservers dynamically hold the state of the set of phones that were last serviced on that microserver. To control the size of the served set, a microserver removes a set of phones from the set such that each phone is in the set if the timestamp for that phone on some other macroserver has changed more recently than the phone was added to the served set.

The PC microservers process phones in pairs, where each pair is of the type “phone_pair: STRUCTURE OF (Waiting: phone, Ringing: phone)”. The variable “serving: boolean” specifies if the microserver is currently serving a pair of phones. The variable “serving_pair: phone_pair” specifies the pair of phones the microserver is going to connect. Finally, the variable “served_set: list of phone_pair” specifies the set of phone pairs whose calls have been processed by the microserver, but whose state has not yet been changed by any of the other macroservers.

The timestamp that a phone P became waiting is the time that P became the waiting phone of a phone pair in the served_set of some microserver. The waiting timestamp of P is zero if no such phone pair exists.

```
IMPL(Waiting(P)) ==
        IF      EXISTS MSP: microserver, PP: phone_pair
                    (   PP ISIN MSP.served_set
                    &   PP[Waiting] = P
                    &   PP[Ringing] ≠ P)
        THEN
                change(EXISTS MSP: microserver, PP: phone_pair
                            (   PP ISIN MSP.served_set
                            &   PP[Waiting] = P
                            &   PP[Ringing] ≠ P))
        ELSE
                0
        FI
```

The timestamp that a phone P became ringing is the time that P became the ringing phone of a phone pair in the served_set of some microserver. The ringing timestamp of P is zero if no such phone pair exists.

```
IMPL(Ringing(P)) ==
        IF      EXISTS MSP: microserver, PP: phone_pair
                    (   PP ISIN MSP.served_set
                    &   PP[Ringing] = P
                    &   PP[Waiting] ≠ P)
        THEN
                change(EXISTS MSP: microserver, PP: phone_pair
                            (   PP ISIN MSP.served_set
                            &   PP[Ringing] = P
                            &   PP[Waiting] ≠ P))
        ELSE
                0
        FI
```

The timestamp that a phone P became busy is the time that P became both the waiting phone and the ringing phone of a phone pair in the served_set of some microserver. The busy timestamp of P is zero if no such phone pair exists.

```
IMPL(Busy(P)) ==
        IF      EXISTS MSP: microserver, PP: phone_pair
                    (   PP ISIN MSP.served_set
                    &   PP[Waiting] = P
                    &   PP[Ringing] = P)
        THEN
                change(EXISTS MSP: microserver, PP: phone_pair
                            (   PP ISIN MSP.served_set
                            &   PP[Waiting] = P
                            &   PP[Ringing] = P))
        ELSE
                0
        FI
```

A phone is connected to a phone P if it is in a phone pair with P on some microserver. Connected_To is set to P otherwise.

```
IMPL(Connected_To(P)) ==
        IF      EXISTS P2: phone
                    (   P2 ≠ P
                    &   EXISTS MSP: microserver, PP: phone_pair
                            (   PP ISIN MSP.served_set
                            &   (   PP[Waiting] = P & PP[Ringing] = P2
                                |   PP[Waiting] = P2 & PP[Ringing] = P)))
        THEN
                choose P2: phone
                    (   P2 ≠ P
                    &   EXISTS MSP: microserver, PP: phone_pair
                            (   PP ISIN MSP.served_set
                            &   (   PP[Waiting] = P & PP[Ringing] = P2
                                |   PP[Waiting] = P2 & PP[Ringing] = P)))
        ELSE
                P
        FI
```

The implementation of the serving set is the set of phones that are the waiting phone in the serving pair of some microserver that is serving.

```
IMPL(serving_set) ==
        setdef P: phone
            (EXISTS MSP: microserver
                (   MSP.serving
                &   MSP.serving_pair[Waiting] = P))
```

Each PC server has two transitions, which correspond to Begin_Serve and Complete_Serve in the upper level. Begin_Serve finds a pair of phones in W_PC to be connected. Complete_Serve commits the connection between the two phones identified in the preparation phase and resets the state of all phones in its list Served_Phones that have been serviced more recently by some other upper level macroserver. The definitions of Begin_Serve and Complete_Serve are given below.

```
Begin_Serve(P: phone)
    ENTRY       [TIME:    serve_dur]
            now MOD (2 * serve_dur) = 0
        &   ~serving
        &   P ISIN W_PD
        &   P ~ISIN serving_all
        &   set_size(setdef P2: phone (P2 ISIN W_PD & P2 ~ISIN serving_all & P2 < P))
                =   set_size(setdef MSP: microserver (~MSP.serving & MSP < self))
    EXIT
            serving
        &   serving_pair = choose PP: phone_pair
                                (   PP[Waiting] = P
                                &   PP[Ringing] =
```

```
IF      GET_ID(PD.Number(P)).Offhook
   |    EXISTS SP: server
           (    SP ≠ TC
        →       SP.ES(GET_ID(PD.Number(P))) >
                        TC.Idle(GET_ID(PD.Number(P))))
   |    EXISTS g: central function
           (GET_ID(PD.Number(P)) ISIN W_g)
THEN
        P
ELSE
        GET_ID(PD.Number(P))
FI)
```

The last conjunct of the entry assertion states that if there are n phones satisfying the condition whose IDs are less than P's ID, then there exist n microservers whose IDs are less than self and are available. This is a simple trick to state that the available microservers are allocated in order of increasing ID number to phones that need their service. In this way, conflicts are avoided and it is possible to easily prove requirements on the number of phone calls that will be served. A start of Begin_Serve in the upper level corresponds to a start of Begin_Serve on some microserver in the lower level. The mapping for an end of Begin_Serve is similarly defined.

```
IMPL(Start(Begin_Serve, now)) ==
        EXISTS MSP: microserver
           (MSP.Start(Begin_Serve, now))

IMPL(End(Begin_Serve, now)) ==
        EXISTS MSP: microserver
           (MSP.End(Begin_Serve, now))
```

Complete_Serve finishes serving the phones in serving_pair.

```
Complete_Serve
   ENTRY       [TIME:    Dur_PC - serve_dur]
           serving
   EXIT
           FORALL PP: phone_pair
              (IF     PP ISIN served_set′
                 &    EXISTS SP: server
                         (SP.ES(PP[Waiting]) >
                            past(change(PP ISIN served_set),
                                    now - Dur_PC + serve_dur)
                 &    EXISTS SP: server
                         (SP.ES(PP[Ringing]) >
                            past(change(PP ISIN served_set),
                                    now - Dur_PC + serve_dur)
              THEN
                      PP ~ISIN served_set
              ELSE
                      PP ISIN served_set′ ↔ PP ISIN served_set
              FI)
```

Notice that the duration of Complete_Serve is now Dur_PC - serve_dur, which is the time it takes to complete processing a call, whereas in the higher levels, the duration was a small duration,

serve_dur, so that phones could complete being serviced at almost any time. Also note that it should be possible to bound the number of phones that any microserver actually has to delete from its serving list in any execution of Complete_Serve as well as the size of the served set based on Dur_g and K_W_g of all macroservers g.

### 7.8.6. Proof of Parallel Refinement of Process Call Server

The proof obligations that will be shown for the parallel refinement of the process call server are the impl_trans_mutex and impl_vars_no_change obligations.

#### *7.8.6.1. Impl_trans_mutex Obligation*

In the Begin_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
    (   past(EXISTS MSP: microserver (MSP.Start(Begin_Serve, t1)), t1)
→       ~past(EXISTS MSP: microserver (MSP.Start(Complete_Serve, t1)), t1)
    &   FORALL t2: time
            (   t1 < t2 & t2 < t1 + serve_dur
            →       ~past(EXISTS MSP: microserver (MSP.Start(Begin_Serve, t2)), t2))
    &   FORALL t2: time
            (   t1 < t2 & t2 < t1 + serve_dur
            →       ~past(EXISTS MSP: microserver (MSP.Start(Complete_Serve, t2)), t2)))
```

Begin_Serve can only start at times that are multiples of 2 * serve_dur, by its entry assertion. Complete_Serve is enabled when serving holds. Begin_Serve sets serving and Complete_Serve resets serving so Complete_Serve can only start immediately when a Begin_Serve ends. Therefore, Complete_Serve can only start at times that are serve_dur after a multiple of 2 * serve_dur. Since a Begin_Serve starts at t1, Complete_Serve cannot have started on any microserver in the interval (t1 - serve_dur, t1 + serve_dur). Thus, the first and the third conjuncts of the consequent hold. Since Begin_Serve only starts at times that are multiples of 2 * serve_dur and t1 is such a multiple, Begin_Serve cannot start in the interval (t1, t1 + 2 * serve_dur), thus the second conjunct holds.

In the Complete_Serve case, it must be shown that the following formula holds.

```
FORALL t1: time
    (   past(EXISTS MSP: microserver (MSP.Start(Complete_Serve, t1)), t1)
→       ~past(EXISTS MSP: microserver (MSP.Start(Begin_Serve, t1)), t1)
    &   FORALL t2: time
            (   t1 < t2 & t2 < t1 + serve_dur
            →       ~past(EXISTS MSP: microserver (MSP.Start(Begin_Serve, t2)), t2))
    &   FORALL t2: time
            (   t1 < t2 & t2 < t1 + serve_dur
            →       ~past(EXISTS MSP: microserver (MSP.Start(Complete_Serve, t2)), t2)))
```

By previous argument, Begin_Serve can only start at times that are multiples of 2 * serve_dur and Complete_Serve always starts at an end of a Begin_Serve. Since a Complete_Serve starts at t1, Begin_Serve cannot start on any microserver in the interval (t1 - serve_dur, t1 + serve_dur). Thus, the first two conjuncts of the consequent hold. Since Begin_Serve cannot start in the interval (t1 - serve_dur, t1 + serve_dur), Complete_Serve cannot start in the interval (t1, t1 + 2 * serve_dur). Therefore, the third conjunct holds.

*7.8.6.2. Impl_vars_no_change Obligation*

For the impl_vars_no_change obligation, it must be shown that the following formula holds. Note that implementation of Vars_No_Change(t1, t1) is not expanded for brevity.

```
FORALL t1, t3: time
   (   t1 ≤ t3
   &   FORALL t2: time
           (   t1 < t2 & t2 ≤ t3
       →       ~past(EXISTS MSP: microserver
                         (MSP.End(Begin_Serve, t2)), t2))
   &   FORALL t2: time
           (   t1 < t2 & t2 ≤ t3
       →       ~past(EXISTS MSP: microserver
                         (MSP.End(Complete_Serve, t2)), t2))
→      FORALL t2: time
           (   t1 ≤ t2 & t2 ≤ t3
       →       IMPL(Vars_No_Change(t1, t2)))))
```

The only way for the implementations of Waiting, Ringing, Busy, Connected_To, and serving_set to change is if the served_set on some microserver changes. The served_set of a microserver only changes when an end of a Begin_Serve or Complete_Serve occurs on that microserver. By the antecedent, there is no such end in the interval (t1, t3], thus the implementations of the variables do not change value in the interval.

## 7.9. Parallel Refinement Guidelines

During the specification of different systems using the parallel refinement mechanism, several issues were encountered that are common to parallel refinement in general. The techniques used to handle these issues in the different systems can be generalized into a set of guidelines for parallel refinement. These guidelines are discussed below.

### 7.9.1. Asynchronous Concurrency

Each process in a system performs some set of actions during its execution. In the implementation of a process, it may be neither feasible nor desirable for lower level processes to execute these actions in

a lockstep fashion. Instead, lower level processes may need to perform actions dynamically and without synchronization with other lower level processes.

In order to allow such asynchronous concurrency in the refinement of a process, the upper level process needs to be specified appropriately. In particular, concurrent actions in the upper level that may be executed asynchronously in the lower levels should not be specified such that they begin and complete execution in the same transition. For example, in the phone system, the top level could have been specified such that there was a single transition Serve that was executed every t time units in which some set of phones was completely serviced in each execution. This would mean, however, that in the lower levels, phones could only be serviced at the rate of the slowest server and that the servers would process phones in lockstep with each other.

To allow asynchronous concurrency, concurrent actions in the upper level should be specified such that a set of actions can start and a set of actions can end at every time in the system. For example, in the top level of the central control, the service of a phone was split into the beginning of servicing and the completion of servicing in the transitions Begin_Serve and Complete_Serve, respectively. The durations of Begin_Serve and Complete_Serve were set to serve_dur, where 2 * serve_dur was chosen to be a divisor of the duration of every action. In general, it is not necessary to have a separate transition for the beginning and completion of an action. It is necessary, however, to have some record of when an action has started so that it can be completed at the appropriate time. In the central control, changes to the serving variable were used to record this information. When serving changed to true for a phone at time t, that phone began being served at t - serve_dur. Thus, when the duration of the function that was serving the phone elapsed, the effect of the function was carried out on the phone's state and serving for that phone was reset to false.

### 7.9.2. Multiple Writers

In the design of complex systems such as the phone system, there is often a need in the lower levels for multiple processes to control the implementation of a particular upper level variable. In the ASTRAL model, however, only a single process can change the value of a variable, thus it is not possible to let multiple lower level processes change the same variable directly. In the refinement of the central control, the Phone_State variable of the upper level needed to be changed by many of the servers. The solution used in that refinement, which will work in general for any refinement in which multiple writers need to be allowed, was to split the variable into a structure of timestamps, with one timestamp allocated to each process that needs to change the variable.

The Enabled_State type was simple because there were a bounded number of values and each value was the responsibility of a single process. In general, however, the same technique can be used for types with arbitrary values and with an arbitrary number of writers of each value. Consider a variable v of type integer in the upper level. Suppose there are n processes P1 ... Pn that need to change the value of the implementation of v in the lower level to any value. In order to specify this, each process Pi has a variable iv of type "STRUCTURE OF (timestamp: time, value: integer)". Whenever Pi changes the value of iv[value], it sets iv[timestamp] to now.

The mapping for v would then be:

```
IMPL(v) ==
        choose i: integer
            (EXISTS P: Pi
                (   P.iv[value] = i
                &   FORALL P2:proc
                        (P.iv[timestamp] > P2.iv[timestamp])))
```

This states that the value of v is the value of iv[value] of the lower level process that has last changed its iv. Thus, each Pi only changes its own variables and yet the implementation of v can effectively be changed by any Pi.

### 7.9.3. Sequential Implementations

In some cases, such as in the central control, there is the possibility that a process may be implemented in both a sequential and a parallel fashion. In these cases, it is necessary for the upper level specification to allow the possibility of multiple actions occurring at the same time and yet not actually requiring multiple actions to occur.

In the top level of the central control, this was achieved by the K_max constant. The K_max restriction in the entry assertion of Begin_Serve limits the number of phones that can be serviced at any given time in the system. In the sequential refinement, K_max was set to one, indicating that only one phone at a time can be serviced. In the parallel refinement, K_max was set to the sum of the capacities of the individual servers, indicating that as many phones as is possible for the servers to serve can be serviced in parallel.

When there is a nondeterministic choice of actions in the upper level, it is necessary to make the choice of actions as a transition parameter in order to allow a sequential refinement of the process. For example, in the top level of the central control, the choice of phones to begin serving was made at the start of Begin_Serve as the set parameter S. This choice could also have been made by a nondeterministic choose expression in the exit assertion of Begin_Serve. This would not have

allowed for a sequential refinement of the top level, however, because in the sequential refinement of the central control, as soon as any transition begins execution, the phone and the service to be performed on the phone is immediately known. If the choice of phones and services in the top level was made in the exit assertion, it would not have been possible to determine which phone was going to be served until serve_dur after a phone actually started being served in the top level.

### 7.9.4. Range Refinement

Besides allowing a process to be implemented by a set of concurrent processes at the lower level, the parallel refinement mechanism also allows increased flexibility in strictly sequential refinements. For example, the critical requirements of a process may hold for a range of different transition durations in that process. In this case, it is desirable to specify the upper level process in such a way as to allow any particular duration or combination thereof to be chosen at the lower level so that more efficient implementations are not penalized by using a fixed maximum duration.

Consider a process P in which a transition T can have a finite range of durations $d_1 ... d_n$. For simplicity, assume T is not exported. ASTRAL does not allow a range of durations to be specified directly. This effect can be achieved, however, by defining n transitions $T_1$, ..., $T_n$ in the upper level of P that are identical except for their durations, which are $d_1$, ..., $d_n$, respectively. In the lower level, only a subset of the possible duration range of T may be implemented. Using the parallel refinement mechanism, each duration $d_i$ that is not implemented can be eliminated by setting both the start and end mappings for $T_i$ to false (i.e. IMPL(Start($T_i$, now)) == FALSE and IMPL(End($T_i$, now)) == FALSE). In this way, the lower level implementation of the process may operate at strictly the fastest possible speed or at any other speed in the original range.

Given that the start and end mappings are set to false, all of the proof obligations except the impl_trans_fire obligation are trivially satisfied. The only way that impl_trans_fire can be proved for a transition $T_i$ that is eliminated by the mapping is if there is some other transition $T_j$, where $j \neq i$, that is enabled at all times $T_i$ is enabled. Given that $T_i$ is not exported, this means that $T_j$ cannot be exported and that the entry assertion of $T_i$ must imply the entry assertion of $T_j$. Since $T_i$ and $T_j$ are identical except for their durations by definition, this condition holds, thus impl_trans_fire will hold for the $T_i$ case as long as there is at least one $T_j$ that is not eliminated by the mapping.

Note that this technique is not specific to range refinement and can be used in any refinement where there is a need to reduce nondeterminism in the lower level. In the general case, $T_i$ and $T_j$ may not be identical, thus the proof of mutual enablement may be more difficult. Even in the general case,

however, it is not necessary for the exit assertion of $T_j$ to imply the exit assertion of $T_i$. If the exit assertion of $T_i$ was essential to the proof of the upper level critical requirements, then it must have been shown that $T_i$ actually fired in the upper level. Without the transition selection clause, which was not considered in the parallel refinement mechanism of this chapter, the only way to guarantee that a particular transition will fire when it is enabled and the process is idle is if no other transition is enabled at the same time. By proving the impl_trans_fire obligation, however, it is shown that there is always another transition enabled at the same time, thus the exit assertion could not have been crucial to the upper level proof.

# Chapter 8

# Classification Schemes and Querying Mechanisms

In order to achieve the goal of a systematic analysis methodology, it was necessary to develop a set of classification schemes that could be used to differentiate between different specification and proof types. This chapter presents the classification schemes that were developed, which are used throughout the presentation of the systematic analysis methodology in the following chapters. Three classification schemes were developed based on transitions, processes, and properties. Each classification scheme is discussed as well as the heuristics used to recognize each classification type and the querying mechanisms that allow the user to obtain the classification information. In addition, two other querying mechanisms that are not related to classification, but that are crucial to the analysis methodology, are presented.

## 8.1. Transition Classification

In ASTRAL, the enablement of a transition depends on four factors: the local state, the imported state, the external environment, and the current time. The local state includes the values of local variables and the start and end times of local transitions. The imported state includes the values of variables imported from other processes and the start and end times of imported transitions. All transitions depend on one or more of the factors. For example, consider the two transitions of the Olympic boxing scoring system shown below.

```
TRANSITION   End_Round
   ENTRY     [TIME:   End_Dur]
             Now - Start(Begin_Round) ≥
                Round_Length
      &      In_Round
   EXIT
             ~In_Round
```

```
TRANSITION   Update(B: Boxer)
   ENTRY     [TIME:   Update_Dur]
             EXISTS S: Set_Of_Judge_ID
                (   SET_SIZE (S) ≥ 3
                &   FORALL j: Judge_ID
                       (   j ISIN S
                       ↔       Now - Judges[j].Start(
                                  Score(B)) ≤ Window))
      &      Now - Start(Update) ≥ Window
      &      Outcome = In_Progress
   EXIT
             Points(B) BECOMES Points′(B) + 1
```

End_Round depends on the local state (In_Round and Start(Begin_Round)) and the current time (Now). Update additionally depends on other processes in the system (Judges[j].Start(Score(B))).

A transition is classified based on which factors its enablement is dependent on. There are seven classifications corresponding to the possible combinations of the imported state (O), external environment (E), and current time (T) factors plus a classification for transitions that only depend on the local state (L). The End_Round transition is of type T while Update is of type OT. Table 8.1 shows the number of transitions of each classification that appear in the testbed systems. This information can be obtained by the user by selecting a transition in the transition browser window as described in section 8.4 and performing the "transition information" query.

| System | L | E | O | T | EO | ET | OT | EOT | Total |
|---|---|---|---|---|---|---|---|---|---|
| Bakery Algorithm | 4 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 6 |
| Cruise Control | 2 | 9 | 2 | 1 | 0 | 0 | 0 | 0 | 14 |
| Elevator | 0 | 3 | 4 | 3 | 0 | 0 | 2 | 0 | 12 |
| Olympic Boxing | 0 | 0 | 2 | 2 | 0 | 0 | 1 | 1 | 6 |
| Phone | 0 | 2 | 16 | 0 | 7 | 0 | 5 | 0 | 30 |
| Production Cell | 14 | 0 | 11 | 20 | 0 | 0 | 10 | 1 | 56 |
| Railroad Crossing | 0 | 1 | 2 | 3 | 0 | 0 | 0 | 0 | 6 |
| Stoplight | 0 | 2 | 4 | 0 | 0 | 0 | 18 | 0 | 24 |
| Total | 20 | 17 | 42 | 30 | 7 | 0 | 36 | 2 | 154 |

Table 8.1: Transition classifications of testbed systems

These classifications are significant because they can be used to help determine the delay between any two consecutive transitions on the same process instance, which in turn can be used to determine the execution time of arbitrary transition sequences. To determine the delay between two consecutive transitions tr1 and tr2, tr2 is first classified. If tr2 depends only on local variables, tr2 must fire immediately after tr1. Suppose tr2 depends only on local variables, but that tr2 does not fire immediately after tr1. Let t1 be the time that tr1 fired and t2 be the time that tr2 fired. In this case, t2 - t1 > Duration(tr1) and tr2 is not enabled at t1 + Duration(tr1) or else by trans_fire, tr2 would have fired. Since tr2 does eventually fire at t2, however, it is enabled at t2. Since tr2 depends only on the local state, there must have been a change to the local state between t1 + Duration(tr1) and t2. This means that there was either a start, an end, and/or a change to a local variable. Since tr1 and tr2 were assumed to be consecutive, however, no other transition could have started or ended in between and as a consequence no changes to local variables could have occurred. This is a contradiction, so tr2 must fire immediately after tr1.

If tr2 depends on more than just the local state, the delay between tr1 and tr2 is more difficult to determine. If tr2 additionally depends on the current time, the history of the system must be examined to determine if and when the events constrained by the current time have occurred. In many cases, the events will be related to the execution of tr1. For example, a common instance of this type is "now - End(tr1) ≥ delay1". Another common instance is "now - Change(v) ≥ delay1" where v is a variable set by tr1. In these cases, the delay between tr1 and tr2 is equal to delay1. If the events are not related to tr1, however, more in-depth analysis of the history of the system must be performed.

If tr2 depends on the environment, the start of tr2 is delayed until a call to tr2 is generated by the external environment. In this case, it is necessary to examine the environment clause for restrictions on calls to tr2. If there are no such restrictions, then the delay between tr1 and tr2 can be arbitrarily large. If there are restrictions, however, then it is necessary to examine the history of the system to determine when calls can or will occur to compute the delay between tr1 and tr2. Similarly, if tr2 depends on other processes, the start of tr2 is delayed until imported variables have the appropriate values and/or imported transitions have occurred at the appropriate times. In this case, the imported variable clause must be examined in a similar fashion to that of the environment clause. If tr2 depends on combinations of the current time, the environment, and other processes, all of the appropriate clauses must be examined.

## 8.2. Process Classification

The goal of process classification is to identify detectable differences in process behavior that lead to significantly different styles of proofs. A number of classification schemes for parallel processes have been devised in other work. Several of these existing schemes were considered to classify process types. These include both behavioral classifications and structural classifications. Behavioral classifications are based on the actual activities of the system while structural classifications are based on system component types and the types of connections between components.

[And 91] describes a taxonomy based on process interactions. The classifications include filters, clients and servers, heartbeat algorithms, probe/echo algorithms, broadcast algorithms, token-passing algorithms, and replicated servers. These classifications can be used in divide and conquer strategies during proofs. For example, if multiple processes are involved in a token-passing algorithm, the proof of a timing requirement can be broken down into finding the maximum time each process can keep the token and multiplying it by the maximum number of processes that can have the token

before the given process. Although this taxonomy can be useful, it is not possible to statically determine the interactions between ASTRAL processes because the behavior of each process is dynamic and depends on interactions with the external environment. This is a problem with all behavioral classification schemes.

A structural taxonomy is described in [SG 96] that is based on software architecture. In this taxonomy, systems are classified based on different types of system components and the read and write interactions between them. Classifications in the taxonomy include pipes and filters, object-orientation, implicit invocation, layered systems, repositories, interpreters, and process control. For example, figure 8.2-1 shows a pipe and filter network. In a pipe and filter network, the processes are the filters and each process only accepts input from a set of predecessors and only passes its output to a single successor. There is at least one filter with no predecessors and at least one filter with no successors.

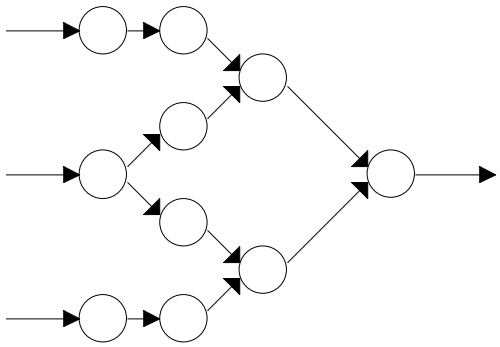


Figure 8.2-1: A pipe and filter network

A pipe and filter network could be statically identified by examining the transitions and import/export clauses of each process. To be a pipe and filter network, there must only be a single transition that references an imported variable or that is exported in each process. Additionally, there must only be a single transition that sets any exported variable. The import/export dependencies between processes can be mapped into a graph. If the transition conditions are met and if no circular dependencies exist in the graph, the system is a pipe and filter network.

Since only one transition in the filter can interact with the outside world, an immediate strategy presents itself for attempting the proofs of properties about the response time between changes to a pipe and filter network's input variables and changes to its output variables. First, each filter can be analyzed to determine the time between its input and its output. From section 8.1, since none of the transitions in a filter's sequence from input to output reference imported items or are exported, each

transition either occurs immediately after its predecessor or after a delay based on the current time. Each delay will likely be a constant time Delay($tr_i$) based on the start/end time of its predecessor. Thus, a likely execution time of the sequence is $\sum_i \text{Duration}(tr_i) + \sum_i \text{Delay}(tr_i)$ for each transition $tr_i$. The response time for the complete network is the sum of the response times for each filter that is in the chain.

Since the pipe and filter network prompts a useful proof strategy and is statically identifiable, the structural taxonomy of [SG 96] seemed promising. The taxonomy was also appealing due to already available classification software. In [DC 95], a software architecture language is presented in which a system is specified as a graph, where nodes represent different components in the system and edges represent different types of connections between components. Node types include tasks, tables, random access repositories, and files. Edge types include streams, memory accesses, messages, procedures, invocations, and productions. Once the graph for a system has been specified, a pattern matching algorithm determines the type of the system based on the [SG 96] taxonomy.

Unfortunately, there are several problems with this approach. In order to use this technique for ASTRAL, each specification must be represented by an appropriate component graph. Many of the classifications of [DC 95], however, are based on structural concepts that are not present in ASTRAL. For example, in ASTRAL, there is no distinction between tables, random access repositories, and files. They are all specified as variables and treated identically, thus only two of the four node types are available. Similarly, the only types of communication that are present in ASTRAL are memory accesses to variables, messages sent between processes, and invocations produced by the external environment. Since ASTRAL abstracts away many of these details and also does not allow arbitrary connectivity between processes and variables (e.g. one process cannot write another process's variables), it is not possible to distinguish between most of the classifications in the taxonomy.

More importantly, systems that are classified identically by this technique may have completely different proof styles. For example, consider the phone system and the elevator control system. A representation of the two systems in the [DC 95] notation is shown in figure 8.2-2.

Both of these systems are classified as client-server systems by the [DC 95] pattern matching algorithm. The proofs of the real-time response requirements in these two systems, however, are substantially different. In the phone system, the main response requirement is that a phone receives a dialtone within a bounded time of when it is picked up. In order to prove this requirement, it is

necessary to determine the maximum load possible on the server (i.e. how many clients can request services at the same time). Once the load is determined, the maximum response time can be calculated by multiplying the maximum load by the maximum time to service each request. In the elevator control system, the main response requirement is that the elevator arrives at a requested floor within a bounded time of when the button that made the request was pushed. In the proof of this requirement, there is no need to determine the load on the elevator. The response time can be bounded regardless of the number of requests that are outstanding in the building. Instead, the maximum amount of time the elevator car can spend on a floor while a request is outstanding in the building and the maximum number of floors that the elevator can stop at before arriving at the requested floor must be determined.

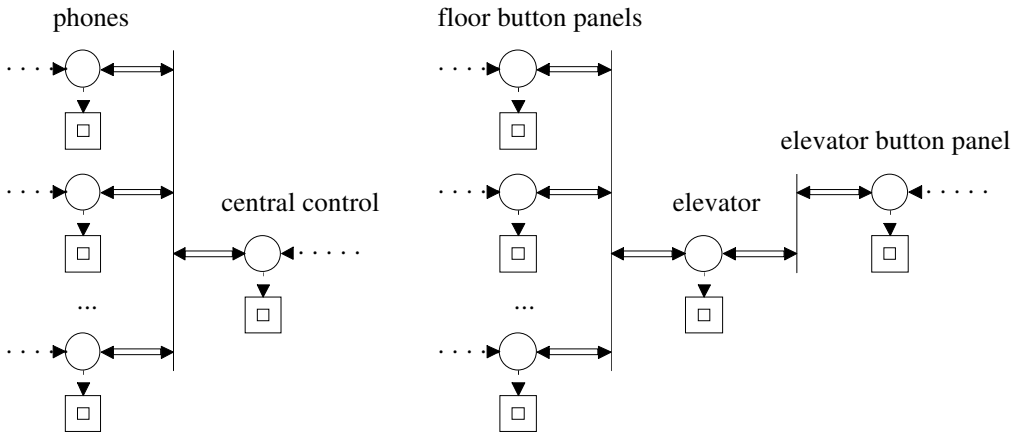



Figure 8.2-2: Phone and elevator control system structural representations

The differences in the proofs results from the differences in the associated processes. In the phone system, there are multiple independent threads interleaved on the central control that can cause a request to be delayed. In the elevator system, there is a single thread that is manipulated by the combination of all the requests outstanding in the building at any given time. Existing process classification schemes were not sufficient to differentiate such systems, thus a new scheme was needed. By examining the proofs of the testbed systems, three process classifications were identified that have significantly different proof styles. These classifications are multi-threaded processes, iterative single-threaded processes, and simple single-threaded processes. Each of these classifications is discussed in the following sections.

### 8.2.1. Multi-Threaded Processes

A multi-threaded process is a process in which multiple threads of execution are occurring at any given time in the process. For example, consider the central control process of the phone system. In this case, the "threads" are the servicing of each phone in the central control's area. Each thread consists of a chain of actions that occurs during the evolution of a call for a particular phone. The central control can only perform one stage of a single phone's thread at any given time. The phone threads of all the phones are interleaved with each other in the central control. Figure 8.2.1 shows the threads of two phones and how they might be interleaved on the central control. The labels GDT, PD, PC, etc. are abbreviations for the central control transitions Give_Dial_Tone, Process_Digit, Process_Call, etc.

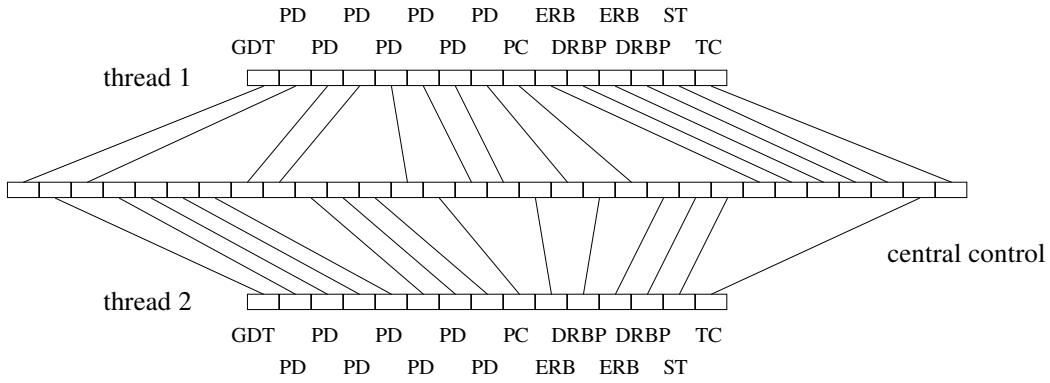



Figure 8.2.1: Interleaved phone threads on the central control

The key fact about multi-threaded processes is that there are multiple independent threads interleaved on the process, thus at any given time, any combination of thread stages may be enabled in the set of threads. This means that there is essentially no way to guarantee a real-time response property of a single thread unless some set of restrictions is placed on the behavior of the complete set of threads. For example, there may be a restriction on the number of threads that require a service at any given time or a scheduling policy such as FIFO, etc. Some of these restrictions and how real-time response requirements can be proved for each are discussed in section 9.2.6.2.

In order for an ASTRAL process to be multi-threaded, it must store information about the state of each thread so that a thread's execution may be resumed at the appropriate stage. In addition, a process must choose the thread that is to execute next when the process is idle. These two facts form the basis for the heuristic used to identify multi-threaded processes.

The process classifier examines each transition for two characteristics. First, the transition must be parameterized by some set of parameters $p_1$, ..., $p_n$. Second, there must exist some parameter $p_i$ that is used in a process identifier expression in the entry assertion that is referenced in the exit assertion. A process identifier expression is any expression that is to the left of the "." in an ASTRAL expression. For example, if procs is an array of process instances, the "procs[i]" portion of procs[i].var is a process identifier expression. If any transition in a process has these two characteristics, the process is classified as a multi-threaded process.

It is assumed that a process is multi-threaded in that each thread is associated with requests from another set of processes. In other words, that a process does not interleave activities that are not in response to the activities of other processes in the system. The thread that is chosen to execute is the thread that responds to requests from the process associated with the process identifier expression that the parameter is used in. The information about the state of the thread is stored in the expression of the exit assertion in which the parameter appears.

The Central_Control of the phone system and the Controller of the stoplight control system are the only two processes that were identified as multi-threaded processes in the testbed systems. In the Central_Control process, consider the Give_Dial_Tone transition as shown below.

```
TRANSITION   Give_Dial_Tone(P: Area_Phone)
   ENTRY     [TIME:   Tim1]
             P.Offhook
         &   Phone_State(P) = Idle
   EXIT
             Phone_State(P) BECOMES Ready_To_Dial
```

In this transition, P is the parameter that is used in a process identifier expression (P.Offhook) and that appears in the exit assertion (Phone_State(P)). In the Controller process, consider the exception of the Give_Yellow_Arrow transition shown below.

```
TRANSITION   Give_Yellow_Arrow(d: direction)
   EXCEPT    [TIME:   change_dur]
             arrow(d) = green
         &   circle(d) = green
         &   ~car(opp(d))
         &   (   car(adj1(d))
             |   car(adj2(d))
             |   LT_car(adj1(d))
             |   LT_car(adj2(d)))
         &   now - Change(arrow(d)) ≥ min_green - change_dur
         &   now - Change(circle(d)) ≥ min_green - change_dur
   EXIT
             arrow(d) BECOMES yellow
         &   circle(d) BECOMES yellow
```

This exception meets the criteria for a multi-threaded process since it is parameterized and a parameter is used in a process identifier expression (in the "car" definition) that is used in the exit assertion. Although this seems counterintuitive since the controller actually makes a single decision about the configurations of the lights, the controller can still be viewed as a multi-threaded process. In this view, there are eight threads corresponding to an arrow thread for each direction and a circle thread for each direction. Each thread cyclically changes from red to green to yellow and back to red.

In this case, however, the threads are not independent as in the phone system. That is, it is not possible for all the threads to be in arbitrary states such as all green. Instead of a single thread changing state at a time, multiple threads can change state. In order to more accurately recognize multiple independent threads, additional heuristics would need to be added to check that only the information of a single thread is updated at any given time.

### 8.2.2. Iterative Single-Threaded Processes

An iterative single-threaded process is a process that repeatedly executes a sequence of actions in a countable fashion. That is, a similar sequence of actions is performed during each iteration. Note that almost all ASTRAL processes are cyclic in some way. In an iterative process, however, there is some record of how many iterations have been performed that affects the behavior of the process. The count may represent floors in a building, loop counts in a program, etc.

For example, consider the Elevator process of the elevator control system. The Elevator process iterates over the position of the elevator car in the building. At each floor, the elevator performs a specific sequence of actions depending on the position, the direction of movement, and the requests outstanding in the building. For example, if the elevator just arrived at a floor i moving up and there is a request on floor i and on floor i+1, the sequence of actions would be door_open, door_stop, door_close, door_stop, move_up, and arrive.

The process classifier classifies a process as an iterative single-threaded process if there is an integer-based variable v of the process that is referenced in an exit assertion of some transition in a form v = g(v'), where g is an arbitrary function, such that v is referenced in the entry assertion of some transition. For example, in the Elevator process, the position variable of the elevator is integer-based and is set in the arrive transition as shown below.

```
TRANSITION   arrive
  ENTRY      [TIME:   arrive_dur]
       moving
    &  FORALL t: time
          (   t ≤ now
          &  (   End(move_down, t)
             |   End(move_up, t))
          →      now - t_move ≥ t)
    &  FORALL t, t1: time
          (   t ≤ now
          &  End(arrive, t)
          &  (   End(move_up, t1)
             |   End(move_down, t1))
          →      t < t1)
  EXIT
       IF  going_up'
       THEN position = position′ + 1
       ELSE position = position′ - 1
       FI
```

In this case, g(i) = i + 1 or i - 1 depending on the direction of travel of the elevator. The position variable can affect the behavior of the elevator in many of the transitions. For example, in the move_up transition, the requests outstanding in the building that are below, at, and above position are checked to determine if the elevator can move up.

```
TRANSITION   move_up
  ENTRY      [TIME:   move_dur]
       ~door_open
    &  ~door_moving
    &  request_above(position)
    &  (   going_up
       |   ~going_up
       &   ~request_below(position)
       &   ~the_floor_buttons[position].up_requested)
    &  (   End(arrive, now)
       &   the_elevator_buttons.floor_requested(position)
       &   ~the_floor_buttons[position].up_requested
       |   FORALL t, t1: time
              (   Change(moving, t)
              &   Change(door_open, t1)
          →       t < t1
              &   now ≥ t1 + request_dur))
  EXIT
       moving
    &  going_up
```

Without the restriction that the iteration variable appear in the entry assertion of a transition, there is the possibility that the variable is used strictly as a counter that does not affect the behavior of the process. For example, consider the Tire_Sensor process of the cruise control. The rotate transition

shown below is the only transition of the process and it adds one to the number of rotations every time it executes.

```
TRANSITION   rotate
   ENTRY     [TIME:   sense_time]
             TRUE
   EXIT
             rotations = rotations′ + 1
```

Without the additional restriction on the process, the Tire_Sensor process would be classified as an iterative single-threaded process even though it is actually just a counter.

Note that the heuristic used does not catch all forms of iterative behavior. For example, it does not catch all equivalent variations such as in the variation of the arrive transition shown below. This transition is for illustration only and does not necessarily represent how a user would specify such behavior.

```
TRANSITION   arrive(f: floor)
   ENTRY     [TIME:   arrive_dur]
             IF  going_up
             THEN f = position + 1
             ELSE f = position - 1
             FI
          &  moving
          &  FORALL t: time
                (   t ≤ now
                &  (   End(move_down, t)
                   |   End(move_up, t))
                →      now - t_move ≥ t)
          &  FORALL t, t1: time
                (   t ≤ now
                &  End(arrive, t)
                &  (   End(move_up, t1)
                   |   End(move_down, t1))
                →      t < t1)
   EXIT
             position = f
```

In this case, the new value of position meets the iterative criteria, but this can only be determined after examining the entry assertion to find the value of f. The heuristic also does not catch variations such as using the length of a list or the size of a set, even though it is possible to iterate over both. For example, using a list, the length of the list can be used to indicate the iteration number. In that case, the length of the list would be referenced in an entry assertion and an element can be added or removed from the list in an exit assertion. The list and set variations were not included in the process classifier, although they could be.

It is possible for a process to be classified as iterative even though it does not exhibit iterative behavior. For example, consider the Update transition of the Tabulate process of the Olympic boxing scoring system shown below.

```
TRANSITION   Update(B: Boxer)
   ENTRY   [TIME:    Update_Dur]
           EXISTS S: Set_Of_Judge_ID
              (   SET_SIZE (S) ≥ 3
              &   FORALL j: Judge_ID
                     (   j ISIN S
                     ↔      Now - Judges[j].Start(Score(B)) ≤ Window))
      &   Now - Start(Update) ≥ Window
      &   Outcome = In_Progress
   EXIT
           Points(B) BECOMES Points′(B) + 1
```

In this transition, Points(B) is incremented in the exit assertion and is used in the entry assertion of the Final_Decision transition. The Tabulate process, however, is a counter and not actually an iterative process. In order to more accurately determine iterative behavior, it would be necessary to examine transition exit assertions to make sure that the iterative variable is actually reset or decreased in some fashion.

Four processes are classified as iterative single-threaded processes by the process classifier. These are the Elevator process of the elevator control system, the Proc process of the bakery algorithm, and the Timer and Tabulate processes of the Olympic boxing scoring system. In addition to reporting that a process is iterative, the process classifier also reports which variable(s) the process iterates on and in which transition(s).

### 8.2.3. Simple Single-Threaded Processes

The simple single-threaded processes are the processes that are neither multi-threaded processes nor iterative single-threaded processes. These processes do not necessarily exhibit "simple" behavior. Rather, a cycle of a simple single-threaded process's execution represents the interval over which properties in the process must be proved. This is in contrast to an iterative single-threaded process in which a property may need to be proved over multiple cycles of the process's execution. Simple single-threaded processes are the most common process type. 19 of the 25 process types in the testbed systems are simple single-threaded processes.

The process classification information can be obtained by the user by selecting a process in the process browser window as described in section 8.4 and performing the "process information" query.

In addition to the classification of the process, the variables and transitions that are imported and exported will also be shown.

## 8.3. Property Classification

Every ASTRAL property can be written in the form "context → requirement", where the context is a conjunction of unnegated conditions and the requirement is a disjunction of unnegated conditions. The context describes the conditions under which the requirement must hold. That is, the requirement is not required to hold under any conditions in which the context does not hold. A property may have an empty context, "TRUE → requirement", or an empty requirement, "context → FALSE". The *context times* are the times that are referenced in some past, change, start, end, or call expression in the conditions of the context. These times may be concrete times such as now - 5, or symbolic times such as a quantified time variable t that has been restricted in some way. The *requirement times* are the times that are referenced in a similar expression in the conditions of the requirement. For example, consider the following property of the Speed_Control process of the cruise control system. In this property, the only context time is now - input_dur - input_dur and the only requirement time is t.

```
      control_dur ≤ input_dur
  &   now ≥ input_dur + input_dur
  &   past(maintaining_speed, now - input_dur - input_dur)
  &   call(set_brake_pedal, now - input_dur - input_dur)
→     EXISTS t: Time
          (   now - input_dur - input_dur ≤ t
          &   t ≤ now
          &   ~past(maintaining_speed, t))
```

Every ASTRAL property is naturally classified based on the section of the specification in which it occurs (i.e. invariant, schedule, etc.). Besides this classification, a property can also be classified based on the forms and the times of the context and the requirement of a property. The following sections describe these classifications. Every ASTRAL property falls into one of these five classifications.

### 8.3.1. Untimed Properties

An untimed property is a property in which the only context and requirement time is now. A constraint property is considered to be untimed if it is untimed when all the primes are removed. The most common form of an untimed property is one that consists solely of boolean combinations of local state variables. For example, the property shown below is an untimed property of the Controller

process of the stoplight control system. In this property, whenever the circle of a direction is green, the arrow of the opposing direction must be red.

```
FORALL d: direction
    (   circle(d) = green
→       arrow(opp(d)) = red)
```

An untimed property may reference timed operators as long as they are always evaluated at the current time. For example, the property shown below is an untimed property of the Proc process of the bakery algorithm. This property states that whenever number changes to zero, the process is not in its critical section. In this case, the change expression significantly limits the context in which the requirement portion of the property must be proved.

```
        change(number, now)
    &   number = 0
→       ~in_critical
```

### 8.3.2. Timed Forward Properties

A timed forward property is a property in which there is some context time that is less than or equal to every requirement time. Thus, in a timed forward property, the reasoning proceeds forward from a known state in the system.

#### *8.3.2.1. Forward Safety Properties*

In a forward safety property, the requirement must hold at all times in an interval of time that begins after the earliest context time. For example, the property shown below is a forward safety property of the Central_Control process of the phone system.

```
FORALL P: Area_Phone, t: Time, t1: Time, t2: Time
    (   t ≤ t1
    &   t1 < t2
    &   change2(Phone_State(P), t)
    &   past(Phone_State(P), t) = Idle
    &   P.end(Pickup, t1)
    &   P.Offhook
    &   change(Phone_State(P), t2)
→       (   past(Phone_State(P), t2) = Ringing
        |   past(Phone_State(P), t2) = Ready_To_Dial))
```

This property states that if a phone changes to Idle and then is picked up, the next change of the phone's state will be to Ringing or Ready_To_Dial. This property is a forward property because there is context time, t, that is less than or equal to every requirement time. It is a safety property because the requirement is required to hold at all times in the interval (t1, now].

*8.3.2.2. Forward Liveness Properties*

In a forward liveness property, the requirement must hold at least once in an interval of time that begins after the earliest context time. For example, the property shown below is a forward liveness property of the P_Robot process of the production cell.

```
FORALL t: Time
    (   start(Arm1_Drop, now)
    &   end(Arm1_Drop, t)
→       EXISTS t1: Time
            (   t < t1
            &   t1 < now
            &   end(Arm2_Pickup, t1)))
```

This property states that between any consecutive drop of an object by arm one, arm two must pickup an object. This property is a forward property because there is context time, t, that is less than or equal to every requirement time. It is a liveness property because the requirement is only required to hold once in the interval (t, now).

### 8.3.3. Timed Backward Properties

A timed backward property is a property in which there is some requirement time that is less than or equal to every context time. Thus, in a timed backward property, the reasoning proceeds backward from a known state in the system.

*8.3.3.1. Backward Safety Properties*

In a backward safety property, the requirement must hold at all times in an interval of time that begins before the earliest context time. For example, the property shown below is a backward safety property of the Sensor process of the railroad crossing.

```
FORALL t: Time
    (   change(train_in_R, now)
    &   now - ((dist_R_to_I + dist_I_to_out) / max_speed - response_time) ≤ t
    &   t < now
→       train_in_R
    |   past(train_in_R, t))
```

This property states that whenever a train has just left the region, the sensor has been reporting that there is a train for at least the past (dist_R_to_I + dist_I_to_out) / max_speed - response_time time. This property is a backward property because there is a requirement time, t, that is less than or equal to every context time. It is a safety property because the requirement is required to hold at all times in the interval [now - ((dist_R_to_I + dist_I_to_out) / max_speed - response_time), now).

*8.3.3.2. Backward Liveness Properties*

In a backward liveness property, the requirement must hold at least once in an interval of time that begins before the earliest context time. For example, the property shown below is a backward liveness property of the Tabulate process of the Olympic boxing scoring system.

```
    Outcome ≠ In_Progress
→ EXISTS t: Time
      (   t ≥ 0
      &   t ≤ now
      &   end(Final_Decision, t))
```

This property states that if the fight is not in progress, then a final decision has previously been reached. This property is a backward property because there is a requirement time, t, that is less than or equal to every context time. It is a liveness property because the requirement is only required to hold once in the interval [0, now].

### 8.3.4. Classification Heuristics

The property classifications presented above correspond closely to temporal logic operators presented in section 3.1.1. Forward safety and forward liveness correspond to the henceforth and eventually operators, while backward safety and backward liveness correspond to the has-always-been and once operators. Unlike temporal logics, however, in which these operators are built into the language and are thus easily recognizable, in ASTRAL it is not possible to identify these classifications without further analysis.

The classification of a property is automatically displayed when the splits of a given clause are generated using the formula splitter discussed in section 5.4. A split formula is first checked to see if it is untimed. If the formula is timed, it is then determined whether the formula is a safety property or a liveness property by checking for an existential time quantifier in the consequent of the split. If such a quantifier exists, then the formula is a liveness property. Otherwise, it is a safety property.

It is not possible to determine precisely whether a property is forward or backward without the use of decision procedures that perform rewriting. The rewriting is necessary to determine the equivalence of context and requirement times. For example, without rewriting, a time t1 + t2 is not necessarily equivalent to t2 + t1. Rewrite rules take advantage of numeric properties such as commutativity, thus would be able to determine this information for some expressions. Rewriting by itself, however, is not enough to determine the equivalence of context and requirement times. For example, the equivalence of t1 + c1 and t1 + c2 cannot usually be determined without additional information about

the relationship between c1 and c2. This information may be stated in the axiom clauses of the specification, which can be arbitrary first-order logic formulas, thus determining if two times are equivalent is undecidable.

Since decision procedures with rewriting can incur a large computational overhead and do not guarantee that a precise classification will be found, instead a heuristic is used to determine whether a given property is backward or forward. All the context times are first collected as they appear. For a liveness property to be backward, an existentially quantified time variable must be constrained to be less than some other time and if it is constrained to be greater than another time, the lesser time must not be a context time. If the lesser time is a context time for all such quantified variables, then the property must be forward. For a safety property to be backward, a universally quantified time variable must be constrained similarly and in addition, must be used in the consequent. Table 8.3.4 shows the number of each type of property in each of the testbed systems. The properties are broken down into the properties that occur in the requirements sections (i.e. invariant, schedule, and constraint clauses) and those that occur in the assumptions sections (i.e. environment and imported variable clauses).

| System | Requirements | | | | | Assumptions | | | | | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | U | FS | FL | BS | BL | U | FS | FL | BS | BL | |
| Bakery Algorithm | 11 | 1 | 0 | 0 | 1 | 3 | 1 | 0 | 0 | 1 | 18 |
| Cruise Control | 5 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 |
| Elevator | 8 | 0 | 8 | 0 | 3 | 2 | 9 | 0 | 0 | 3 | 33 |
| Olympic Boxing | 8 | 2 | 0 | 0 | 1 | 1 | 5 | 0 | 0 | 0 | 17 |
| Phone | 26 | 14 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 7 | 55 |
| Production Cell | 32 | 3 | 6 | 0 | 8 | 0 | 1 | 4 | 0 | 4 | 58 |
| Railroad Crossing | 0 | 7 | 0 | 1 | 0 | 0 | 2 | 0 | 1 | 0 | 11 |
| Stoplight | 17 | 4 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 23 |
| Total | 107 | 31 | 16 | 1 | 15 | 6 | 26 | 4 | 1 | 15 | 222 |

Table 8.3.4: Property classifications of testbed systems

## 8.4. Browsers

The process, transition, and variable browsers in the right portion of figure 5.1 and the formula splitter in figure 5.4 enable the user to view various relationships between the four types of items. To simplify the discussion, let “browser” refer to either the formula splitter or to the variable, transition, or process browsers. The queries available in the browsers range from simple queries involving a single step to compound queries involving multiple complex steps. Some simple queries include finding the transitions exported from a process, finding the variables referenced in the entry assertion

of a transition, finding the processes that import a variable, and finding the transitions referenced in a formula. More sophisticated queries include finding the transitions that are the predecessors of a transition, finding the formulas that reference a variable in their antecedents, finding the transitions that set the same variables as a transition, and finding the transitions with higher priority than a transition.

In small specifications, the user may be able to determine the results of simple queries manually in a reasonable amount of time. In larger specifications or for more complex queries, however, the difference in speed and accuracy between obtaining the information manually and obtaining it using the browsers will be markedly different. The browsers make use of symbol tables maintained during editing to quickly ascertain and display the appropriate information.

The querying process begins by selecting an item in one of the browsers and pressing the "Query.." button in the corresponding browser, which brings up a pulldown menu with four types of queries. Each query type corresponds to the browser in which the corresponding result of the query will be displayed. Some of the query types are currently empty. For example, the variable browser currently does not have any variable type queries. The transition and process query menus have an additional entry "transition information" and "process information", which brings up a window with the relevant transition or process classifier information as discussed in sections 8.1 and 8.2, respectively.

When a query type is selected, a pullright menu appears with all of the queries of that type. Once a query is selected, the appropriate items are retrieved and are displayed in the browser associated with the query type. Thus, the result of one query becomes the input of the next. Any of the items that appear within the browser windows as the result of the query can be double clicked on to move the navigation window to the item's declaration within the specification. Since a browser may have queries of its own type and sequences of queries may be displayed in a browser whose results were still being used, the query results are stored on a stack in each browser. Thus, for each query, a new frame is pushed onto the appropriate browser's stack and the new query results are stored on the new frame. When a frame is no longer needed, the "Pop" button of each browser can be used to pop the frame and display the new top of the stack.

Since many frames may be on a stack of a browser, it is useful to be able to determine which query created each frame of the stack. The first line of results in each browser indicates the number of the frame, the query the frame was generated by, and the browser and frame number of that browser that the query was invoked on. For example, in the variable browser, a top line of "<<frame 3 from V_EXP on PB frame 1>>" indicates that the displayed frame is the third frame on the variable

browser stack, and that it was generated by the V_EXP query invoked on the first frame of the process browser. The V_EXP query refers the "variables..exported by Selected Process" query.

An ASTRAL specification may consist of multiple process levels. Thus, it is necessary to be able to query items of different levels and to distinguish between these items. In the variable and transition browsers, items are displayed in the form "[pname__lnum] vname", where vname is a variable in level lnum of process pname. For the top level, the lnum portion is discarded. Each level below the top level is consecutively numbered starting from one. For example "[Input__1] Msg" indicates the Msg variable of the first refinement level of the Input process. In the process browsers, each level is displayed separately in the form "pname__lnum". In the formula splitter, the level is indicated in the same form on the "property classification" line.

In general, queries are always performed with respect to the level of the item being queried. For example, a transition query of a second level variable would only search transitions of the second level of the same process and not transitions of the top level of that process. It would, however, search transitions of the top level of other processes.

Just as processes may have multiple levels, transitions may have multiple exceptions. Thus, it is also necessary to be able to query different exceptions and to distinguish between them. In the transition browser, items are displayed in the form "[pname__lnum] tname__enum", where enum is an exception of the transition tname in level lnum of process pname. Every exception of a transition is considered a separate transition in each transition type query. Thus, in essence, the process browser is actually a "level browser" and the transition browser is actually an "exception browser".

A browser session can begin in two ways. The most common way is to use the special "processes..defined in specification" query in the process browser, which does not require any browser item to be highlighted. This query displays all levels of all processes declared in the current specification. After this query, the other process browser queries can be used to display items in other browsers. The other way a browser session can begin is to split a formula in the navigation window with the "Split" button as discussed in section 5.4. The split formulas that are shown can be queried to display items in other browsers.

The browsers in figure 5.1 demonstrate the results of a sample browser session on the railroad crossing specification. First, all processes declared in the specification were listed with the "processes..defined in specification" process browser query. Then, the "variables..declared in or imported by Selected Process" query was performed on the Gate process. Finally, the

"transitions..using Selected Variable in an entry clause" query was performed on the train_in_R variable. The end result is a listing in the transition browser window of transitions in which train_in_R appears in an entry clause.

The browsers are especially useful during the maintenance phase, since in many cases it is someone other than the original developer who is responsible for maintaining the specification. In addition, even the original specifier may be unclear on some of the details due to the elapsed time between updates. In either case, the browsers can be used to quickly determine the portions of the specification that may be affected by any proposed changes. For example, suppose that during maintenance the effect that a transition has on some variable needs to be changed. In this case, it is desirable to determine those transitions that may be affected by the change, namely those that use the variable in an entry assertion. Once the transitions are listed with the appropriate browser queries, they can be quickly scanned to determine which ones will be affected by the update.

The browsers can also assist in the proof process. In chapter nine, the browsers are used in a variety of ways during model checker test case generation and proof sketch construction. For example, in the proof of an untimed property "A $\rightarrow$ C" as discussed in section 9.2.4.1.1, the browsers are used to find the transitions that make C false or A true. This is done by first bringing up the property in the formula splitter. To find the transitions that make C false, the "variables..used in consequent of Selected Formula" query is used. Similarly, the "variables..used in antecedent of Selected Formula" query is used to find the transitions that make A true. Once the variables are displayed in the variable browser, the "transitions..using Selected Variable in an exit clause" query is used, which displays the transitions that can possibly violate the property. Any transition that is not listed cannot possible violate the property.

## 8.5. Transition Sequence Generator

Determining the order in which transitions can fire on a given process is essential to proving that the critical requirements of the process hold. Without this information, it is impossible to determine the transition sequences that can occur on a process, thus it is impossible to determine which states are reachable and which are unreachable. Therefore, it is impossible to guarantee any property of the process. Since sequencing is so crucial to the proof process, it is useful to provide the user with a tool to view the transition sequences that can occur in a given process type. A sequence generator tool is not only useful in itself, but it can also be used as the foundation for more complex analysis tools and/or techniques.

For example, a transition sequence generator can be used as the basis for a symbolic executor, which helps the user visualize the operation of the system. To accomplish this, the browsers are first used to determine the transitions that must fire in order to achieve the start state given by the user. The sequence generator is then used to find the sequences of transitions that are possible between the transitions obtained from the browsers. The transition classifier is used to classify the transitions in each sequence to determine whether each transition fires immediately after its predecessor or is delayed according to the current time, the other processes in the system, or the external environment. This time can be estimated by inspecting the entry assertion of the transition and the relevant environment and imported variable clauses. Finally, the path condition of each sequence is constructed by conjoining the entry and exit assertions of all the transitions in the sequence together. The end result is an expression for the process state at each point in the execution of the sequence with an approximate running time up until that point. Other possible uses of the sequence generator will be discussed in later chapters.

Unlike graphical state-machine languages in which the successor information of each transition is part of the specification, in textual languages such as ASTRAL, sequencing cannot be determined without more in-depth analysis. Determining whether one transition is the successor of another in ASTRAL, however, is undecidable since transition entry/exit assertions may be arbitrary first-order logic expressions. Many successors, however, can be eliminated based only on the simpler portions of the entry/exit assertions, such as boolean and enumerated variables. Based on this fact, a transition sequence generator tool has been developed.

### 8.5.1. Sequence Generator Proof Obligations

The sequence generator first eliminates as many transition successors as possible using the PVS prover. This is done by attempting the proof of an obligation Not_Sequence(tr1, tr2) for each pair of transitions (tr1, tr2) as shown below. Note that Not_Sequence only states that some transition must end between tr1 and tr2 and does not exclude tr1 or tr2 from firing. If Not_Sequence(tr1, tr2) holds, however, it is sufficient to prove that a transition besides tr1 and tr2 must fire in between any firing of tr1 and tr2. If only tr1 and tr2 fire in between t1 and t2, then since t2 - t1 is finite and the durations of tr1 and tr2 are constant and non-null, eventually a contradiction can be achieved by applying Not_Sequence(tr1, tr2) repeatedly on an ever shortening interval. An obligation Not_Initial(tr1) is also attempted to prove that each transition tr1 is not the first to fire after the initial state.

```
Not_Sequence(sub1: transition,
        sub2: transition): bool =
    (FORALL (tr1: transition):
      (FORALL (tr2: transition):
        (FORALL (t1: time):
          (FORALL (t2: time):
            tr1 = sub1 AND
            tr2 = sub2 AND
            t1 + Duration(tr1) ≤ t2 AND
            Fired(tr1, t1) AND
            Fired(tr2, t2) AND
            (FORALL (tr3: transition, t3: time):
                t1 + Duration(tr1) <
                    t3 + Duration(tr3) AND
                t3 + Duration(tr3) ≤ t2 IMPLIES
                    NOT Fired(tr3, t3)) IMPLIES
                FALSE))))
```

```
Not_Initial(sub: transition): bool =
    (FORALL (tr2: transition):
        (FORALL (t2: time):
            tr2 = sub AND
            Fired(tr2, t2) AND
            (FORALL (tr1: transition, t1: time):
                t1 + Duration(tr1) ≤ t2 IMPLIES
                    NOT Fired(tr1, t1)) IMPLIES
            FALSE))
```

The quantifications over the transitions in the definitions of Not_Sequence and Not_Initial (tr1 and tr2 in Not_Sequence and tr2 in Not_Initial) are trivial quantifications that were added to allow the PVS strategies that discharge these obligations to be written for arbitrary transition names. That is, the strategies can always reference the transitions by tr1 and tr2 whether the obligation is Not_Sequence(up, down), Not_Sequence(lower, raise), or any other instantiated form.

### 8.5.2. Sequence Generator Strategies

The PVS strategies try-seq-gen and try-seq-gen-0 shown in appendix C were written to automatically discharge these obligations. The try-seq-gen strategy uses abstract machine axioms to introduce the entry and exit assertions of tr1, the entry assertion of tr2, and the fact that if nothing ended between the end of tr1 and the start of tr2, then all variable values remained constant during this time. Once all of this information is present, a modified version of the PVS grind command, which is a heavy-duty decision procedure that performs rewriting, skolemization, and automatic quantifier instantiation, is invoked to finish the proof. Grind in unmodified form rewrites all definitions in a specification. The modified version, my-grind, does not rewrite the timed ASTRAL operators, since it is unlikely that the decision procedures could use the information efficiently, thus expanding the operators would only increase the running time of the strategy. The my-grind strategy is discussed in more detail in section 10.4.6. The try-seq-gen-0 strategy is similar but uses the initial clause of the process in place of the information about tr1.

### 8.5.3. Sequence Generator Strategy Results

Table 8.5.3 shows the results of using these strategies to compute the successors for each process type in the testbed systems. For each process type, the table shows the maximum number of successors, the number of successors that are provably possible, and the number that were computed automatically using the try-seq-gen strategies. Note that the number of successors that are provably possible is the number that can be proved without any additional assumptions such as the environment or imported variable clauses.

| System | Process Type | maximum successors | actual successors | computed successors |
|---|---|---|---|---|
| Bakery Algorithm | Proc | 42 | 8 | 25 |
| Cruise Control | Accelerometer | 2 | 2 | 2 |
| | Speed_Control | 132 | 76 | 94 |
| | Speedometer | 2 | 2 | 2 |
| | Tire_Sensor | 2 | 2 | 2 |
| Elevator | Elevator | 42 | 13 | 24 |
| | Elevator_Button_Panel | 6 | 4 | 4 |
| | Floor_Button_Panel | 20 | 14 | 14 |
| Olympic Boxing | Judge | 2 | 2 | 2 |
| | Tabulate | 12 | 4 | 6 |
| | Timer | 6 | 3 | 3 |
| Phone | Central_Control | 420 | 235 | 312 |
| | Phone | 110 | 50 | 69 |
| Production Cell | P_Crane | 156 | 13 | 36 |
| | P_Deposit | 6 | 3 | 3 |
| | P_Deposit_Sensor | 6 | 3 | 3 |
| | P_Feed | 20 | 14 | 14 |
| | P_Feed_Sensor | 6 | 3 | 3 |
| | P_Press | 42 | 7 | 7 |
| | P_Robot | 420 | 21 | 129 |
| | P_Table | 72 | 9 | 21 |
| Railroad Crossing | Gate | 20 | 7 | 7 |
| | Sensor | 6 | 3 | 3 |
| Stoplight | Controller | 506 | 92 | 198 |
| | Sensor | 6 | 3 | 3 |
| Total | | 2064 | 593 | 986 |

Table 8.5.3: Transition successors of testbed systems

There are two main factors that contribute to the difference between the number of successors that are provably possible and the number computed by the try-seq-gen strategies in the testbed systems. The first factor is that entry assertions do not usually constrain all of the state variables of a process. For example, the entry assertion of the Arrived_At_Upper transition of the P_Table process in the

production cell shown below constrains the value of v_status, but does not constrain the value of h_status, which is the other variable of the P_Table process.

```
TRANSITION   Arrived_At_Upper
   ENTRY      [TIME:   table_arrive_dur]
         v_status = moving_to_robot
      &  now - Change(v_status) ≥
            t_move_table_level
   EXIT
         v_status = at_robot
```

```
TRANSITION Arrived_At_Robot
   ENTRY      [TIME:   table_arrive_dur]
         h_status = moving_to_robot
      &  now - Change(h_status) ≥
            t_rotate_table
   EXIT
         h_status = at_robot
```

When proving Not_Sequence(Arrived_At_Upper, Arrived_At_Robot), PVS does not have information about the value of h_status at the start of Arrived_At_Upper, which is only derivable from the transitions preceding Arrived_At_Upper. Thus, PVS must assume an arbitrary symbolic value for h_status. One possible value that h_status can have is moving_to_robot, thus PVS cannot eliminate the possibility that Arrived_At_Robot immediately follows Arrived_At_Upper. It is provable that this is not the case, however, because it is not possible to find a sequence of transitions starting from the initial state in which Arrived_At_Robot can immediately follow Arrived_At_Upper. The only possible predecessor to Arrived_At_Upper is Move_To_Upper and by the entry assertion of Move_To_Upper, h_status = at_belt, which means that Arrived_At_Robot cannot be enabled after Arrived_At_Upper. By similar analysis, it is possible to show that there is a single sequence of execution in the P_Table process, which is Move_To_Upper, Arrive_At_Upper, Rot_CW_To_Robot, Arrived_At_Robot, Rot_CCW_To_Feed, Arrived_At_Feed, Move_To_Lower, and Arrived_At_Lower.

In order to improve the performance of the sequence generator for these processes, it would be necessary to examine sequences back to a transition that causes a contradiction. This is a nonterminating procedure, however, whenever the second transition of Not_Sequence actually is a successor of the first, thus it is necessary to specify termination conditions such as a specific number of transitions into the past or similar criteria. In general, this procedure is not worth the additional time it would require unless the number of successors that could be eliminated using a small number of backward steps is significantly higher than the number of actual successors. As an alternative, the user can fully constrain all of the state variables in the entry assertions.

The second factor that contributes to the difference between the number of provable successors and the number computed by the try-seq-gen strategies is the use of timed operators to define the sequencing between different operations. For example, consider the set_number transition of the bakery algorithm specification shown below.

```
TRANSITION   set_number
  ENTRY      [TIME:   exec_time]
             choosing
         &   FORALL t: time
                 (   Change(number, t)
             →       t < Change(choosing))
  EXIT
             FORALL i: procs_int
                 (   number ≥ procs[i].number + 1)
                 &   EXISTS i: procs_int
                         (number = procs[i].number + 1)
```

Not_Sequence(set_number, set_number) is provable because when set_number first fires, Change(number) < Change(choosing). By the exit assertion of set_number, number must change because the new value of number is at least one more than the largest number of all the processes including itself. Therefore, when set_number fires again, Change(number) > Change(choosing), which is a contradiction by the entry assertion.

The use of change is essential to the proof that set_number cannot follow itself. The definition of the change operator within PVS, however, is quite complex with several quantifiers, and the use of change without a time expression complicates the definition even more. Thus, there is little hope that PVS could automatically prove such an obligation. For this reason, my-grind does not expand any of the timed operators in the try-seq-gen strategies. This prevents work from being wasted.

In many cases, the timed operators are used only to specify ordering among transitions. For example, in the set_number transition shown above, the quantified change expression is used to specify that set_number can only follow set_choose. In these cases, it would be advantageous to add a "follows" operator to the language, which takes a transition tr1 and returns true if and only if tr1 was the last transition to fire on the process. The follows operator can be defined as:

```
follows(tr1: transition): bool ==
        EXISTS t1: time
            (   t1 < now
            &   past(Start(tr1, t1), t1))
            &   FORALL tr2: transition, t2: time
                    (   tr2 ≠ tr1
                    &   t2 < now
                    &   past(Start(tr2, t2), t2)
                →       t2 < t1)
```

This operator would not only make specifications more readable, but would also allow the sequence generator obligations to be proven with more accuracy. With this operator, the entry assertion of set_number would become "choosing & follows(set_choose)".

### 8.5.4. Parameterized Transition Sequences

When a transition is parameterized, each set of parameters represents one possible choice that a process can make. Usually, the start of a transition with one set of parameters does not preclude the start of the same transition with a different set of parameters immediately afterward. Thus, the sequences generated for parameterized transitions are not particularly useful since many of the sequences will consist of the same transitions repeated over and over again. For example, in the Central_Control process of the phone system, all of the transitions are parameterized, thus any transition can essentially follow any other.

Since the standard sequence generator proof obligations do not ordinarily produce a useful result, a parameterized extension has been added to the sequence generator. In this extension, if two transitions have the same parameter list (i.e. the same number of parameters and parameter types), the sequence generator proof obligations are attempted assuming that the parameters are the same. That is, the sequences are generated with a fixed set of parameters among consecutive transitions. For example, in the Central_Control process, this would find the sequences of a single thread of execution. In the Controller process of the stoplight control system, this would find the sequences of a single direction. Note that once a transition appears in the sequence that does not have the same parameter list, a new parameter is assumed. That is, a sequence would be generated as t1(p1), t2(p1), t3, t4(p2), t5(p2) rather than t1(p1), t2(p1), t3, t4(p1), t5(p1).

In the definition of Not_Sequence below, when a "true" argument is given, then it is assumed that the transitions have equivalent parameter lists and the parameterized extension should be used. When a "false" argument is given, this definition reverts to the definition of section 8.5.1. The actual and computed results in table 8.5.3 take the parameterized extension into account.

```
Not_Sequence(sub1: transition, sub2: transition, parm_eq: bool): bool =
    (FORALL (tr1: transition): (FORALL (tr2: transition):
        (FORALL (t1: time): (FORALL (t2: time):
            tr1 = sub1 AND
            tr2 = sub2 AND
            t1 + Duration(tr1) ≤ t2 AND
            Fired(tr1, t1) AND
            Fired(tr2, t2) AND
            IF parm_eq THEN
                Fire_Parms(Base_Trans(tr1), t1) = Fire_Parms(Base_Trans(tr2), t2)
            ELSE TRUE
            ENDIF AND
            (FORALL (tr3: transition, t3: time):
                t1 + Duration(tr1) < t3 + Duration(tr3) AND
                t3 + Duration(tr3) ≤ t2 IMPLIES
                    NOT Fired(tr3, t3)) IMPLIES FALSE))))
```

An observation about parameterized transitions is that any parameter types associated with such a transition is almost always associated with a variable in the same process as well. For example, all of the variables in the Central_Control process of the phone system are parameterized by an "Area_Phone", which is also the parameter type of most of the transitions. Another example is the Controller process of the stoplight control system. In this case, both variables of the process are parameterized by "direction", which is the parameter type of all of the transitions. In these transitions, the parameter values for which the transitions fire are used to set a particular parameter value of a variable. For example, in the Give_Dial_Tone transition shown below, the transition parameter P is used to set Phone_State(P).

```
TRANSITION Give_Dial_Tone(P: Area_Phone)
   ENTRY       [TIME:    Tim1]
               P.Offhook
      &        Phone_State(P) = Idle
   EXIT
               Phone_State(P) BECOMES Ready_To_Dial
```

This observation allows an optimization to be applied in the try-seq-gen strategies. When the transition in a Not_Initial obligation or either of the transitions in a Not_Sequence obligation is parameterized, the try-seq-gen-0-p and try-seq-gen-p strategies are used in place of try-seq-gen-0 and try-seq-gen, respectively. These strategies optimize the application of the vars_no_change axiom. Normally, each parameterized variable in a process adds an additional quantification to the Vars_No_Change definition of that process. For example, the Vars_No_Change definition of the Controller process is shown below.

```
Vars_No_Change(T1: time, T2: time): bool =
   (FORALL (V1: direction): circle(V1)(T1) = circle(V1)(T2)) AND
   (FORALL (V1: direction): arrow(V1)(T1) = arrow(V1)(T2))
```

Thus, when the vars_no_change axiom is applied in the try-seq-gen strategies, PVS spends a significant amount of time trying to instantiate these quantifications automatically. Since parameterized transitions normally reference parameterized variables with the same parameters they are given by the above observation and transitions with the same parameter type are assumed to fire with the same parameters by the definition of Not_Sequence, the quantifications of Vars_No_Change can be instantiated with an appropriate value of Fire_Parms. This reduces the number of quantifications that PVS must attempt automatically and significantly reduces the execution time of the try-seq-gen strategies for parameterized transitions.

### 8.5.5. Transition Sequence Construction

The sequence generator is invoked by the "SeqGen" button shown in figure 5.1, which generates sequences for the process level currently being viewed in the navigation window. This button brings up the sequence generator dialog box, shown on the left of figure 8.5.5, which allows the user to direct the construction of the transition sequences. The options include the first and last transitions, the direction to generate sequences from the first transition, the maximum number of transitions per sequence, and the maximum number of sequences. There is also an option to disallow sequences in which the same transition appears more than once (besides as the first or last transition). The user must provide the maximum number of transitions per sequence and if the search is backward, must provide the first transition. Once this information is provided, the SDE constructs the sequence generator obligations, invokes PVS, runs the proof scripts, retrieves the results, and then generates the sequences according to the user query. Since running the proof scripts can be time-consuming, the results are saved between changes to the specification, so that sequences can be quickly displayed after the proofs are attempted once.

For each sequence generated, an approximate running time of the sequence is constructed using the transition classifier information. The sum of the durations of all the transitions plus an appropriate delay based on the classification of each transition in the sequence is displayed next to the sequence. A delay produced by the current time is denoted delay_T. A delay caused by other processes in the system is denoted delay_O. Finally, a delay caused by the external environment is denoted delay_E. For delays involving multiple causes, the notations are combined. For example, if a delay is caused by both the current time and other processes, it is denoted delay_OT.

The sequence generator is complete without the parameterized extension (i.e. if a sequence is possible, it will appear as a result) since the successor obligations are performed using the PVS encoding, which will only eliminate a successor if it is derivable that it cannot occur. The sequence generator is not complete with the parameterized extension because it does not display any sequences in which two parameterized transitions with the same parameter lists are given different parameters.

The performance of the sequence generator can be improved by manually performing the proofs of those successor obligations that can actually be proved but could not be automatically proved by the try-seq-gen strategies. The time used to run the proof scripts or to refine the performance of the sequence generator is not wasted because any successor eliminated can be used as a lemma in the main proof obligations. For any two transitions tr1 and tr2 for which Not_Sequence(tr1, tr2, FALSE) has been proved, the not_seq_ax lemma, shown below, can be used in the main proof obligations to

show that tr2 cannot follow tr1. Note that this lemma is not valid for Not_Sequence(tr1, tr2, TRUE) because even though tr2 cannot follow tr1 when it is given the same parameters, it may be able to follow tr1 with different parameters. The lemma not_init_ax can be used for any transition tr1 for which Not_Initial(tr1) has been proved.

```
not_seq_ax: LEMMA
    (FORALL (tr1: transition):
     (FORALL (tr2: transition):
       Not_Sequence(tr1, tr2, FALSE) IMPLIES
          (FORALL (t1: time):
           (FORALL (t2: time):
             t1 < t2 AND
             Fired(tr1, t1) AND
             Fired(tr2, t2) IMPLIES
                (EXISTS (tr3: transition,
                         t3: time):
                   tr3 ≠ tr1 AND tr3 ≠ tr2 AND
                   t1 < t3 AND t3 < t2 AND
                   Fired(tr3, t3))))))
```

```
not_init_ax: LEMMA
    (FORALL (tr2: transition):
       Not_Initial(tr2) IMPLIES
          (FORALL (t2: time):
             Fired(tr2, t2) IMPLIES
                (EXISTS (tr1: transition,
                         t1: time):
                   tr1 ≠ tr2 AND
                   t1 < t2 AND
                   Fired(tr1, t1))))
```

As an example of a sequence generator query, consider the sequences of length seven generated backward from the arrived_at_middle transition of the P_Press process of the production cell. Figure 8.5.5 shows the sequence generator dialog box and the first of the two sequences that are generated.

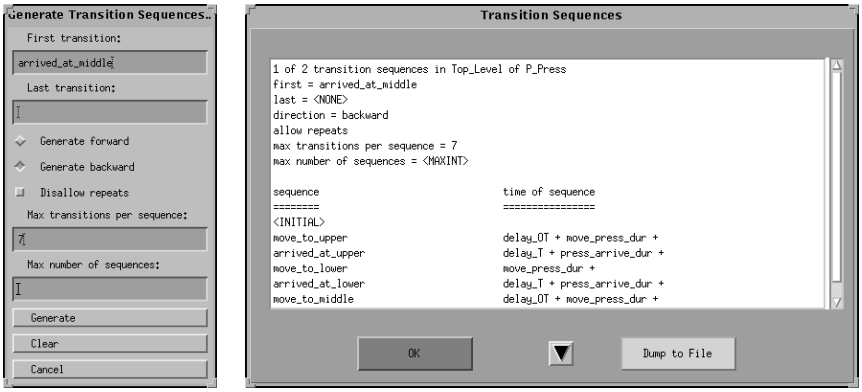



Figure 8.5.5: Transition sequences of the production cell press

# Chapter 9

# Test Case Generation and Proof Sketch Construction

In order to assure that an ASTRAL specification satisfies its requirements, it is necessary to generate and prove the appropriate proof obligations. ASTRAL proofs are divided into three categories as discussed in section 5.6: intra-level proofs, inter-level proofs, and composition proofs. In this chapter, the focus will be on the intra-level proofs. The intra-level proofs are the most basic type of proof on which the inter-level and composition proofs are based. That is, any technique developed for the intra-level proofs can be readily applied to the inter-level and composition proofs.

ASTRAL proof obligations can be further broken down into *correctness proofs* and *consistency proofs*. In correctness proofs, the critical requirements of the system are proven to hold based on the executions of each process. In consistency proofs, it is proven that any assumptions made in the system are never false. Examples of correctness proofs include proving that the invariant and schedule clauses of a process hold at all times. Examples of consistency proofs include proving that environmental assumptions do not contradict each other, proving that the initial clause of each process is satisfiable, and proving that transition exit assertions never evaluate to false. In this and the following chapter, only the correctness proofs are discussed. Although consistency proofs cannot be disregarded, in most cases, an unprovable consistency proof is due to a poorly written specification rather than a design flaw. Since most consistency problems are easily avoidable and design flaws are much more critical to the analysis of a specification, the focus of the systematic analysis methodology is on correctness proofs.

The intra-level proofs are also the most ASTRAL-independent of the proof obligations. In all the languages of chapter three, real-time properties are proven over a timed sequence of events. Each real-time language has its own set of event types. For example, in DL, discussed section 3.1.3.1, the event types are transition events, external events, notifier events, and notification events, while in CCSR, discussed in section 3.3.1, there are connection events and annotation events. In ASTRAL,

the events in timed sequences are calls, starts, and ends of transitions, and changes to variable values.

In this chapter, techniques are presented for generating test cases for model checking and constructing proof sketches, which can be used during theorem proving or as proofs by themselves. These techniques are based on deriving permissible timed event sequences, which can then be used to prove the intra-level proof obligations. Many of the techniques can be applied to other real-time specification languages by taking differences in event types into account as well as how to find the next/previous event from a given event.

When an ASTRAL specification has been validated without error, it is ready for formal analysis. The goal of analysis is to provide the maximum assurance that the specification meets its critical requirements. Such a high degree of assurance can be obtained by performing system proofs within an interactive theorem prover. A theorem prover allows the user to discharge a proof obligation using only steps that are known to be sound and requires that each detail of the proof be proved completely. Although a theorem prover does provide the desired level of assurance, using a theorem prover for system proofs can be an extremely time- consuming task. Furthermore, when a design error is found, proofs already performed are not always valid when the specification is changed to fix the error; thus, they must be redone. Given this fact, it is desirable to find as many design errors as possible before invoking the prover, thus requiring a stage before the theorem prover that is less time-consuming. This can be generalized into an analysis hierarchy of multiple stages, where the goal of each stage is to find as many errors in the specification as possible before having to move on to the next stage. Since the goal is to save time, earlier stages should be less time-consuming than later stages so that errors can be fixed with as little wasted effort as possible. Each stage is supported by a set of tools and techniques that can be used to most effectively find errors. Once an error is found, the user must revert back to the design phase to fix the problem and then resume analysis. Depending on the portions of the specification that were changed, the user may be able to resume analysis from where s/he left off or may need to begin analysis from scratch. When the user finishes the last (theorem prover) stage, the analysis of the specification is complete. Throughout the analysis, the specification manager discussed in section 5.7 directs the user to the steps that should be performed next as well as saving which steps have been completed and when.

## 9.1. Test Case Generation

The first stage in the analysis of a specification consists of using the ASTRAL model checker. The ASTRAL model checker [DK 97], which was developed by Zhe Dang of the Reliable Software Group

at UCSB, is a highly automated tool that can check a large number of states for violations in the critical requirements. Figure 9.1-1 shows the model checker window that is brought up by clicking the "ModelCheck" button in the SDE as shown in figure 5.1.

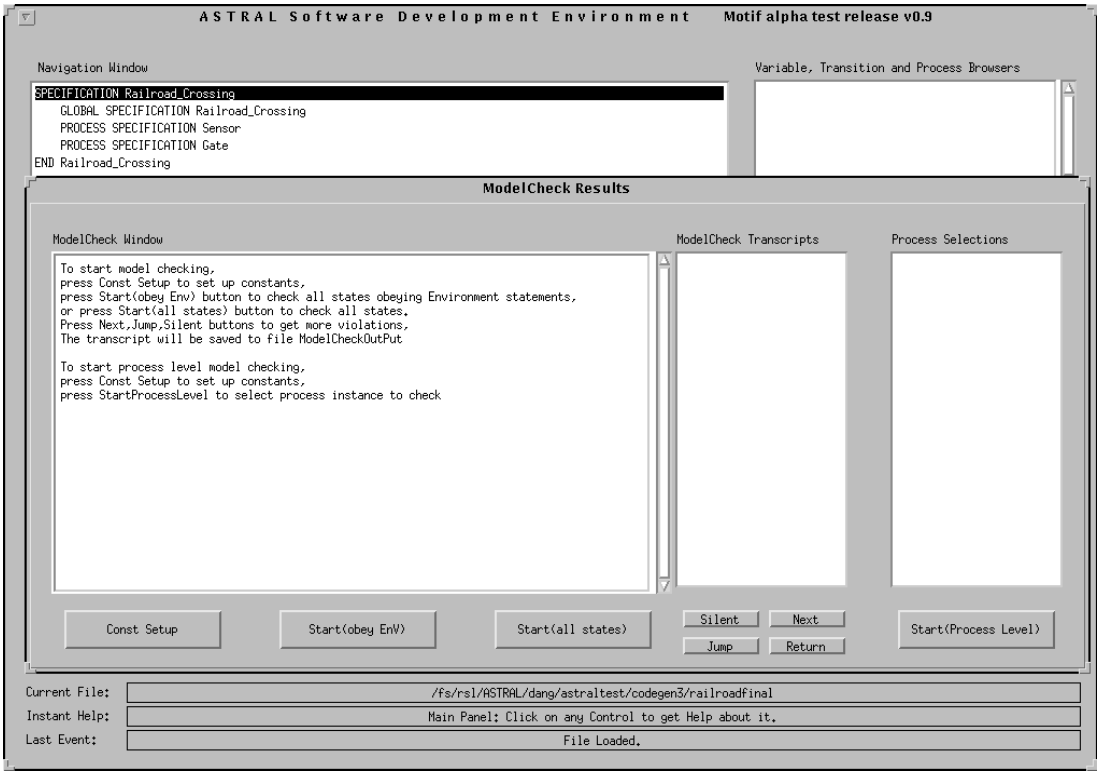



Figure 9.1-1: The model checker window

Since ASTRAL can specify infinite state machines, certain restrictions are needed to limit the number of states explored by the model checker. These include using a discrete rather than dense time domain, restricting the allowed syntax of certain portions of the specification, providing explicit numeric/boolean constants for all symbolic constants, only checking the requirements beginning at the initial state, and providing an explicit number of timesteps to check. Given these restrictions, not all ASTRAL specifications can be model checked. It must first be checked whether the specification meets the syntactic limitations of the model checker. This is done automatically by the SDE. If the specification does not meet the syntactic restrictions, then the model checking portion of the analysis must be skipped. If a specification is syntactically suitable, then the next step is to transform the specification into a form that meets the discrete time domain restriction. This is done by redefining the built-in real type to be an alias type of the built-in integer type. This transformation is performed by the SDE before every model checking session.

Once the dense to discrete transformation has been performed, the model checking procedure is as follows. The user first needs to set up a finite time bound and values for all system or process level constants in the specification by clicking the "Const Setup" button. The time bound indicates the maximal depth that the model checker will search for the current specification. The reason for assigning concrete values to these constants is that currently the model checker can only check a specific instance of the specification. After doing this, the user has two choices for invoking the model checker: "Start(all states)" or "Start(obey Env)". The "Start(all states)" button causes the model checker to enumerate all of the possible states within the time bound and to check that the critical requirements of the specification hold in each state. The "Start(obey Env)" button, in contrast, will check only those states that can be reached by satisfying the environment clauses. The default mode of the model checker is to check the system globally. That is, the actual behavior of every system process is modeled explicitly. The "Start(Process Level)" button allows each process to be checked modularly. That is, only the actual behavior of a particular process is modeled and the behavior of the other processes is assumed from the imported variable clause of that process.

If a failure is detected by the model checker, the transcript window will indicate the actual detailed trace of the states that violated the requirements of the specification. Each state in the trace contains the truth values of all critical requirements, the values of all local variables and the status information of all transition instances for every process instance, as well as other information. The user can easily follow the trace to locate errors in the specification, since each trace is for a single execution branch of the specification and is presented at the specification level. Figure 9.1-2 shows the transcript of a violating trace of states from an earlier version of the railroad crossing specification.

Although [DK 97] describes the design of the model checker and its applications, it does not propose how to generate the test cases for a given specification. It is important to choose the test cases carefully to assure meaningful results and computational feasibility. To illustrate the importance of choosing constants, consider the first portion of the Gate's schedule in the railroad crossing specification:

```
FORALL s: sensor_id
    (   s.train_in_R
    &   now - Change(s.train_in_R) ≥ dist_R_to_I / max_speed - response_time
→       position = lowered)
```

Suppose the user selects the following constants: dist_R_to_I = 100, max_speed = 5, response_time = 1, etc. and the number of timesteps to be 19. The user then runs the model checker and does not find any errors, thus assumes s/he has gained some assurance that the requirement holds. In reality,

however, no assurance has been gained because the constants chosen do not adequately test this requirement. In this case, dist_R_to_I / max_speed - response_time = 19, thus the gate must be down at this time. At first glance, 19 timesteps seems reasonable, but upon closer examination, the earliest time any variable can change value in a discrete time domain is at time one, thus the maximum value of now - Change(s.train_in_R) in this case is 18. In 18 timesteps, however, the requirement is trivially satisfied because the antecedent never holds. Thus, the requirement would hold with the constants given regardless of the actual implementation within the specification. It would appear that this type of problem could be avoided just by setting the number of timesteps to a large value. This is not practical, however, because the running time of the model checker is exponential with respect to the number of timesteps, thus the number of timesteps should only be as large as necessary to ensure that a property is tested adequately.

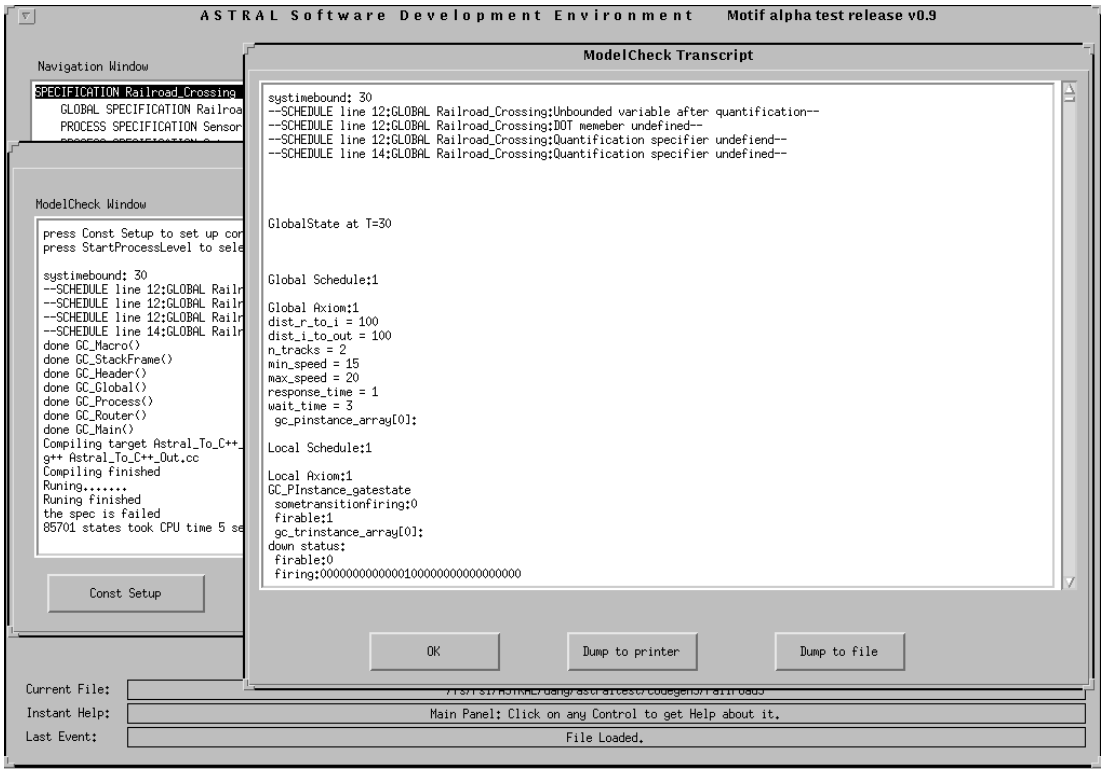



Figure 9.1-2: Transcript of a violating trace of states

Since choosing constants is so important for maximizing the effectiveness of the model checker, it is useful to provide guidelines to the user about how to generate test cases such that the requirements will be tested as adequately as possible. In general, there are two basic types of constants that occur in ASTRAL specifications. One type consists of constants that describe how many objects of

different types there are in the system. The most common examples of this type are constants used in the global processes section to declare the number of process instances of a given process type that exist in the system. For example, in the elevator control system, the "n_floors" constant specifies the number of floors in the building, which in turn specifies the number of Floor_Button_Panel instances. Other examples include the tire circumference in the cruise control system and the number of rounds in the Olympic boxing scoring system. These constants are relatively simple to choose because they rarely affect the correctness of the requirements. That is, system requirements are not usually satisfied by one such constant value and violated by another, other than special values like zero and one. Since additional assurance is not usually gained by choosing large values for these constants, the values should be kept as small as possible so that the execution time of the model checker is minimized. For most specifications, values of two or three should be sufficient. For example, in the railroad crossing, two tracks are sufficient to test the worst case. In the elevator control system, however, as discussed in section 9.2.5, the worst case occurs when the elevator is just about to arrive at a floor when a stop request is made that cannot be serviced before the elevator continues on to the next floor. In this case, the value of n_floors must be set to at least three to adequately test this scenario.

The other basic type of constants are timing constants. These constants include transition durations, response times, etc. as well as the number of timesteps to model check. Some constants are considered to be of this type even if they do not by themselves denote an amount of time. For example, in the railroad crossing specification, the constants dist_R_to_I, max_speed, and min_speed are not in units of time, but when they are taken together as dist_R_to_I / max_speed and dist_R_to_I / min_speed, they represent the minimum and maximum times, respectively, that a train can take to reach the crossing after entering the region. As shown earlier, the values chosen for these types can affect the adequacy of the model checking dramatically. In the earlier example, the constants chosen were not adequate because the antecedent never held so the property was always trivially true. The test case generation guidelines try to ensure that the antecedent will change at least once from false to true and back to false in the chosen time interval. The time period between antecedent changes is significant because the only way an implication can be violated is for the consequent to be false between times the antecedent has changed to true and then back to false. By choosing timing constants appropriately, the consequent can be tested for at least one complete time period in which the antecedent is "enabled". Note that it is not necessary to base the constants on anything in the

consequent because the model checker will generate all possible scenarios and then check each one for violations.

The first step in generating a test case for the model checker is to split the clause of interest (i.e. invariant or schedule) using the formula splitter so that the various tasks to be performed are more manageable. For each of the resulting formulas, there will be one or more sets of constants to test. Although the model checker only checks an entire invariant or schedule, constants can still be derived based on the individual formulas and then run using the clause as a whole, because if one of the split formulas is false, then the entire formula is false by the definition of conjunctive normal form, which is the form generated by the formula splitter. For each split formula, two times must be determined to generate the appropriate test cases. These are the time at which the antecedent is first true (t_true) and the amount of time that the antecedent remains true (dur_true).

### 9.1.1. Determining the Value of T_true

To determine the value of t_true, the events that cause the antecedent to become true must be determined. The simplest case is when the antecedent is true in the initial state of the system. In this case, no events need to occur so the value of t_true is zero. Otherwise, the events that need to occur may include changes to the values of local and imported variables, calls, starts, and ends of local and imported transitions, and delays based on the current time. For example, in the Gate property shown earlier, there must be a sensor S such that there has been a change to train_in_R on S and then a delay of dist_R_to_I / max_speed - response_time.

After the set of events has been determined, the events must be temporally ordered based on the conditions in the antecedent. Once this ordering has been determined, each local variable change and external event must be transformed into a transition event. That is, a transition must be found such that the start or end of the transition implies the event. For local variable changes, a transition must be found that can produce the required change. This can be done with the assistance of the browsers. The user first performs the “variables..used in antecedent of Selected Formula” query from the variable browser menu. This shows all the variables that occur in the antecedent. For each local variable, the user performs the “transitions..using Selected Variable in exit clause” query from the variable browser menu. This query lists the transitions that can change the value of the variable. The user then examines the exit assertion of each transition shown to determine those transitions that can bring about a change of the variable to the appropriate value. For calls to local exported transitions, the call can be replaced by the start of the same transition since a call must have occurred for the transition to start. Finally, for changes to imported variables and calls, starts, and ends of

imported transitions, a transition must be found that implies the required event. In almost all cases, such a transition exists because if no such transition exists, the event in the antecedent is meaningless since that item cannot affect the behavior of the process. The exception is when the event implies some condition in an assumption that is referenced in an entry assertion. Once such a transition is found, the imported event can be replaced by a start of that transition.

For example, consider again the Gate property of the railroad crossing system. Since train_in_R is initially false in all the sensor processes, there must be a change to train_in_R on some sensor s and then a delay of at least dist_R_to_I / max_speed - response_time. The change to train_in_R must be transformed into a transition event. After listing the transitions that reference train_in_R in an entry assertion, the lower transition has the necessary condition. Thus, there must be a start of lower and then the appropriate delay.

Once each event has been transformed, the sequence generator is used to find the shortest reasonable transition sequence forward from the initial state of the system in which all the transition events occur in their proper order. Since the sequence generator is not completely precise due to undecidability issues, a reasonable sequence is one that seems feasible based on the user's knowledge of the specification. Transition durations are chosen by the user during model checking, thus the shortest sequence will most often be the sequence that has the fewest number of transitions. As discussed in section 8.5.5, an estimate of the running time of sequence output by the sequence generator is displayed next to the sequence and consists of transition durations and three types of delay. A delay_E refers to a delay due to the external environment. A delay_O refers to a delay caused by other processes in the system. Finally, a delay_T refers to a delay due to a restriction on the current time. After the sequence is found, the value of t_true is found by replacing every delay_E, delay_O, and delay_T in the running time estimate of the sequence with a concrete value or a value based on the timing constants of the system. This is done by examining the entry assertion of the transition(s) incurring the delay and determining the events that need to occur in the environment, other processes, or by the passage of time as described in the following two sections.

*9.1.1.1. Delay_E and Delay_O Events*

To find delay_E and delay_O substitutions, the user must first check whether the property is from the schedule or from the invariant. If the property is from the schedule, the user must check for assumptions about when the relevant events can occur. For changes to an imported variable, the user selects the transition that incurs the delay by using the transition browser and performs the "variables..used in entry clause of Selected Transition" query. This shows all the variables that occur

in the transition entry assertion. For each imported variable that is relevant, the "formulas..using Selected Variable" query should be performed from variable browser menu. This query lists the formulas in the formula splitter that reference the selected variable.

For calls, starts, and ends of imported transitions, the user first performs the "transitions..used in antecedent of Selected Formula" query from the formula splitter query menu with the property in the formula splitter window. This shows all the transitions that occur in the antecedent. The user then finds the imported transition or local exported transition of interest and performs the "formulas..using Selected Transition" query from transition browser menu. This query lists the formulas in the specification that reference the selected transition.

For each assumption listed, the user must determine if it is relevant to the transition entry assertion and if so, what limitations are placed on when the given event can occur in terms of other events. If there are any such limitations, the restrictions on these events must be determined in a similar fashion. If no such assumptions exist, it can be assumed that the delay is zero in all cases unless the delay occurs immediately after the initial state. In this case, it can be assumed that calls and starts of imported transitions have a delay of zero, but that ends of imported transitions and changes to imported variables have a delay of up to the maximum duration of any transition in the process the item was imported from. The maximum duration of a process P can be represented by a symbolic value max_dur(P), which will be used later. The zero delay assumption is almost always valid because the model checker checks all possible operating environments, thus if an error is found in a scenario in which an event is delayed, it will almost certainly be found in the scenario in which the event occurs as early as possible. This assumption is not valid, however, when system behavior is conditionally controlled by absolute time references to an event (e.g. IF Call(tr1, 0) THEN ... ELSE IF Call(tr1, 1) THEN ... ELSE ... FI), but very few systems have such references.

In the Gate property, the sequence generator is used to generate the sequences of transitions from the initial state to lower. The shortest most reasonable sequence is "<INITIAL>, lower" with a running time of "delay_O + lower_dur". After using the browsers to find assumptions relevant to the entry assertion, no assumptions are found that are pertinent to the entry assertion. Thus, delay_O is set to max_dur(Sensor) since it is a change from the initial state. There must then be a delay of dist_R_to_I / max_speed - reponse_time. Thus, the value of t_true is max_dur(Sensor) + dist_R_to_I / max_speed - response_time. Note that lower_dur was dropped because only the start of lower was required and not the end.

*9.1.1.2. Delay_T Events*

As discussed in section 8.1, delays based on the current time are often in a form such as "now - End(tr1) ≥ delay1" or "now - Change(v) ≥ delay1" where v is a variable set by a transition tr1. In these cases, delay_T can be set to delay1. These delays can be found by inspecting the entry assertion of the transition incurring the delay.

For example, consider the following schedule property of the P_Robot process of the production cell.

```
    robot_status = arm1_at_press
→ (  press.press_status ≠ at_upper
    &  press.press_status ≠ moving_to_upper
    |  arm1_status = retracted)
```

In the initial state of the P_Robot process, robot_status = arm2_at_deposit, thus a change to a local variable needs to occur. The variables of the antecedent are brought up and the "transitions..using Selected Variable in exit clause" query is performed. The only transition that sets robot_status to arm1_at_press is Arm1_Arrived_At_Press. The sequence generator is then used to determine the sequences from the initial state to Arm1_Arrived_At_Press. The shortest and most reasonable sequence is "<INITIAL>, Rot_Arm1_CCW_To_Press, Arm1_Arrived_At_Press" with a running time of "rotate_arm_dur + delay_T + arm_arrive_dur". The only delay in the running time is a delay_T from the entry assertion of Arm1_Arrived_At_Press. After examining this entry assertion, delay_T is found to be t_move_arm1_to_press. Thus, the value of t_true is rotate_arm_dur + t_move_arm1_to_press + arm_arrive_dur.

### 9.1.2. Determining the Value of Dur_true

The formula splitter displays split formulas in the form $(A_1 \,\&\, ... \,\&\, A_n) \rightarrow (C_1 \mid ... \mid C_n)$, similar to the form used by PVS. Since all terms in the antecedent are conjoined, only a single term needs to become false for the antecedent as a whole to become false. For the most part, the user may select a change to any term to determine the value of dur_true. After an appropriate event has been selected, the techniques for that event type from the previous section are used to determine the delay from that event back to when the antecedent became true.

For example, in the Gate property, the sensor that detected the train must change to ~train_in_R (and no other sensor can be reporting a train). A change to an imported variable corresponds to finding the value of a delay_O. After listing the variables used in the antecedent and displaying the relevant assumptions, an imported variable assumption is found stating that a change to ~train_in_R can only occur after (dist_R_to_I + dist_I_to_out) / max_speed time has elapsed from the change to

train_in_R. The change to train_in_R was assumed to have occurred when lower fired, thus the earliest the change could occur is at max_dur(Sensor) + (dist_R_to_I + dist_I_to_out) / max_speed. The antecedent became true, however, at max_dur(Sensor) + dist_R_to_I / max_speed - response_time, thus dur_true = max_dur(Sensor) + (dist_R_to_I + dist_I_to_out) / max_speed - (max_dur(Sensor) + dist_R_to_I / max_speed - response_time) = dist_I_to_out / max_speed + response_time.

For the P_Robot property, robot_status needs to change from arm1_at_press. In this case, the technique for local variable changes and transition starts and ends is used. The transition that changes robot_status from arm1_at_press is the Rot_Arm1_CW_To_Table transition. Thus, the transition sequences are generated between Arm1_Arrived_At_Press and Rot_Arm1_CW_To_Table. The shortest reasonable sequence is "Arm1_Arrived_At_Press, Extend_Arm1, Arm1_Extended, Arm1_Drop, Retract_Arm1, Arm1_Retracted, Rot_Arm1_CW_To_Table" with a running time of "arm_arrive_dur + delay_O + move_arm_dur + delay_T + arm_moved_dur + delay_O + arm_object_dur + move_arm_dur + delay_T + arm_moved_dur + rotate_arm_dur". The two delay_T's are both found to be t_move_arm. For the first delay_O, no pertinent assumption clause is found, thus the delay can be assumed to be zero. For the second delay_O, an imported variable assumption is found stating that the press only moves from its middle position after arm1 drops a blank. Since the press was at its middle position when arm1 was extended and the blank has not yet been dropped at the time Arm1_Drop is to fire, the press must still be at its middle position, thus this delay is also zero. Thus, dur_true = arm_arrive_dur + move_arm_dur + t_move_arm + arm_moved_dur + arm_object_dur + move_arm_dur + t_move_arm + arm_moved_dur + rotate_arm_dur.

### 9.1.3. Deriving the Time Bound

In order to generate the test cases, a system of inequalities is constructed based on t_true and dur_true. The system consists of the inequalities from the local and global axiom clauses, the constant refinement clause, and an inequality "timesteps ≥ t_true + dur_true", where each max_dur(P) is replaced by max({Duration(tr1) | tr1 is a transition of P}). The solutions to the resulting system of inequalities describe sets of constants that can be used to test the given property. Since multiple solutions will exist, the user has some leeway as to what the chosen values of the constants can be. This has several consequences. First, the user still must choose actual values for the constants. The inequalities, however, will rarely be complex enough to warrant an inequality solver. In general, the number of timesteps must be kept as small as possible to minimize running

time. This means that the constants in the main timestep inequality must be instantiated with this in mind. It may be that it is not possible to keep the number of timesteps to a computationally reasonable value. For example, it may be that the amount of time needed for the antecedent to become false makes the number of timesteps impractical to perform within the model checker. If this is the case, the number of timesteps can be reduced so that only a partial interval of the antecedent being enabled is tested. If this is not the case, that particular set of inequalities may need to be skipped.

The main consequence of multiple inequality solutions is that it offers the possibility of testing several different varieties of configurations while still ensuring that the resulting scenarios will test the requirements as adequately as possible. One such configuration would be a "normal" or "reasonable" configuration. In this case, the behavior of the system can be tested with constants that are likely to occur in an actual implementation of the system. Other configurations would include "extreme" configurations in which response times are set as low as the inequalities allow and transition durations are set as high as the inequalities allow and vice-versa. The purpose of these configurations is to try and violate liveness requirements with a small "response interval" and large delays, and to violate safety requirements with large intervals and small delays. This configuration is well suited to check the specification for missing axioms that would disallow such configurations. Transition durations can also be set large or small one by one to try to pinpoint which durations are the most important to limit.

Besides testing for missing axioms, it is also possible to use the model checker to check for missing or inadequate imported variable assumptions by using the model checker's local/global option. When using the model checker, the user can choose between checking a process locally (i.e. using the actual behavior of the process and the assumed behavior of the other processes) or globally (i.e. using the actual behavior of all processes). Suppose the user checks a process with the same set of constants both locally and globally. If no violations are found using local checking, but are found using global checking, then the implementations of the other processes are not meeting their behavioral assumptions in the imported variable clause. On the other hand, if violations are found using local checking, but are not found using global checking, then the imported variable assumptions are too weak of an abstraction of the actual behavior to prove the requirements.

For the Gate property, the resulting system of inequalities is shown below.

- timesteps ≥ max({enter_dur, exit_dur}) + dist_R_to_I / max_speed + dist_I_to_out / max_speed - response_time
- max_speed ≥ min_speed
- response_time < dist_R_to_I / max_speed
- wait_time ≥ raise_dur + raise_time + up_dur
- dist_R_to_I / max_speed ≥ response_time + lower_dur + lower_time + down_dur + raise_dur
- dist_R_to_I / max_speed ≥ response_time + lower_dur + lower_time + down_dur + up_dur

The constants used in [KDK 99] were dist_R_to_I = 100, dist_I_to_out = 100, min_speed = 15, max_speed = 20, wait_time = 3, timesteps = 25, and the remaining constants set to one. The following are the results of substituting the chosen constants into the inequalities.

- 25 ≥ 1 + 5 + 5 - 1 or 25 ≥ 10
- 20 ≥ 15
- 1 < 5
- 3 ≥ 1 + 1 + 1 or 3 ≥ 3
- 100/20 ≥ 1 + 1 + 1 + 1 + 1 or 5 ≥ 5
- 100/20 ≥ 1 + 1 + 1 + 1 + 1 or 5 ≥ 5

Since all of the inequalities hold, the chosen constants were adequate for testing the first portion of the Gate property. The system of inequalities for the P_Robot property is shown below.

- timesteps ≥ rotate_arm_dur + t_move_arm1_to_press + arm_arrive_dur + arm_arrive_dur + move_arm_dur + t_move_arm + arm_moved_dur + arm_object_dur + move_arm_dur + t_move_arm + arm_moved_dur + rotate_arm_dur
- feed_length > blank_length + blank_length + feed_response * feed_speed
- deposit_length > blank_length + blank_length + deposit_response * deposit_speed
- table_response ≤ rotate_arm_dur

## 9.2. Proof Sketch Construction

Once the model checker has been used to gain as much assurance as possible that the critical requirements of the specification hold, the user must next construct a *proof sketch* for the system. In this stage, the user attempts to derive a general plan of how the proofs of the various system properties are to be carried out. The sketch is not intended to be a full formal proof of the system, but should cover the crucial aspects of each proof. These include how each proof can be broken down into cases, what axioms, environmental assumptions, imported variable assumptions, and inductive invariant/schedule assumptions in the specification are needed in the proof of each case, etc. Like the test case generation stage, the purpose of this stage is to attempt to uncover design errors before resorting to the theorem prover. By determining the cases needed for each proof and examining each scenario individually, there is a strong possibility that the user will discover errors such as inadequate and/or missing assumptions necessary to complete the proof. Although not required to be a full

proof, the more work that is put into this stage, the more likely it is that errors will be found before the much more time-consuming theorem prover stage is attempted. If no errors are found in this stage, the work is not wasted because to discharge the proofs within the theorem prover, it is necessary to split up the proof into cases and set them up within the prover. By having a breakdown of the cases, assumptions needed for each, and a general strategy for proving each, the user can focus almost exclusively on the "tactical" side of using the theorem prover (i.e. actually carrying out each strategy with prover commands).

Attempts have been made to use classification information to direct the proofs of real-time systems. The approach that is most similar to the techniques below is in [HMP 94], as discussed in section 4.2.4. In this paper, two different specification styles are identified and different proof techniques are presented for each. The proof rules for proving bounded-invariance and bounded-response properties correspond to the timed safety and timed liveness classifications, respectively, of section 8.3. There is also some mention about multi-threaded processes and scheduling policies, which are discussed in section 9.2.6.2. The authors only discuss how such systems can be specified as timed transition systems, however, and not how to identify such processes given an arbitrary specification nor how the proofs of such systems differ from the proofs of other system types.

### 9.2.1. Proof Ordering

A critical factor in the construction of the proof sketch and the later theorem proving stage is that the order in which proofs are performed significantly affects the amount of work that must be redone in the case of specification errors. In general, this is an issue for any formal language with a modular proof system and is not specific to ASTRAL. In ASTRAL, this problem can be illustrated by a process P1 with a schedule clause P1.SCH that depends upon an imported variable clause P1.IV, which in turn depends on the invariant clause of a process P2, P2.INV. In this system, there are three proof obligations, which correspond to the proof obligations for P1.SCH, P1.IV and P2.INV. To illustrate the worst case scenario, suppose the user decides to prove P2.INV first. Furthermore, suppose the proof of P2.INV fails so the user changes the transitions of P2 until P2.INV is satisfied. The user next proves P1.IV. Suppose P2.INV is not strong enough to prove P1.IV so P2.INV must be changed, meaning that the proof of P2.INV must be redone. Now the proof of P1.SCH is performed. Again, suppose P1.IV is not strong enough to prove P1.SCH so P1.IV must be changed, which in turn requires a change to P2.INV. Thus, the proof of P2.INV must be redone, then the proof of P1.IV, then the proof of P1.SCH. In this worst case, six proof obligations must be discharged.

In the optimal proof sequence, P1.SCH would be proved first. The user would discover that P1.IV

was not strong enough and change it accordingly to finish the proof of P1.SCH. In the proof of P1.IV, the user would determine that P2.INV was not strong enough and change it accordingly to finish the proof of P1.IV. Finally, the user would discover that the transitions of P2 did not preserve P2.INV and change them accordingly to finish the proof of P2.INV. In this sequence, the user must perform three proofs, which is half the number required in the worst case.

In general, for a system with n proof obligations with a totally ordered dependency graph, the optimal number of proofs to be performed is n and the worst case is n * (n + 1) / 2, with even worse cases possible for a partially ordered dependency graph. Since ordering can affect the number of proofs that need to be performed so dramatically, it is to the user's advantage to determine the optimal ordering before proofs begin. For this reason, the first step in the construction of a proof sketch is to build a hierarchy of proofs that can be consulted to determine the proof obligation that should be attempted next. The crucial assumption that is made to determine optimality is that the user states critical requirements correctly and that process execution (i.e. transition and initial state declarations) must be changed to meet the requirements. That is, the user does not change requirements based on process execution. A natural hierarchy of proof obligations is based on which clauses may be referenced in the proofs of other clauses. This hierarchy is shown in figure 9.2.1-1. An edge from a node i to a node j indicates that the proof of the clause associated with i can reference the clause associated with j. Note that only the relationships between P1 and the global specification are shown and the inductive assumptions are not shown.

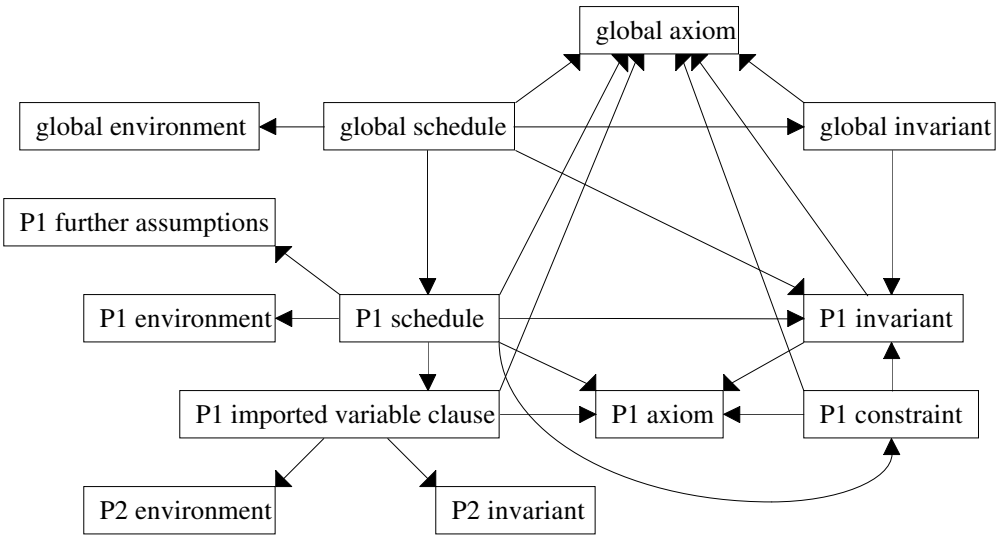



Figure 9.2.1-1: Proof obligation relationships

In ASTRAL, a proof ordering can be computed automatically that is optimal for totally ordered dependency graphs and that uses a heuristic for partially ordered graphs. To compute this ordering, a dependency graph between proof obligations is first built using the graph of figure 9.2.1-1 as a starting point. The graph is expanded, however, to include the clauses of all processes and the relationships between each imported variable clause and the invariant and environment clauses of the other processes. An edge is only added between the imported variable obligation of a process $P_i$ and the environment and invariant of a process $P_j$ if $P_i$ imports a variable or transition from $P_j$. The resulting graph is either a partial or total ordering among the clauses of the specification. Initially, each node in the graph is considered "unmarked". The sequence of proofs is obtained by selecting any unmarked node i that is not a dependency of any unmarked node j. If more than one such node exists, then the graph is partially ordered and any such node may be selected. Nodes that are associated with "non-empty" clauses (i.e. clauses that are not the expression "TRUE"), however, should be chosen before nodes associated with empty clauses. After the proof of i is complete, i is "marked" as complete and the procedure continues until every node that is associated with a proof obligation has been marked. Any time a clause corresponding to a node i is changed during the course of a proof obligations, any node j that is directly dependent on i is unmarked.

If a total ordering exists among proof obligations or a partial ordering with choices only between non-empty and empty clauses, then the above algorithm produces the optimal sequence of proof obligations because exactly n obligations must be proved. If only a partial ordering exists, however, then optimality cannot be guaranteed. In this case, however, determining the optimal ordering is undecidable since it depends on whether or not each proof obligation is provable, which from appendix B, is undecidable. For example, let P1 and P2 be two processes with a proof obligation relationship as shown in figure 9.2.1-2. In this case, the schedules of P1 and P2 are partially ordered, thus a guess must be made as to which one should be attempted first. Suppose the algorithm guesses that the schedule obligation of P1 should be attempted first. Furthermore, suppose that the schedule of P1 is initially provable, but the schedule of P2 is initially unprovable due to a weak imported variable clause and the imported variable obligation of P2 is unprovable due to a weak invariant of P1. P1.SCH is attempted first and succeeds and then P2.SCH is attempted and fails. Since P2.SCH fails due to a weak imported variable clause, P2.IV must be changed to strengthen it. Since P2.IV was initially unprovable due to a weak invariant of P1, P1.INV cannot be strong enough to prove the strengthened P2.IV, thus must be changed. Since P1.INV must be changed, the P1.SCH must be reproved, resulting in a waste of the initial proof of P1.SCH that succeeded.

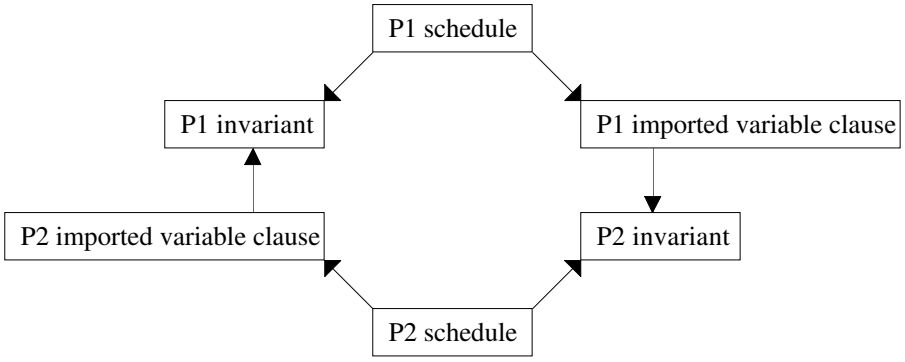



Figure 9.2.1-2: Partially ordered proof obligation relationship

If it was known whether a proof obligation was provable or not initially, an optimal proof sequence for partially ordered proof obligations could be determined by always picking obligations that are not provable before obligations that are provable. In the example above, this would mean proving P2.SCH before P1.SCH. As mentioned above, however, this is an undecidable problem, thus the algorithm described above is the best possible. To try to minimize the effects of selecting the wrong node among a choice of partially ordered nodes, a heuristic is used that selects nodes of processes with the fewest number of transitions first. The rationale for this is that the proofs of processes with fewer transitions are simpler than the proofs of processes with more transitions, thus if a wrong ordering is selected, less work must be redone if the simpler processes are selected first. As an example of a proof ordering, consider the railroad crossing system. In this system, the computed proof ordering is global.SCH, Sensor.SCH, Gate.SCH, Gate.IV, and Sensor.INV. After the appropriate proof sequence is constructed, the proof sketch can be constructed using the techniques discussed in the following sections.

### 9.2.2. Transition Steps

The guidelines for proof sketch construction are based on finding the transition sequences that are possible in a given process type. All ASTRAL requirements are based on the current time, the values of local variables, the call, start, and end times of local transitions, the values of imported variables, and the call, start, and end times of imported transitions. From a sequence of transitions, all of this information can be derived, thus any ASTRAL requirement can be proven (if possible) by analyzing transition sequences. The start and end times of local transitions can be found directly from the sequence. The values of local variables can be derived from the entry and exit assertions of each transition in the sequence. The values of imported variables and the call, start, and end times of imported transitions can be derived from the imported variable clause using the values of exported

local variables and the start and end times of exported local transitions. The call times of exported local transitions can be derived from the environment clause using the values of exported local and imported variables and the start and end times of exported local and imported transitions. Finally, a symbolic value for the current time can be derived from the sequence using the other information and the entry assertion of each transition.

A transition step is computed from any arbitrary state and can be any combination of forward or backward, timed or untimed, and conditional or unconditional. The result of a transition step is set of transition-delay pairs if the step is timed or a set of transitions if the step is untimed. If the step is forward, then the resulting set indicates the transitions that can fire immediately after the given state. If the step is backward, then the resulting set indicates the transitions that can fire immediately before the given state. In this case, "immediately" refers to that fact that no other transition will fire between the given state and a transition from the set. If the step is timed, the delay interval indicates the minimum and maximum times from the given state until the transition fires. If the step is untimed, the delay was not taken into account and could potentially be any value. Conditional steps are computed using information from all the clauses of the process, while unconditional steps are computed using only the information from the invariant and axiom clauses. All conditional steps have an unconditional counterpart, while the reverse is not necessarily true. This is because the assumptions specified in the process may eliminate certain scenarios but cannot add any scenarios that are not also possible in an arbitrary operating environment. Similarly, the delay interval of any conditional step will be a subinterval of the delay interval of its unconditional counterpart.

It is first necessary to determine which transitions can fire next from the given state for forward steps or could have fired last from the given state for backward steps. If the given state references a start or end of a transition, the browsers can be used to determine this information. The "transitions..following Selected Transition" and "transitions..preceding Selected Transition" browser queries use the successor information of the sequence generator to display the appropriate transitions. If the given state does not reference a start or end, however, these queries cannot be used. For an arbitrary state, it is first necessary to determine the values of the local state variables. If the state does not contain any local process information, then it is necessary to find the assumptions that are relevant to the state to obtain the information for conditional steps. In the case of unconditional steps, this means that any transition may be a successor or predecessor of the given state. Once the values of the local state variables have been determined, the possible successors can be computed by

examining the transitions that reference the variables in their entry assertions. To find the possible predecessors, the transitions that reference the variables in their exit assertions must be inspected.

The list of possible successors or predecessors displayed by the transition browser or obtained manually must be inspected to eliminate possibilities that cannot actually occur because of conditions in the known state. For example, it may be that a transition is a possible successor in one situation but not in another. The impreciseness of the sequence generator as discussed in section 8.5.3 is also an issue. To eliminate these possibilities in both forward and backward steps, it is necessary to step backward from the given state or the derived transition, respectively. In forward steps, it is necessary to step backward to further clarify the state of the process at the given state. In backward steps, it is necessary to step backward to make sure that the given state is still possible from the computed predecessor's predecessor.

Once the list of possible successors or predecessors has been computed, it is necessary to compute the delay interval for timed steps. For forward steps, the delay is based on the transition classification of the successor transition. The classification of the appropriate transition can be displayed using the "transition information" query in the transition browser. This query displays the factors that the enablement of a transition is dependent on. For unconditional steps, the delay interval is [0, ∞] for any transition that depends on other processes or the environment. For conditional steps involving these transitions, the assumptions that are relevant to the transition's entry assertion must be inspected to determine when the appropriate events may occur with respect to the given state. These can be found by first invoking the formula splitter on the transition's entry assertion and then performing the "formulas..using items used in Selected Formula" query in the formula splitter. This query brings up the relevant formulas in the formula splitter. If no pertinent assumptions are found, the delay interval is [0, ∞]. For transitions that only depend on local variables and transitions, the delay interval is [0, 0]. Finally, for transitions that depend on the current time, the delay can be found as discussed in section 9.1.1.2. For backward steps, the procedure is similar except that the transition associated with the given state is first classified. If no transition is associated with the given state, then the assumptions relevant to the state must be examined to determine if there are any assumptions that restrict when the predecessor can occur.

The main distinction between forward and backward steps is that when a forward step is made, all possible nondeterministic choices that can be made in the system, such as events occurring or not occurring in the external environment and choices between which transition will fire, must be considered. When a backward step is made, however, the path along which the system has evolved

has already been determined, thus nondeterminism is usually less of a factor. This difference between forward and backward steps can be seen in the portion of an execution tree of a process as shown in figure 9.2.2. The tree shows transitions occurring at different points in time where branches indicate possible choices in the process's execution. When a forward step is made, a move is made from a root node to one of several branches. When a backward step is made, however, a move is made from a leaf node to a root node.

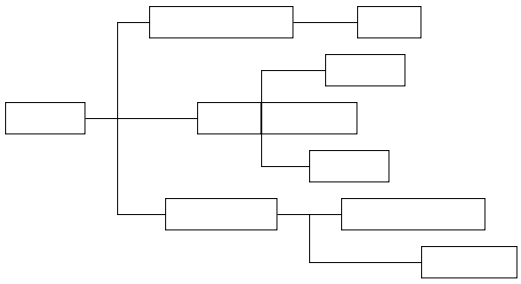


Figure 9.2.2: The execution tree of a process

As a more concrete example, consider a forward timed unconditional step computed from the end of the raise transition in the Gate process of the railroad crossing. Using the sequence generator, the possible successors to raise are up and lower. After analyzing the entry assertion of up, it is seen that up will fire exactly at End(raise) + raise_time if no other transition disables raise, thus the delay interval is [raise_time, raise_time]. For lower to fire, there must be a sensor that starts reporting a train after raise starts. This change must occur sometime at or before End(raise) + raise_time, however, or else up will fire. Thus, the delay interval is [0, raise_time].

Now consider a backward timed unconditional step computed from the start of the up transition in the Gate process. Using the sequence generator, the only possible predecessor of up is raise. From the previous example, up occurs exactly raise_time from the end of raise, thus the delay interval is [raise_time, raise_time]. In the previous case, there were two possible successors to raise based on the behavior of the operating environment. In this case, however, it is known that no trains were detected in the region by the fact the up had fired. There is also another difference between forward and backward steps as illustrated by this example. In the forward case, since the process was idle after raise_time had elapsed from the end of raise and some transition was enabled (up), some transition *must* fire at that time. In the backward case, however, it is necessary to show more. That is, not only must it be shown that the process is idle and that the predecessor was enabled, it is also necessary to show that the predecessor *was not* enabled any earlier. In the up case, this condition

holds because if raise was enabled earlier, it would have fired earlier, thus up would have fired earlier, which is a contradiction.

### 9.2.3. Global and Imported Variable Obligations

The proofs of ASTRAL specifications are performed modularly. In order to prove a global property, it is not necessary to examine the exact executions of each process instance in the system. Rather, the local properties of each process type must imply the global properties in the system. Similarly, the exported portions of the local invariants of each process must imply the imported variable clause of another process. In almost all cases, these proofs are trivial to perform because the actual execution of individual processes cannot be used in the proofs, thus the clauses must be simple implications that are proved by instantiation.

For example, consider the global schedule obligation of the bakery algorithm. For this obligation, it is necessary to prove that the global schedule holds at all times. The global invariant, global environment, and exported portions of the local invariants and schedules from all processes in the system can be used to prove this obligation. The global schedule of the bakery algorithm states that only one Proc process may be in its critical section at any given time.

```
FORALL i, j: procs_int
    (   procs[i].in_critical
    &   procs[j].in_critical
→       i = j)
```

An exported portion of the schedule of the Proc process that is generated for the proof obligations is shown below.

```
FORALL PID1: id
    (   Id_Type(PID1) = Proc
→       (   PID1.in_critical
    →       FORALL i, j: procs_int
                (   procs[j] = PID1
                →       procs[i].number = 0
                    |   PID1.number < procs[i].number
                    |   PID1.number = procs[i].number
                    &   j < i))
```

This property is not enough to prove the global schedule. Additionally, the following exported portion of the invariant of the Proc process is needed.

```
FORALL PID1: id
    (   Id_Type(PID1) = Proc
→       (   PID1.in_critical
    →       PID1.number ~= 0))
```

Suppose both procs[$i_0$].in_critical and procs[$j_0$].in_critical hold for $i_0 \neq j_0$. From the exported portion

of the Proc invariant, neither procs[$i_0$].number nor procs[$j_0$].number is zero. Suppose procs[$i_0$].number = procs[$j_0$].number. From the exported portion of the Proc schedule for PID1 = procs[$i_0$], it is known that $j_0 < i_0$. Similarly, for PID1 = procs[$j_0$], it is known that $i_0 < j_0$, which is a contradiction. If procs[$i_0$].number ≠ procs[$j_0$].number, then a similar contradiction can be achieved. Thus, the global schedule of the bakery algorithm holds.

### 9.2.4. Simple Single-Threaded Processes

The techniques presented in the following sections are for the local properties of simple single-threaded processes. These techniques are also the foundation upon which the more complex techniques for iterative single-threaded processes and multi-threaded processes are based. The techniques are presented according to the property classification that each is associated with.

#### *9.2.4.1. Untimed Properties*

Although in principle, untimed properties are the simplest of all the property classifications, in reality, the proof of an untimed property can be just as complex as the proof of any timed property. This is due to a variety of factors. One factor is that ASTRAL is expressive enough to allow any timed property to be specified as an untimed property. This is accomplished for a timed property P by first introducing a boolean variable v_P, which is initially true. Then, a transition tr_P is introduced with an entry assertion of "EXISTS t: time (t ≤ now & ~past(P, t))" and an exit assertion of "~v_P". P is then replaced in the invariant or schedule with the expression "v_P = TRUE". If the new invariant or schedule can be proved, then it is known that v_P is true at all times. Since v_P is always true, tr_P never fires, which, by the entry assertion, implies that there is never a time at which P is violated. In this case, the proof that v_P is always true is essentially identical to the proof of P. Although it is possible that timed techniques may be required to construct the proof sketch of an untimed property, in general, the proof sketch of almost all untimed properties that occur in practical specifications can be constructed using only the untimed techniques presented below.

A property "A → C" can only hold if for any interval I in which A holds, there is an interval in which C holds that contains I as shown in figure 9.2.4.1-1.

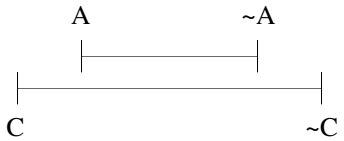



Figure 9.2.4.1-1: Property holds

There are two ways in which a property can be violated. The first violation type occurs when A changes to true while C is false as shown in figure 9.2.4.1-2.

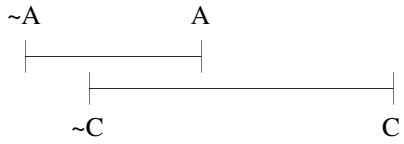



Figure 9.2.4.1-2: Violation type 1

The second violation type occurs when C changes to false while A is true as shown in figure 9.2.4.1-3.

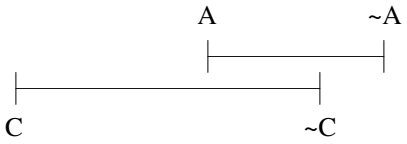



Figure 9.2.4.1-3: Violation type 2

The first two techniques are for untimed properties that only reference local state variables. That is, they do not contain pasts, starts, ends, call, changes, or references to imported variables. For these untimed properties, changes to A and C can only occur when some transition of the process ends. For properties that reference more than just local state variables, a third technique is presented.

*9.2.4.1.1. Transition Entry/Exit Analysis*

The first technique is based on the proof obligations for the ASLAN language, which are discussed in section 3.2.1, and are intended for use with untimed properties that only reference local state variables. To prove a property in ASLAN, the proof obligation "property′ & entry′ & exit → property" must be proved for each transition in the specification, where a prime indicates the value an expression had when the entry assertion of the transition held. This obligation states that if a property holds when the entry assertion holds, then it must hold when the exit assertion holds. This proof obligation can be used for invariant properties and without the "property′", can also be used for constraint properties. Unlike ASLAN, which is based on an untimed model, ASTRAL is based on a timed model, so properties must be proven at all times. By the vars_no_change axiom, however, variables can only change at the end of a transition. Thus, if a property can be proven at those times, it holds for all possible times.

As mentioned, the only way for such an untimed property "A → C" to be violated is for A to become true while C is false or for C to become false while A is true. Thus, the first technique is to identify the transitions that make C false or A true and to verify that such a violation cannot occur. To find these transitions, the browsers can be used. First, the user brings up the property being proved in the formula splitter. For the transitions that make C false, the "variables..used in consequent of Selected Formula" query is performed. For the transitions that make A true, the "variables..used in antecedent of Selected Formula" query is used. Once the variables are displayed in the variable browser window, the "transitions..using Selected Variable in exit clause" query is performed. The resulting list of transitions contains those transitions that can cause the property to be violated. Any transition that is not listed does not require any further analysis as it does not reference the variables of A or C in its exit assertion, thus cannot violate the requirement. Each transition in the transition browser window is then double clicked on to show the transition in the navigation window. The user must manually inspect the exit assertion to determine whether the transition sets A to true or C to false.

Once the user has the list of transitions that can set A to true or C to false, it must then be shown that each exit assertion implies C is true or A is false, respectively. This can occur in one of three ways.

- C is true or A is false by the exit assertion. For example, in the property "maintaining_speed → cruise_on" of the Speed_Control process of the cruise control system, the disable_cruise transition sets C to false, but A is also false by the exit assertion, so the property is preserved.

```
TRANSITION disable_cruise
   ENTRY      [TIME:   input_dur]
         cruise_on
   EXIT
         ~cruise_on
      &  ~maintaining_speed
      &  ~increasing_speed
      &  throttle = foot_throttle′
```

- C is true or A is false because it holds by the entry assertion and C or A is not changed in the exit. For example, in the same property as above, the maintain_speed transition sets A to true, but C is true because cruise_on is true in the entry assertion and unchanged in the exit assertion, so the property is preserved.

```
TRANSITION maintain_speed
   ENTRY      [TIME:   input_dur]
         cruise_on
      &  ~maintaining_speed
   EXIT
         cruise_throttle = throttle′
      &  desired_speed = the_speedometer.speed
      &  maintaining_speed
```

- C is true or A is false because the entry assertion implies a condition that implies C is true or A is false from the invariant (or schedule). For example, if the property "increasing_speed → cruise_on" were being proved in the same process as above, the begin_speed_increase transition sets increasing_speed to true, but maintaining_speed is true in the entry assertion, which from the property "maintaining_speed → cruise_on", implies that cruise_on is true in the entry. Since cruise_on is unchanged in the exit assertion, the property is preserved.

```
TRANSITION begin_speed_increase
   ENTRY       [TIME:    input_dur]
               maintaining_speed
          &    ~increasing_speed
   EXIT
               increasing_speed
          &    desired_speed = desired_speed′ + speed_step
```

Thus, if one of the three conditions above holds for the list of transitions obtained by the user, then the property holds. As an example of the use of this technique, consider the property "maintaining_speed → cruise_on" of the Speed_Control process of the cruise control system. The only transitions that set maintaining_speed are maintain_speed and resume_speed. Both of these transitions require cruise_on in their entry assertions and do not reset cruise_on. The only transition that sets ~cruise_on is disable_cruise. The exit assertion of disable_cruise also sets maintaining_speed to false. Thus, the property holds.

*9.2.4.1.2. Transition Sequence Analysis*

Transition entry/exit analysis is not guaranteed to completely prove untimed invariant properties and is almost never enough to completely prove untimed schedule properties. If none of the three conditions discussed in the previous section can be met for a transition that sets A to true or C to false, further analysis must be performed. For each transition tr1 that sets A to true, it is necessary to show that every sequence backwards from tr1 contains a transition that asserts C is true more recently than a transition that asserts C is false. Namely, it is checked that the situation in 9.2.4.1-1 holds and not the situation in 9.2.4.1-2. Similarly, for each transition tr1 that sets C to false, it is necessary to show that every sequence backwards from tr1 contains a transition that asserts A is false more recently than a transition that asserts A is true. In this case, it is checked that the situation in 9.2.4.1-3 does not hold.

Thus, transition sequence analysis is the process of making untimed backward steps to determine whether or not a violation can indeed occur. This analysis continues from the transition entry/exit analysis of the previous section. For each transition tr_A that can change A to true that did not meet

the entry/exit criteria, the following steps must be performed. First, the list of transitions that assert C is true or C is false must be constructed. This can be accomplished by listing the variables of the consequent and using the browsers to find the transitions that reference these variables in their entry or exit assertions. Then, backward steps are taken from tr_A until any of these transitions is reached. If for every backward sequence, the first transition reached is one that asserts C is true, then tr_A preserves the property. Otherwise, the property is violated.

Similar steps are performed for each transition tr_~C that can change C to false that did not meet the entry/exit criteria. In this case, however, the list of transitions that assert A is false or A is true is constructed. Then backward steps are taken from tr_~C. If for every backward sequence, the first transition reached is one that asserts A is false, then tr_~C preserves the property. Otherwise, the property is violated. If all transitions tr_A and tr_~C preserve the property, then the property holds at all times.

In general, it is not necessary to find every exact sequence back to the appropriate transition. For example, in the case of tr_A, it is sometimes possible to assume that a transition that asserts C is true fires at some time in the past, which causes a condition that disallows a transition that asserts C is false to be enabled up until tr_A fires. This is shown in the Enter_Digit case of the following proof.

As an example of a proof by transition sequence analysis, consider the property "Busytone → ~Ringback" of the Phone process of the phone system. The only transition that asserts Busytone is Start_Busytone. The entry and exit assertions of Start_Busytone do not constrain Ringback, so it is necessary to show that whenever Start_Busytone fires, ~Ringback holds. Start_Ring, Start_Busytone, and Hangup cannot fire immediately before Start_Busytone from their entry and exit assertions. If Pickup or Stop_Ringback fire immediately before Start_Busytone, ~Ringback holds directly from their exit assertions. If Start_Tone, Stop_Ring, or Stop_Busytone fire immediately before Start_Tone, then ~Busytone holds from the invariant and/or inductively from the schedule.

Suppose Enter_Digit fires immediately before Start_Busytone. If Phone_State(P) is Ready_To_Dial at Start(Enter_Digit), then Dialtone holds and ~Ringback holds inductively from the schedule. If Phone_State(P) is Dialing, then the last change of Phone_State(P) was to Ready_To_Dial and Enter_Digit fired since this change from the imported variable clause. Again, Dialtone must have held, so ~Ringback held at the end of this Enter_Digit. The only way for Ringback to have changed from this end is for Start_Ringback to have fired. Start_Ringback requires Phone_State(P) to be Waiting; thus, since Phone_State(P) was only Ready_To_Dial or Dialing up until

Start(Start_Busytone), Start_Ringback could not have fired. Thus, ~Ringback holds if Enter_Digit fires immediately before Start_Busytone. Finally, suppose Start_Ringback fires immediately before Start_Busytone. Again, Start_Ringback requires Phone_State(P) to be Waiting. Thus, there was a change of Phone_State(P) to Busy after Start(Start_Ringback). From the imported variable clause, however, Phone_State must have changed to Busy from Dialing and there was an end to Enter_Digit since this time. Thus, Start_Ringback could not have fired immediately before Start_Busytone, which is a contradiction. The proof of the transitions that assert Ringback, namely Start_Ringback, is almost identical to the proof of Start_Busytone.

*9.2.4.1.3. Timed Operator Analysis*

From the definition of an untimed property in section 8.3.1, an untimed property may contain timed operators as long as they are evaluated at the current time. Additionally, an untimed property may reference imported variables. The complexity of the property depends on where these additional items appear in the property. There are three forms that appear most often in specifications. The most common of these is when timed operators and/or imported variables appear in the property and only in the antecedent. For example, the following property of the Proc process of the bakery algorithm states that whenever number changes to a value other than 0, the new value must be greater than or equal to the numbers of all processes exec_time in the past.

```
        Change(number, now)
    &   number ≠ 0
→       FORALL i: procs_int
            (number ≥ past(procs[i].number + 1, now - exec_time))
```

In these properties, the additional timed operators and imported variable references serve to further limit the transition cases that need to be checked in the two analysis techniques above. The change operator in the above property means it is necessary to only check this property for transitions that change number. In the Proc process, the only transition that changes number to a value other than zero in its exit assertion is set_number. By the exit assertion of set_number, number is chosen to be a value greater than or equal to the number of all processes exec_time in the past, thus the property holds.

The second most common form is when only starts and ends of local transitions and/or changes to local variables appear in the property in both the antecedent and the consequent. For example, the following could be an additional property of the Proc process. This property states that set_number is the only successor of set_choose and fires immediately at the end of set_choose.

```
        End(set_choose, now)
→       Start(set_number, now)
```

The proofs of these properties consist of showing that the only possible predecessors or successors of the transitions referenced directly (in start/end expressions) or indirectly (in change expressions) in the antecedent are the transitions referenced directly or indirectly in the consequent.

The third most common form is when calls to local transitions, calls, starts, or ends of imported transitions, and/or changes to imported variables occur in the antecedent and a start of a local transition occurs in the consequent. For example, consider the following property of the Judge process of the Olympic boxing scoring system.

```
        Call(Score, now)
→       Start(Score, now)
```

In the proofs of these properties, it is necessary to show that the process is idle whenever the external event occurs. This can usually only be shown when some assumptions are made about the frequency at which the antecedent events can occur. In the Judge process, the following environment clause states that calls to Score must be separated by at least 2 * Window.

```
        EXISTS t: time
            (   t ≤ now
            &   Call2(Score, t))
→       Call(Score) - Call2(Score) ≥ 2 * Window
```

The analysis techniques for forward liveness properties are then used to show that if two external events occur as closely together as possible, then the appropriate transition will start at the time of the second external event.

*9.2.4.2. Timed Properties*

Before the distinguishing features of liveness and safety proofs can be discussed, it is necessary to define two terms. An *embedded liveness requirement* is a condition of the requirement of a safety property that is not trivially known to hold at the time the context becomes true. For example, consider the following forward safety property of the Gate process of the railroad crossing system, which has been rewritten into an equivalent form to clarify the presentation.

```
FORALL t: time, s: sensor_id
    (   s.train_in_R
    &   Change(s.train_in_R) - dist_R_to_I / max_speed + response_time ≤ t
    &   t ≤ now
→       past(position, t) = lowered)
```

In this property, the context becomes true at Change(s.train_in_R) - dist_R_to_I / max_speed + response_time. At that time, it is not known whether position = lowered. Thus, this property has an embedded liveness requirement. In other words, before it can be proved that position = lowered at all times in the interval [Change(s.train_in_R) - dist_R_to_I / max_speed + response_time, now], it

must first be shown that position has *become* lowered by the time the interval begins. In contrast, consider the following backward safety property of the Sensor process of the railroad crossing.

```
        Change(train_in_R, now)
    &   ~train_in_R
→       FORALL t: time
            (   now - ((dist_R_to_I + dist_I_to_out) / max_speed - response_time) ≤ t
            &   t < now
            →       past(train_in_R, t))
```

In this property, the context becomes true at current time. At that time, it is known that train_in_R has just changed to false, thus it is known that it was true at the instant before. In this property, it is not necessary to derive that train_in_R was true when the proof interval started. It is only necessary to prove that train_in_R does not change value in the interval. Let a *proper safety property* be a safety property that does not have an embedded liveness requirement. The Sensor property shown above is a proper safety property. For the remainder of this chapter, let "safety properties" refer to only proper safety properties and "liveness properties" refer to either liveness properties or safety properties with embedded liveness requirements.

*9.2.4.2.1. Liveness Properties*

Unlike untimed properties in which portions of the execution history can be abstracted away, in liveness properties, it is necessary to derive all of the exact transition sequences that can occur in the process. Otherwise, it is not possible to compute the running time to prove that it is less than the required response time. The basic technique to prove a liveness property is to determine the possible states of the process and the operating environment when the context holds, to step forward or backward from each possible state until the requirement holds, and then to check that the time of each sequence is less than the required response time. The state of a process includes which transition is currently firing, if any, in the process and what the value of each variable is. The state of the operating environment includes what the values of imported variables are, what is occurring on other processes, and which transitions have been called but not yet serviced. The process state is the more critical of the two.

The transitions that can be firing in the process when the context holds depends on the types of conditions in the context. If the context contains restrictions on the local state (i.e. variable values and transition start/end times), then only a few transitions may be possible. If the context only contains restrictions on the external or imported state (i.e. calls from the environment and imported variable values), all transition may be possible. In these cases, the transitions possible may be limited by environmental and/or imported variable assumptions referencing the context conditions. This can

be determined by performing the "formulas..using items used in antecedent of Selected Formula" query in the formula splitter. This query displays the assumptions about items referenced in the context of the property. These assumptions must be examined by the user to determine whether there are any limitations on the transitions that can be firing.

The values that variables have when the context holds also depend on the types of conditions in the context. The variables of most interest are the *control variables*, which affect the basic flow of execution (such as the "position" variable of the Gate process), as opposed to *data variables*, which store computed results (such as the "number" variable of the Central_Control process of the phone system). Control variables are referenced in transition entry assertions, which means they affect when different transitions are enabled, while data variables are only referenced in exit assertions, thus have no affect on the execution of the process. The control variables usually have enumerated and boolean types, while the data variables usually have integer and real types. The control variables of a process can be displayed using the "variables..controlling behavior of Selected Process" query of the process browser. A similar query can be used to display the data variables. Since most control variables are simply typed, all possible values of these variables can be explicitly enumerated. The possible values of variables with infinite domains that affect process behavior, however, must be broken down into equivalence classes (e.g. $x < 0$, $0 \leq x \leq 10$, $10 < x$, for a variable x), where equivalence in this case is affecting the transition that will fire next. This is possible because specifications are finite in length, thus there can only be a finite number of such equivalence classes. If it is necessary to base cases on such variables, the specification must be examined to determine the appropriate equivalence classes. This can be assisted by using the browsers to find all the transitions that reference the variable in their entry assertions and then inspecting each entry assertion by hand. In most cases, the possible variable values will be limited by the entry assertion of the transition that is firing in the process. If the process is idle, however, then all possible cases must be considered.

In many instances, the state of the operating environment will not have to be split explicitly into cases as with the local state, but will be determined implicitly by what can be happening in the local process. After the transition and variable splits have been performed, browser queries can be used to find the assumptions that are relevant to each, which can be used to limit the possible operating environments. As is the case with variables that have infinite domains discussed above, the values of imported variables and call, start, and end times of imported transitions may need to be split into equivalence classes if their values cannot be implicitly derived. Figure 9.2.4.2.1 shows a portion of the case splits for one of the transition cases.

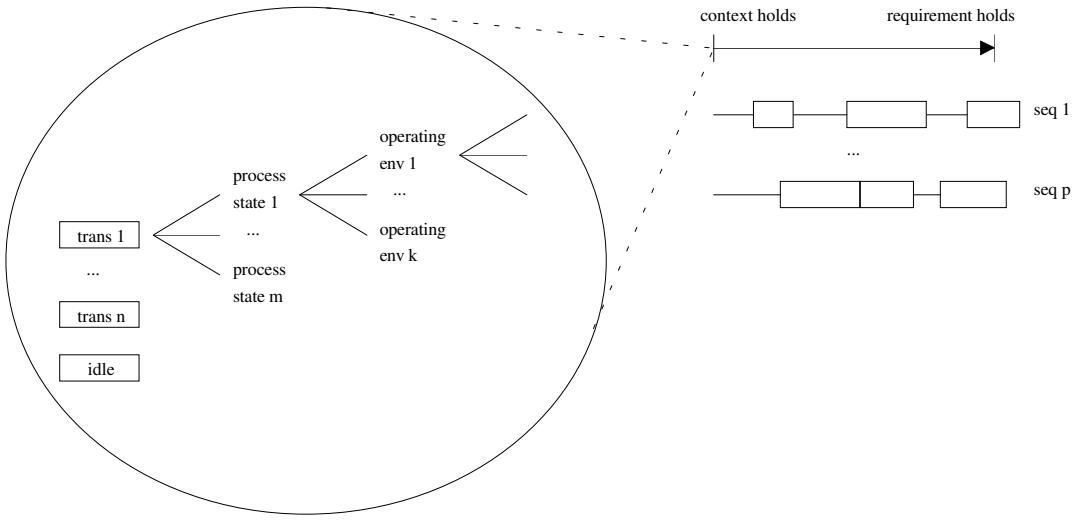



Figure 9.2.4.2.1: Case splitting for a forward liveness property

After the process state and operating environment state has been split up appropriately, it is necessary to determine what sequences of transitions must occur in each case for the requirement to hold. This is done by performing forward or backward steps (depending on the direction of the property) from the context until the requirement holds. The key to minimizing the number of cases that must be fully expanded and to keep the amount of work to a reasonable level is to carefully choose the order in which cases are performed. The goal is to choose the cases that have the longest possible running time first. This is important for two reasons. First, the sequences of longest length are the ones that are most likely to violate the response requirement. If shorter sequences are expanded first, work may be wasted when a longer case results in a violation. Second, the longest sequences are likely to subsume most of the other cases, which is critical to keeping the amount of work reasonable. A general guideline is that the maximum time will occur when some transition has just started firing at the time the context holds rather than when the process is idle. This guideline usually holds because the transition execution delays the time when the process can begin to respond to the conditions of the context. The sequence generator can be used to estimate the cases of longest length. The first step is to find the transition(s) that must occur in order for the requirement to hold. This can be done using a combination of the browsers and visual inspection. For transition starts and ends, the appropriate transitions are known directly. For specific variable values, the appropriate transitions are those that set the variable in their exit assertions. Once the appropriate transitions are found, the sequence generator is used to generate the sequences between the transition that is firing on the

process and the transitions that satisfy the requirement. The transition case that results in the longest sequence with the most delays in the running time estimate should be attempted first.

After each case is selected, the appropriate forward or backward steps must be performed. The running time of the complete sequence is obtained by summing the maximum delay associated with each step. The notion of "maximum delay" is a central theme in the proofs of liveness properties. In a liveness property, an event is required to occur within a specific amount of time. These properties are most likely to be violated when the smallest possible number of transitions fire in the proof interval since the fewer the transitions that fire, the less likely it is for the required response to occur. The number of transitions that can occur in an interval is minimized when delay between each transition is maximized. After the running time of the sequence is computed, it is necessary to check it against the required time. Many times, the required response time will be in terms of a single constant, while the time computed will be in terms of several transition durations and delays so there must be axioms and/or constant refinement clauses that relate the two expressions. These can be found by performing the "formulas..using items used in consequent of Selected Formula" query from the formula splitter with the liveness property in the splitter window. If the property is violated, the specification must be fixed. If not, the user must expand each of the other cases that is not subsumed by an earlier case until all cases have been proved.

As an example of how these guidelines can be used to formulate a proof sketch, consider the Gate forward liveness property of the previous section. In order for the context to hold, there must be a change of some sensor's train_in_R variable to true. Since this is a condition on the operating context of the process and no assumptions exist about when a change to train_in_R can occur, any transition may be firing or the process may be idle when the context holds. The control variable in the Gate process is the position variable. Each transition sets position to a unique value, so after the transitions complete execution, the local state is fully known. If the gate is idle when the change occurs, position may be any one of its four possible values. The only imported item referenced in the gate is train_in_R from each sensor, and from the antecedent of the requirement, one sensor has the value true throughout the period of interest. In order for the requirement to hold, the position must be lowered. The transition that achieves this is the down transition. After using the sequence generator to estimate the cases with the longest running time, it is found that the cases when up is firing or raise is firing seem to have the longest running time.

In both the up and raise cases, once the process becomes idle, the only transition that can fire next is lower. Lower will fire immediately since a train is in the region by the context, and the position is

raised or raising by the exit assertion of up or raise, respectively. Since a train is in the region throughout the proof interval, the only transition that can fire after lower is down. Down fires at lower_time after lower ends, which satisfies the requirement. The time of this sequence is {up_dur, raise_dur} + lower_dur + lower_time + down_dur. Thus, in order for the requirement to hold {up_dur, raise_dur} + lower_dur + lower_time + down_dur ≤ dist_R_to_I / max_speed - response_time, which is true by the local axiom of the Gate process. No transition can fire after down in the proof interval because raise is the only possibility, but raise cannot fire since a train is in the region. All of the other cases result in a sequence that is a subsequence of the up and raise cases, thus the property holds.

*9.2.4.2.2. Safety Properties*

In the proof of a liveness property, the user must show all of the exact sequences of events that occur from the time the context holds until the time the requirement holds. When proving a safety property, however, it can be assumed that a violation occurs at some time in the proof interval and then forward or backward steps are taken to achieve a contradiction at the context times of the property. In contrast to liveness properties, in which "maximum delay" is the central theme, in safety properties, the central theme is "minimum delay". In a safety property, certain events are required to *not* occur within a specific period of time. These properties are most likely to be violated when the largest possible number of transitions fire in the proof interval since the more transitions that fire, the more likely it is for the undesired response to occur. The number of transitions that can occur in an interval is maximized when delay between each transition is minimized. Thus, if a violation exists, it can usually be discovered by stepping towards the context using the minimum delay associated with each step. It is not always necessary to step completely to the context if a contradiction can be achieved immediately. The direction of the steps that are taken is always the opposite of the direction of the property. That is, in a forward property, backward steps are taken to the context times as shown in figure 9.2.4.2.2-1.

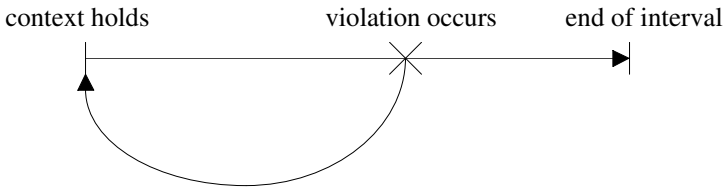



Figure 9.2.4.2.2-1: Proving a forward safety property

In a backward property, forward steps are taken to the context times as shown in figure 9.2.4.2.2-2.

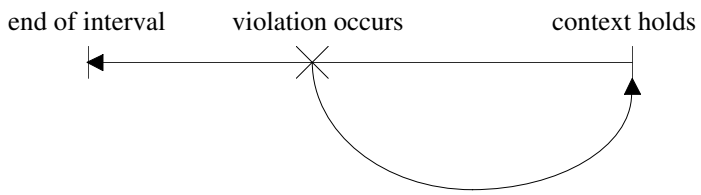



Figure 9.2.4.2.2-2: Proving a backward safety property

For example, consider the Sensor backward safety property shown in section 9.2.4.2. Since train_in_R just changed to false at now, exit_I fired at now - exit_dur. For this property, it can be assumed that train_in_R is false at some time in the interval [now - ((dist_R_to_I + dist_I_to_out) / max_speed - response_time), now). Since train_in_R is false in this interval and is true at now - exit_dur by the entry assertion of exit_I, there was a change of train_in_R to true at some time in the interval (now - ((dist_R_to_I + dist_I_to_out) / max_speed - response_time), now). Enter_R is the only transition that can achieve such a change, thus enter_R ended at some time within this interval. By the entry assertion of exit_I, exit_I can only fire when (dist_R_to_I + dist_I_to_out) / min_speed - exit_dur time has elapsed since the last time enter_R started. The earliest that enter_R could have started is just after now - ((dist_R_to_I + dist_I_to_out) / max_speed - response_time) - enter_dur. Thus, it is necessary to show that (now - exit_dur) - (now - ((dist_R_to_I + dist_I_to_out) / max_speed - response_time) - enter_dur) ≥ (dist_R_to_I + dist_I_to_out) / min_speed - exit_dur, or equivalently that (dist_R_to_I + dist_I_to_out) / max_speed - response_time + enter_dur - exit_dur ≥ (dist_R_to_I + dist_I_to_out) / min_speed - exit_dur. All of the constants in this formula are positive numbers and by the global axiom clause, max_speed ≥ min_speed, so (dist_R_to_I + dist_I_to_out) / max_speed ≤ (dist_R_to_I + dist_I_to_out) / min_speed. By the local axiom clause, response_time ≥ enter_dur, thus the inequality is false, so exit_I could not have fired at now - exit_dur, which is a contradiction. Thus, the property holds.

### 9.2.5. Iterative Single-Threaded Processes

Iterative single-threaded processes operate similarly to simple single-threaded processes except that they record the number of iterations they perform, which allows properties between iterations to be expressed. Thus, in general, the techniques presented for simple single-threaded processes can also be applied to iterative single-threaded processes. For liveness properties that span multiple values of the iteration count, however, different techniques must be used. The techniques for liveness properties of simple single-threaded processes are based on generating the possible transition sequences between two events in the process. Since these events mainly span a single iteration of the

process, the number of sequences possible and the number of transitions in each sequence are usually small. In an iterative single-threaded process, however, a property can span an arbitrary number of iterations in the process. For example, consider the following liveness property of the Elevator process of the elevator control system.

```
FORALL f: floor
    (    the_elevator_buttons.Call(request_floor(f), now - t_service_request)
→       EXISTS t: time
            (    now - t_service_request < t
            &    t ≤ now
            &    past(position, t) = f
            &    past(Change(door_open, t), t)
            &    past(door_open, t))))
```

When the call to request_floor occurs, the elevator may be at any position in the building and moving in either direction. In order to satisfy the liveness property, the elevator might have to move all the way up to the top of the building and all the way back down. Thus, the number of iterations performed by the elevator is based on the maximum value of n_floors. Since n_floors is an unbounded symbolic constant, however, the concrete number of iterations that the elevator might have to perform to satisfy the liveness property cannot be determined. Therefore, it is not always possible to generate the exact transition sequences between two events of the process. Thus, the techniques presented in section 9.2.4.2 must be modified for iterative single-threaded processes.

The first thing to notice in iterative single-threaded processes is that in order for a liveness property to be guaranteed, the maximum time that can be spent in any iteration must be bounded. The other thing is that the number of full iterations between when the context holds and when the requirement is to hold must also be bounded. The main technique in constructing the proof sketch for these properties is to determine these bounds and then to derive the maximum response time accordingly. For forward liveness properties, the maximum response time is derived as shown in figure 9.2.5-1, while for backward liveness properties, it is derived as shown in figure 9.2.5-2.

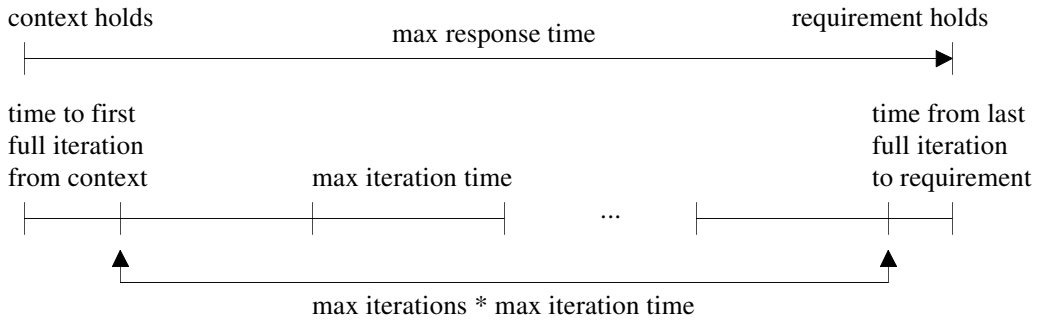



Figure 9.2.5-1: Deriving the maximum forward response time

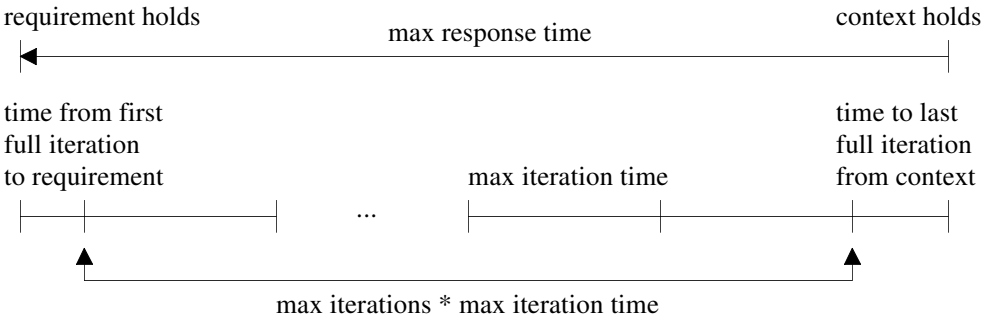



Figure 9.2.5-2: Deriving the maximum backward response time

*9.2.5.1. Determining the Maximum Time to a Full Iteration from the Context*

To determine the maximum amount of time to a full iteration from the context, it must first be determined what constitutes an iteration, where an iteration is a sequence of transitions between two changes to the iteration count. An iteration starts immediately at the first change and continues until just before the second change. When a process is classified as an iterative single-threaded process by the process classifier, the variable that holds the iteration count and the transition(s) that change the variable are also displayed. For example, in the Elevator process, the iteration count is stored in the "position" variable, which is changed by the "arrive" transition. Once this information has been determined, the techniques from section 9.2.4.2 can be used to find the maximum time of the property "context → End(iterate transition)". Namely, it is determined what can be occurring in the process when the context holds and then stepped forward or backward to the first or last full iteration, respectively. For example, in the Elevator process, this property is shown below. In this property, max_time is the time to be derived.

```
FORALL f: floor
    (    the_elevator_buttons.Call(request_floor(f), now – max_time)
→       EXISTS t: time
            (    now – max_time ≤ t
            &    t ≤ now
            &    End(arrive, t)))
```

One instance of the maximum time occurs when the elevator is moving up from floor one to two and two has not been requested on the elevator panel nor has any request been made on two's button panel. Let t_arrive be the next time such that End(arrive, t_arrive). Up_request and down_request are simultaneously called on the elevator button panel an "instant" after t_arrive - 2 * request_dur and down_request fires first. Thus, the up request is not posted in time for the elevator to service it and the elevator must continue on to floor three. At t_arrive, no request has been made to stop at floor two, and since floor three has been requested, the elevator moves up. This is not yet a full

iteration since the elevator has not opened its door. It takes t_move time for the elevator to reach floor three at which time arrive fires. Since a full iteration starts when arrive ends, the time to the first full iteration is 2 * request_dur + clear_dur + move_up + t_move + arrive_dur. The maximum time to the first full iteration cannot take longer than this because the only way to increase this time is to increase the time before the elevator initially gets to floor three. If request_floor is called any earlier, however, than the request will be posted by the time the elevator gets to floor two, so it will be serviced immediately.

*9.2.5.2. Determining the Maximum Iteration Time*

The next step is to determine the maximum amount of time that can be spent between two ends of the transition that changes the iteration count. This time is also found using the techniques described in section 9.2.4.2. In many cases, the maximum time spent in an iteration can only be bounded when certain assumptions are made. For example, in the Elevator process, the time spent on a particular floor can be unbounded if no requests are outstanding in the building. For the liveness property stated above, however, a request will be outstanding after the call to request_floor has been "processed".

As an example of determining the maximum amount of time that can be spent in an iteration, consider the Elevator process. For any floor f, open_dur + t_move_door + door_stop_dur + t_stop + close_dur + t_move_door + door_stop_dur + request_dur + move_dur + t_move + arrive_dur is the longest time spent with position = f when a request is outstanding on another floor besides f in the building. The position of the elevator changes in the arrive transition, so the elevator is "officially" at a floor at End(arrive). Suppose the elevator has just arrived at f, thus position = f, moving, ~door_moving, and ~door_open hold. If a request has not been made concerning f, then move_up or move_down can fire immediately upon arrival so the total time will be less than that of the case in which a request has been made. If a request is outstanding for f, then open_door is the only transition that can fire next because ~door_moving holds so door_stop cannot fire, ~door_open holds so close_door cannot fire, and Change(moving) > Change(door_open) because the elevator has just arrived so move_up and move_down cannot fire. Thus, Start(open_door, End(arrive)), ~moving, and door_moving hold at End(open_door). The only transition that can fire when door_moving holds is door_stop. Thus, Start(door_stop, End(open_door) + t_move_door) and ~door_moving hold at End(door_stop). By this time, the request concerning floor f must have been cleared. Clear_{floor, up, down}_request was enabled when the door started opening, no other clear_{floor, up, down}_request can be enabled because of the position requirement, and n_floors request_floor's can

be enabled. From the global axiom clause, clear_dur + n_floors * request_dur < t_move_door and from the imported variable clause, no other requests can be cleared, so clear_{floor, up, down}_request will finish firing before the door is fully opened.

The only transition that can fire when door_open and ~door_moving hold is close_door. Thus, Start(close_door, End(door_stop) + t_stop) and door_moving hold at End(close_door). The only transition that can fire when door_moving holds is door_stop. Thus, Start(door_stop, End(close_door) + t_move_door) and ~door_moving hold at End(door_stop). The transitions that can possibly fire at this point are move_up, move_down, and open_door. From the imported variable clause, another request concerning floor f cannot be made until after the door is fully closed. Thus, such a request cannot be posted until now > End(door_stop) + request_dur. Since another request exists in the building by the hypothesis, however, move_up or move_down (depending on where the "next" request is) will be enabled and fire at End(door_stop) + request_dur, so open_door cannot be the next transition to fire. By reasoning similar to that above, Start(arrive, End(move_{up, down}) + t_move) holds so at End(arrive), the elevator will be at a new floor. The total time spent on floor f is:

End(arrive) - $End_2$(arrive)
= (End(move_{up, down}) + t_move + arrive_dur) - $End_2$(arrive)
= (End(door_stop) + request_dur + move_dur) + t_move + arrive_dur - $End_2$(arrive)
= (End(close_door) + t_move_door + door_stop_dur) + request_dur + move_dur + t_move + arrive_dur - $End_2$(arrive)
= (End(door_stop) + t_stop + close_dur) + t_move_door + door_stop_dur + request_dur + move_dur + t_move + arrive_dur - $End_2$(arrive)
= (End(open_door) + t_move_door + door_stop_dur) + t_stop + close_dur + t_move_door + door_stop_dur + request_dur + move_dur + t_move + arrive_dur - $End_2$(arrive)
= ($End_2$(arrive) + open_dur) + t_move_door + door_stop_dur + t_stop + close_dur + t_move_door + door_stop_dur + request_dur + move_dur + t_move + arrive_dur - $End_2$(arrive)
= open_dur + t_move_door + door_stop_dur + t_stop + close_dur + t_move_door + door_stop_dur + request_dur + move_dur + t_move + arrive_dur

*9.2.5.3. Determining the Maximum Number of Full Iterations*

Once the maximum iteration time has been determined, the next step is to determine the maximum number of full iterations that can occur between when the first full iteration is reached and when the requirement holds. This is equivalent to finding one less than the maximum number of times the iteration count can change its value between when the context holds and when the requirement holds. In general, this will be the number of iterations in a "full cycle" of the process. For example, one full cycle of the Elevator process occurs when the elevator starts at the bottom and moves all the way up to the top and all the way back down to the bottom or an equivalent scenario, which consists of

n_floors - 2 changes to position. In the Proc process of the bakery algorithm, a full cycle consists of n_procs iterations of the for_loop transition.

In general, it is necessary for there to be some restriction on the number of times the iteration count can "switch directions" (i.e. change from increasing to decreasing or vice-versa) before a particular value is reached. In the Elevator process, this is guaranteed by the entry assertions of move_up and move_down, which state that the elevator can only switch direction if no requests have been made in the current direction of movement. The constraint of the Elevator process states this fact. With this restriction, it is possible to determine the maximum number of full iterations after the elevator reaches floor three. From the conditions of the worst case of section 9.2.5.1, the elevator must continue up to the top since a request has been made on every floor, then must travel back down to the bottom, then must travel back up to floor two. The maximum time possible to spend on any floor when a request is outstanding elsewhere in the building is spent once on floors one, two, and n_floors, and twice on every other floor, thus 2 * n_floors - 3 is the maximum number of full iterations.

*9.2.5.4. Determining the Maximum Time from a Full Iteration to the Requirement*

After the maximum number of full iterations has been determined, it is necessary to find how long it takes from when the conditions of the context hold until the first full iteration is reached. To find this, the techniques from section 9.2.4.2 can be used to find the maximum time of the property "End(iterate transition) → requirement". For example, in the Elevator process, this property is shown below. In this property, max_time is the time to be derived.

```
FORALL f: floor
    (   End(arrive, now – max_time)
→       EXISTS t: time
            (   now - max_time < t
            &   t ≤ now
            &   past(position, t) = f
            &   past(Change(door_open, t), t)
            &   past(door_open, t))))
```

By previous reasoning, it takes open_dur + t_move_door + door_stop_dur for the elevator door to be fully opened.

*9.2.5.5. Deriving the Maximum Response Time*

After all of the information from the previous sections has been determined, the maximum response time can be constructed using either figure 9.2.5-1 for a forward property or figure 9.2.5-2 for a

backward property. For the liveness property of the Elevator process, the maximum response time is shown below.

```
    maximum time to first full iteration from context +
    maximum number of iterations * maximum iteration time +
    maximum time from last full iteration to requirement
=   (2 * request_dur + move_dur + t_move + arrive_dur) +
    (2 * n_floors - 3) * (open_dur + t_move_door + door_stop_dur + t_stop + close_dur +
                          t_move_door + door_stop_dur + request_dur + move_dur + t_move +
                          arrive_dur) +
    (open_dur + t_move_door + door_stop_dur)
```

After the maximum response time has been derived, it is necessary to check that the derived time is less than the required response time. This either follows directly from the property or indirectly from local and global axioms or constant refinement clauses that restrict the constants in the time expression. For example, in the elevator schedule fragment, the response is required within t_response_time. The maximum time derived, however, is not in terms of t_response_time. Thus, the appropriate clauses must be examined to find a relationship between the two expressions. An appropriate axiom is found in the local axiom clause of the Elevator process as shown below.

```
(t_service_request ≥
        2 * request_dur + move_dur + t_move + arrive_dur +
        (2 * n_floors - 3) * (open_dur + t_move_door + door_stop_dur + t_stop +
                              close_dur + t_move_door + door_stop_dur +
                              request_dur + move_dur + t_move + arrive_dur) +
        open_dur + t_move_door + door_stop_dur)
```

This completes the proof of the elevator liveness property.

### 9.2.6. Multi-Threaded Processes

The main technique used to construct the proof sketch of a single-threaded process is to analyze the possible transition sequences of the process and then derive the required properties. While it is possible for a single-threaded process to have a large amount of nondeterminism, in most realistic systems, the amount of nondeterminism is limited. This means that there are only a small number of possible successors or predecessors to each transition, thus the number of transition sequences that need to be analyzed can be kept reasonable. Multi-threaded processes, however, are inherently nondeterministic since each individual thread is essentially independent of every other thread. This means that it is not practical to reason about transition sequences, because for a process with n threads and m transitions there are n * m possible transitions that can execute at every "step". For this reason, different techniques must be used for multi-threaded processes.

*9.2.6.1. Untimed Properties*

In general, the techniques presented for untimed properties in section 9.2.4.1 can also be used for multi-threaded processes. The transition sequence analysis technique, however, must be modified somewhat for untimed properties that only reference the variables associated with a single thread. Let the *property thread* be the thread that is referenced in such a property. For these untimed properties, it is necessary to abstract away much of the execution history of the process instead of finding the exact transition sequences. Namely, it is necessary to ignore the portions of the execution history that deal with transitions that do not set the variables associated with the property thread. Thus, when untimed backward steps are made, they are made only for the transitions of the property thread. The parameterized extension of the sequence generator discussed in section 8.5.4 does this automatically when the sequences of a multi-threaded process are displayed.

As an example of this type of property, consider a possible property of the Central_Control process of the phone system as shown below.

```
FORALL P: Area_Phone
    (   Phone_State(P) = Busy
→       ~Enabled_Ringback_Pulse(P))
```

This property states that whenever the state of a particular phone is busy, the ringback pulse associated with that phone is not enabled. To prove this property, transition sequence analysis can be used on the thread associated with an arbitrary phone P. For example, the first exception of the Process_Local_Call transition can set Phone_State(P) to Busy. Thus, every backward sequence must be checked to make sure that a transition that asserts ~Enabled_Ringback_Pulse(P) is true occurs before a transition that asserts ~Enabled_Ringback_Pulse(P) is false. Any transition that fires for a phone other than P can be ignored while stepping backward.

*9.2.6.2. Timed Liveness Properties*

The focus of this section is on liveness properties that reference the events of only one of the multiple threads since those are usually the liveness properties of most interest in multi-threaded processes. The techniques below, however, can be generalized to arbitrary liveness properties. In general, it is impossible to guarantee a liveness property of a specific thread in a multi-threaded process without making assumptions about the behavior of the set of threads. Namely, it is necessary to guarantee that the thread will not be "starved" by the other threads. This is usually achieved by choosing an appropriate *scheduling policy* and by placing various limitations on the number of transitions enabled or threads that require service at any given time. In multi-threaded operating systems, the scheduling

policy specifies how to select the next thread to execute on the processor when multiple threads are waiting for service. In ASTRAL, the scheduling policy specifies how to select the next transition to execute on a process when multiple transitions are enabled. A number of scheduling policies have been developed for multi-threaded operating systems [Tan 92] that are relevant to ASTRAL. The most common of these policies, which will be the ones considered, include fixed priority scheduling, first in-first out (FIFO) scheduling, and round robin scheduling. In ASTRAL, the fixed priority and FIFO policies are applicable to any system, whereas the round robin policy is only applicable to multi-threaded systems.

*9.2.6.2.1. Determining the Scheduling Policy*

To construct the proof sketch of a liveness property of a multi-threaded process, the first step is to determine the scheduling policy of the process. In *fixed priority scheduling*, each transition is assigned a static priority such that whenever more than one transition is enabled, the transitions with the highest priority are always executed before those with lower priorities. The priority assignment can be modified to allow transitions to have different priorities in different circumstances. In *FIFO scheduling*, transitions that have been enabled for a longer period of time have priority over transitions that have been enabled more recently. Finally, in *round robin scheduling*, transitions enabled in a given thread thd1 have priority over transitions enabled in threads that have executed a transition after the transition in thd1 was enabled. Currently, ASTRAL only has built-in mechanisms to support fixed priority scheduling, which can be specified in transition selection clauses. Other scheduling policies can be specified only by explicitly adding the scheduling constraints into transition entry assertions. This is undesirable as it limits the implementation choices for a particular process. A simple extension to the transition selection mechanism presented in section 5.2.1, however, would allow other policies to be specified without limiting implementation choices.

Besides some slight syntactic and typing issues, the extension consists of:

(1) defining enabled_transitions and eligible_transitions in terms of transition-parameter pairs
(2) allowing setdef expressions to be used to construct transition-parameter sets
(3) allowing quantification over transitions and parameters

(1) allows a transition of a specific thread to be given priority over transitions of other threads rather than giving priority to the same transition in all threads. (2) allows transition sets to be constructed using complex expressions such as when transitions became enabled and when transitions started. (3) allows greater flexibility in defining setdef conditions. Namely, it allows a transition to be given priority over a large number of other transitions without explicitly listing each one. Quantification

over parameters is assumed to include an "empty parameter" case for transitions without parameters.

Using this extension, a FIFO scheduling policy can be specified as shown in the following transition selection clause.

```
        TRUE
→       eligible_transitions =
            {setdef tr1(p1): transition(parameter)
                (FORALL tr2: transition, p2: parameter
                    (   enabled_transitions CONTAINS {tr2(p2)}
                    →   Change(enabled_transitions CONTAINS {tr1(p1)}) ≤
                            Change(enabled_transitions CONTAINS {tr2(p2)})))}
```

This clause specifies that if a transition tr1 in a thread thd1 became enabled before any transition tr2 of any thread thd2, then tr1 on thd1 has priority over tr2 on thd2. Note that the notation "tr1(p1)" indicates the transition-parameter pair (tr1, p1) and refers to the transition tr1 given parameter p1. Because of the assumption that threads in multi-threaded processes are always indicated by parameterized transitions as discussed in section 8.2.1, this is equivalent to saying transition tr1 on the thread associated with parameter p1.

Similarly, a round robin scheduling policy can be specified as shown below.

```
        TRUE
→       eligible_transitions =
            {setdef tr1(p1): transition(parameter)
                (FORALL tr2: transition, p2: parameter
                    (   enabled_transitions CONTAINS {tr2(p2)}
                    →   EXISTS tr3: transition
                            (Change(enabled_transitions CONTAINS {tr1(p1)}) ≤
                                Start(tr3(p2)))))}
```

This clause specifies that if two transitions tr1 and tr2 are enabled in threads thd1 and thd2, respectively, then tr1 has priority over tr2 if another transition tr3 has fired on thd2 after tr1 became enabled.

As an example of a scheduling policy, consider the following schedule property of the Central_Control process. Although this property is classified as a forward safety property by the property classifier, it contains liveness aspects. Namely, if Phone_State(P) of a phone P changes from Idle, either it changes to Ringing or it changes to Ready_To_Dial *within 2 time units*.

```
FORALL P: Area_Phone, t, t1, t2: Time
    (   t ≤ t1
    &   t1 < t2
    &   Change2(Phone_State(P), t)
    &   past(Phone_State(P, t) = Idle
    &   P.End(Pickup, t1)
    &   P.Offhook
```

```
    &  Change(Phone_State(P), t2)
→       past(Phone_State(P), t2) = Ringing
    |   past(Phone_State(P), t2) = Ready_To_Dial
    &   t2 ≤ t1 + 2)
```

This property is guaranteed using a fixed priority scheduling policy. The scheduling policy is stated in the transition selection clause of the Central_Control process as shown below.

```
        enabled_transitions CONTAINS {Give_Dial_Tone}
    &   TRUE
→       eligible_transitions = {Give_Dial_Tone}
```

This clause states that Give_Dial_Tone is given priority over all other transitions.

*9.2.6.2.2. Determining the Sequences of the Property Thread*

After the scheduling policy of the process has been determined, the next step is to determine the sequence of transitions that need to occur in the property thread for the requirement to hold. This can be accomplished using the techniques in section 9.2.4.1.2 modified as discussed in section 9.2.6.1. Note that the techniques for finding *untimed* transition sequences are used. This is because it is assumed that transition delays will be subsumed by the delays caused by the other threads in the process, thus only the sequencing information is important at first.

As an example of determining the sequences of the property thread, consider the property above for a phone P. When Phone_State(P) = Idle, the only transitions that can change Phone_State(P) are Receive_LD, Process_Local_Call, and Give_Dial_Tone. If Receive_LD or Process_Local_Call fire, Phone_State(P) = Ringing by their exit assertions. Suppose Receive_LD and Process_Local_Call do not fire before Give_Dial_Tone. Give_Dial_Tone is enabled at t1, when P becomes Offhook and must fire within 2 time units to satisfy the property. Thus, the only sequence of interest in the property thread is the sequence that consists solely of Give_Dial_Tone.

*9.2.6.2.3. Determining the Scheduling Policy Limits*

The next step is to determine the limits necessary for the particular scheduling policy of the process. For fixed priority scheduling, it is necessary to know the maximum number of transitions at each priority level that can be enabled at a given time in all threads. In particular, these numbers must be known for transitions that have equal or higher priorities than the transitions in the sequence of the property thread. These transitions can be found using the "transitions..with same priority as Selected transition" and "transitions..with higher priority than Selected Transition" queries in the transition browser. These queries examine the transition selection clauses to determine the appropriate transitions. For FIFO scheduling, it is necessary to know the maximum number of transitions that

can be enabled in all threads at the same time. For round robin scheduling, it is necessary to know the maximum number of threads that may need servicing at any given time.

For example, the Central_Control process uses a fixed priority scheduling policy. Thus, to prove the above property, it is first necessary to find the maximum number of transitions that can be enabled at a given time in all threads that have equal or higher priorities than Give_Dial_Tone. Using the transition browser, it is found that no transition has a higher priority than Give_Dial_Tone and the only transition that has the same priority is itself, thus it is necessary to determine how many threads may have Give_Dial_Tone enabled at the same time. It can be determined that no more than Max_Cust + 1 Give_Dial_Tone transitions can fire consecutively, without the Central_Control being idle or firing a transition other than Give_Dial_Tone in between. Initially, ~P.Offhook holds for all P and P.Pickup must fire to set P.Offhook, so at first, the Central_Control is not firing Give_Dial_Tone. Without loss of generality, suppose the Central_Control started firing a transition T other than Give_Dial_Tone at time t0, immediately followed by Max_Cust + 1 consecutive Give_Dial_Tone transitions, where the sequence finishes at time t1. Give_Dial_Tone(P) was not enabled at t0 for any P, or else it would contradict the transition selection clause, where Give_Dial_Tone has the highest priority. Give_Dial_Tone(P) can only become enabled by Terminate_{Local_Call, LD_1, LD_2} setting Phone_State(P) = Idle, or by P.Pickup, which sets P.Offhook. From the imported variable clause, the number of Area_Phones that can fire Pickup within two time units is bounded by Max_Cust. Thus, the number of Area_Phones that complete firing Pickup within two time units is also bounded by Max_Cust. Thus, in order for the sequence to occur, T = Terminate_{Local_Call, LD_1, LD_2} and Max_Cust Pickups finish firing in the interval (t0, t1]. When Give_Dial_Tone(P) fires, it can no longer fire until Phone_State(P) returns to Idle. At t1, all Give_Dial_Tones that were enabled between t0 and t1 have completed firing. The only way for another Give_Dial_Tone to fire at t1 would be for another Pickup to complete firing at or before t1. From the constant refinement clause, however, 2 > MAX(Tim1, ..., Tim16) + (Max_Cust + 1) * Tim1, thus t1 - t0 = MAX(Tim13, Tim15, Tim16) + (Max_Cust + 1) * Tim1 < 2, so there cannot be another Pickup at or before t1, thus the Central_Control must be idle or fire a transition other than Give_Dial_Tone at t1. Therefore, there cannot be a sequence of more than Max_Cust + 1 Give_Dial_Tones in a row.

*9.2.6.2.4. Deriving the Maximum Response Time*

After the scheduling policy has been determined and the appropriate limits have been found, an estimate of the maximum response time can be derived based on the scheduling policy of the process.

The maximum response time can be derived for each scheduling policy as shown in figures 9.2.6.2.4-1, 9.2.6.2.4-2, and 9.2.6.2.4-3. In these figures, the response time is only shown for a single transition of the property thread. If the sequence of transitions that must occur in the property thread contains more than one transition, then the response time indicated for one transition must be computed for each transition in the sequence and added together.

For fixed priority scheduling, the maximum response time can be derived as shown in figure 9.2.6.2.4-1. In this case, there may be some arbitrary transition firing when all the requests are made and thus before the priority scheduling policy takes effect. In the figure, there are n - 1 priority levels that are higher than the priority of the property thread transition.

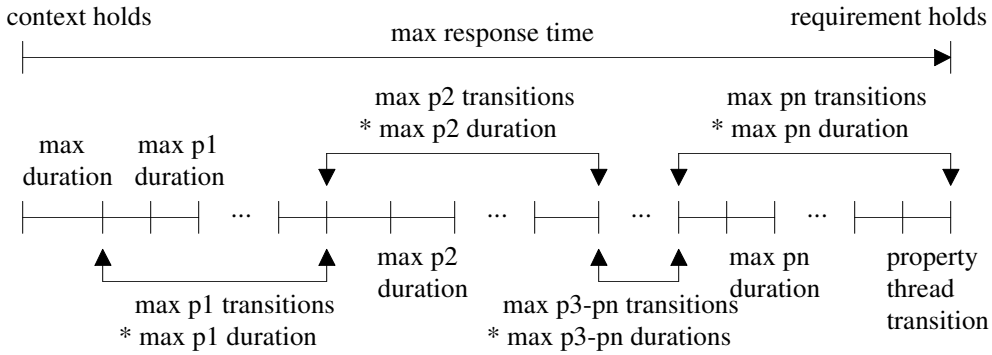



Figure 9.2.6.2.4-1: Deriving the maximum response time for fixed priority scheduling

For FIFO scheduling, the maximum response time can be derived as shown in figure 9.2.6.2.4-2. In this case, there are a maximum number of requests that can be outstanding at any time, thus the FIFO policy is always in effect.

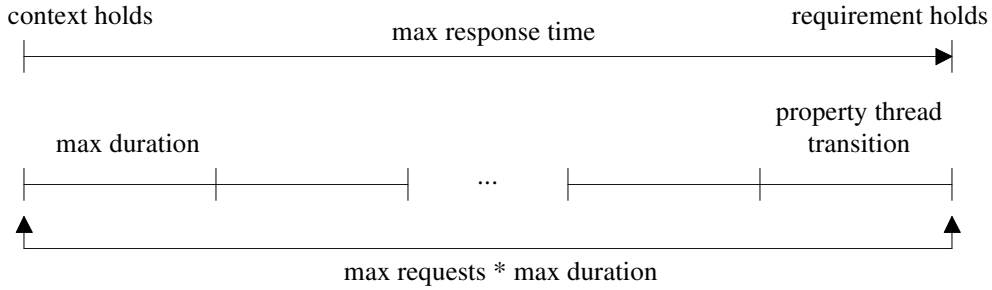



Figure 9.2.6.2.4-2: Deriving the maximum response time for FIFO scheduling

For round robin scheduling, the maximum response time can be derived as shown in figure 9.2.6.2.4-3. Similarly to the FIFO case, there is a maximum number of threads that can require service at any given time, so the round robin policy is always in effect.

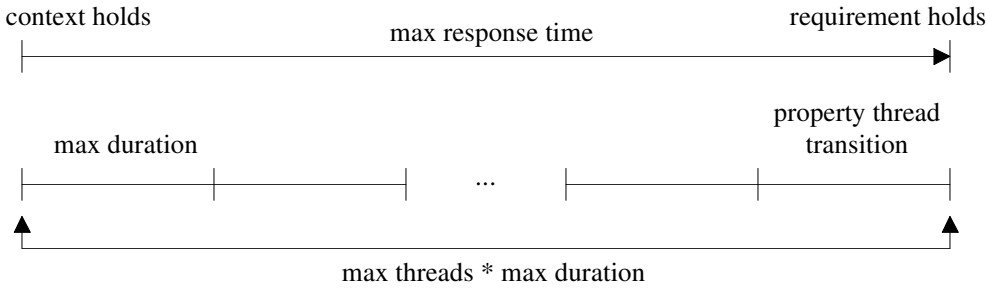



Figure 9.2.6.2.4-3: Deriving the maximum response time for round robin scheduling

To complete the proof of the schedule property of the Central_Control process, figure 9.2.6.2.4-1 is used. The maximum transition duration that can delay the sequence of Give_Dial_Tones is MAX(Tim2, ..., Tim16). Note that Tim1 (i.e. the duration of Give_Dial_Tone) is not included because if Give_Dial_Tone was the transition that was firing, then only Max_Cust other Give_Dial_Tones could fire. From the proof above, the maximum number of Give_Dial_Tone transitions that can fire in a row is Max_Cust + 1. Thus, the maximum time between P becoming Offhook and Phone_State(P) becoming Ready_To_Dial is MAX(Tim2, ..., Tim16) + (Max_Cust + 1) * Tim1, which is less than two from the constant refinement clause, thus the property holds.

# Chapter 10

# Theorem Prover Utilization

Although human proofs provide some assurance that a specification meets its critical requirements, the proofs themselves may not be valid due to flaws in reasoning caused by human error. To provide maximal assurance that the critical requirements are met, a mechanical theorem prover must be used. A mechanical theorem prover prevents flaws in reasoning by allowing proofs to proceed only in sound, well-defined steps. Besides keeping reasoning sound, theorem provers have many other benefits. They assist in the manipulation of formulas and have the ability to finish trivial subproofs automatically. Theorem provers also provide bookkeeping features such as recording the completion status of each proof. In addition, proofs can be saved, which allows them to be rerun during the maintenance phase and provides a standard proof documentation style. Finally, a theorem prover aids in the rigorous definition of a specification language by allowing its semantics to be formally defined within the language of the prover instead of using a "pencil and paper" semantics.

## 10.1. The Drawbacks of a Theorem Prover

The use of a mechanical theorem prover, however, also suffers from a number of drawbacks that can often outweigh the benefits. Thus, it is necessary to develop techniques to alleviate as many of the drawbacks as possible to make the use of a theorem prover practical. The following sections discuss each of the drawbacks and the techniques developed for each.

### 10.1.1. Explicit Proofs of Obvious Subgoals

In hand proofs, many of the details of the proof are obvious to human intuition and can be labeled "trivial" or "obvious" and not warrant further mention. In a theorem prover proof, however, these proofs must be performed explicitly and may oftentimes encompass a large number of theorem prover steps. For example, consider the proof that $a / b \geq a / c$, for positive reals a, b, c, where $b \leq c$. For a human, this proof is obvious by basic arithmetic reasoning. In a theorem prover, however, the proof requires much more effort. For instance in PVS, which provides limited automated support for division, the user must search the PVS "prelude" (i.e. built-in definitions) for the appropriate lemma,

instantiate the lemma correctly, and possibly prove any resulting type correctness conditions (TCCs). In this case, the appropriate lemma is "both_sides_div_pos_ge2: LEMMA pz / px ≥ pz / py IFF py ≥ px".

It is not possible to exempt the user from performing the proofs of obvious subgoals due to the nature of theorem proving. If each detail were not proved completely, maximal assurance that the critical requirements are met could not be guaranteed. Some of this burden can be eliminated, however, by providing automated assistance for as many proof steps as possible. This assistance takes the form of decision procedures, lemmas, and PVS strategies. The decision procedures are provided as part of the PVS prover and allow many trivial proofs to be performed automatically. Lemmas reduce work by encapsulating many different axioms that appear together often into a single form that can be used in place of the combination. The lemmas developed for ASTRAL are discussed in section 10.3. PVS strategies allow frequently occurring proof patterns to be automated into commands that can be used during the proof process. The PVS strategies developed for ASTRAL are discussed in sections 10.4, 10.5, 10.7, and 10.8.

### 10.1.2. Unrecognizable Results

Another drawback of theorem proving is that formulas can become unrecognizable during a proof due to the decision procedures of the prover. For example, in PVS the main cause of this is the use of the grind command. The grind command performs rewriting, skolemization, and automatic quantifier instantiation. When the grind command fails, the resulting subproofs usually have little in common with the original formula and it is often difficult to decipher what needs to be proved. For instance, consider the two sequents shown in figure 10.1.2. The sequent on the left is a portion of the proof Not_Sequence(set_number, set_number) in the Proc process of the bakery algorithm. The sequent on the right is one of the sequents resulting from the grind command applied to the sequent on the left. The goal of the resulting sequent is confusing. For example, it is unclear where the times t2!2 and t2!3 came from and what they refer to.

To minimize the effects of such rewriting, the PVS strategies developed for ASTRAL were written such that whenever grind or a variant of grind is used, either the proof is completed or it is stepped back to the sequent from which grind was invoked. This is performed by the PVS command "(try (try (grind) (fail) (skip)) (skip) (skip))". In this command, grind is attempted first. If grind completes the proof, the other portions of the command are ignored. If grind generates subgoals, fail is used to step the proof back to the previous sequent. The failure should not be propagated back to

the main strategy, however, or else the strategy itself would be aborted, thus the second try is used to generate a skip instead. In the example of figure 10.1.2, grind fails, thus the proof would revert back to the sequent on the left.

Another way in which formulas may become unrecognizable is that the constructs in the language of the theorem prover are often different than the constructs in the encoded specification language. For example, some languages have special symbols that are used for operators. In the encoding of the Duration Calculus into PVS [SS 94], the ∫ operator was encoded as "dur", which means that the user may need to learn twice as many operators. To minimize the differences between the ASTRAL and PVS specification languages, each ASTRAL construct was encoded in such a way as to allow ASTRAL formulas to be specified almost identically to their PVS counterparts as discussed in section 6.3.1.

```
{-1}   (FORALL (tr2: transition, t2: time):
           Duration(tr1!1) + t1!1
               < t2 + Duration(tr2)
           AND t2 + Duration(tr2) ≤ t2!1
               IMPLIES NOT Fired(tr2, t2))
    IMPLIES
       (FORALL (t2: time):
           Duration(tr1!1) + t1!1 ≤ t2
           AND t2 ≤ t2!1 IMPLIES
       Vars_No_Change(
           Duration(tr1!1) + t1!1, t2))
{-2}   Exit(tr1!1, Duration(tr1!1) + t1!1)
{-3}   Entry(tr2!1, t2!1)
{-4}   Entry(tr1!1, t1!1)
[-5]   tr1!1 = set_number
[-6]   tr2!1 = set_number
{-7}   Duration(tr1!1) + t1!1 ≤ t2!1
[-8]   Fired(tr1!1, t1!1)
[-9]   Fired(tr2!1, t2!1)
{-10}  FORALL (tr3: transition, t3: time):
           Duration(tr1!1) + t1!1
               < t3 + Duration(tr3)
           AND t3 + Duration(tr3) ≤ t2!1
               IMPLIES NOT Fired(tr3, t3)
 |-------
```

```
{-1}   t2!3 ≥ 0
{-2}   t2!2 ≥ 0
{-3}   1 ≤ V1!1
{-4}   V1!1 ≤ n_procs
{-5}   1 + i_proc__number(procs(V1!1))(t1!1)
     ≥ 1 + i_proc__number(procs(V1!1))(t1!1)
{-6}   number(exec_time + t1!1)
     = 1 + i_proc__number(procs(V1!1))(t1!1)
{-7}   next_i(exec_time + t1!1) = next_i(t1!1)
{-8}   choosing(exec_time + t1!1)
{-9}   choosing(t1!1)
{-10}  in_critical(exec_time + t1!1)
{-11}  in_critical(t1!1)
{-12}  delay(exec_time + t1!1) = delay(t1!1)
{-13}  choosing(t2!1)
{-14}  exec_time + t1!1 ≤ t2!2
{-15}  t2!2 ≤ t2!1
{-16}  exec_time + t1!1 ≤ t2!3
{-17}  t2!3 ≤ t1!1
{-18}  set_number?(tr1!1)
{-19}  set_number?(tr2!1)
{-20}  exec_time + t1!1 ≤ t2!1
[-21]  Fired(tr1!1, t1!1)
[-22]  Fired(tr2!1, t2!1)
  |-------
{1}    number(t2!2) = number(t2!1)
{2}    number(t2!3) = number(t1!1)
```

Figure 10.1.2: A sequent before and after grind

### 10.1.3. Locating the Cause of Failed Proof Attempts

A related drawback is that it is sometimes difficult to examine a failed proof attempt and locate the portion of the original specification that caused the failure. In many cases, this is due to the fact that the formula becomes unrecognizable after using the decision procedures. In the above example, it is difficult to determine from the resulting sequent exactly why grind failed. The other main cause is that a proof in a theorem prover must often be performed in a different order or in a different fashion than in the corresponding proof by hand. This makes it difficult to determine what the problem was in the original specification that caused the failure.

The similarity between the ASTRAL and PVS constructs makes it easier to find the location of an error from a failed proof attempt. This by itself, however, is not sufficient to guarantee that the errors will be found. To help guarantee this, the PVS proofs are performed such that they parallel the proof sketches constructed in the previous stage. Thus, when an error is found in a particular location of the PVS proof, it corresponds to some step in the proof sketch stage that can be found directly in the original ASTRAL specification.

### 10.1.4. Unnecessary and Repeated Steps

The most significant drawback of using a mechanical theorem prover is the large number of ways in which time and effort can be wasted by performing unnecessary or repeated steps. The following sections discuss each of these causes and the techniques that have been developed to help reduce their impact.

#### *10.1.4.1. Error-Riddled Specification*

Performing proofs within a mechanical theorem prover can take a significant amount of time and effort. Thus, there is a large overhead associated with finding errors in a specification. The more errors that are present in a specification when theorem proving begins, the more times a particular proof must be attempted before it can be completed. Thus, it is desirable to be as confident as possible that a specification is correct before a theorem prover is invoked.

In order to assure that a specification will contain as few errors as possible before attempting the proof with a theorem prover, a sequence of less costly steps is performed before theorem proving to find as many errors as possible. These steps include model checking and proof sketch construction as discussed in sections 9.1 and 9.2, respectively.

#### *10.1.4.2. Impromptu Proof Ordering*

As was shown in section 9.2.1, the order in which proofs are performed significantly affects how many times each proof must be attempted. When using a theorem prover to attempt a proof, this effect is amplified due to the added time and complexity of performing a proof within a prover, thus choosing the appropriate proof ordering becomes even more critical. To assure that an optimal proof ordering is chosen, the SDE computes a near-optimal proof ordering for the current specification and the proof manager directs the user as to which proof should be performed next as discussed in section 5.7.

#### *10.1.4.3. Impromptu Plan of Attack*

If the user attempts to perform a theorem prover proof without first constructing an overall plan of attack as to how the proof is to be carried out, there is a high probability that the user will waste a large amount of time backtracking and reproving during the proof attempt while different avenues of attack are attempted. In order to prevent backtracking, the proof sketch stage forces the user to plan out the proof attempt before the theorem prover is invoked, which allows the theorem prover proof to be performed in a similar manner to the proof sketch.

#### *10.1.4.4. Premature Splitting*

There are many opportunities throughout the course of a proof for the user to split the proof into subgoals that can be proved separately. Proof splitting must be done with care, however, or it can result in a large number of subgoals that have identical proofs. For this reason, it is important to introduce as much information as necessary into the proof before it is split to minimize the amount of repetition that occurs. For example, suppose the user needs to apply the trans_fire axiom. If the user instantiates this axiom and immediately splits it, three subgoals result. The main subgoal states that some transition fires at the given time. In the other two subgoals, it must be proved that some transition is enabled at the given time and that the process is idle. All of these subgoals require the values of the state variables between the last known state of the system and the time the transition is to fire. If the axiom is split before this information is introduced into the sequent, then it must be introduced in all three cases. Although it is not possible to prevent the user from splitting a proof prematurely, the ordering in each of the PVS strategies developed has been optimized to prevent premature splitting.

*10.1.4.5. Similar Subproofs*

Frequently when attempting a proof within a theorem prover, a number of subgoals may be generated that have identical or similar proofs. Premature splitting is just one of the ways in which this can occur. For example, it may be that a particular sequence of axioms is used to bring about the same type of result in several places. It is also common for the same information to be needed in the proofs of different requirements. For example, in ASTRAL proofs a recurring subgoal is that one transition is or is not the successor or predecessor of another.

The main technique for reducing the number of similar proofs that must be performed is the use of lemmas and PVS strategies that capture frequently occurring proof patterns. In addition, mechanisms have been developed for declaring successor and predecessor information, which can be proved once and is accessible to all proof obligations as discussed in sections 10.7.2.2, 10.8.1.3, and 10.8.2.3. Similarly, the not_sequence_ax and not_initial_ax axioms discussed in section 8.5.5 allow Not_Sequence and Not_Initial declarations to be used in all the proof obligations after they are proved once.

*10.1.4.6. Losing Track of Sequent Goal*

During the course of a proof within a theorem prover, the user must perform the proofs of many subgoals that are not directly related to the original goal. This happens in a variety of situations such as when proving type correctness conditions that result from a particular instantiation or when proving conditions that must hold for a particular axiom or lemma to be applied. In PVS, the main goal remains visible throughout all the subgoals unless explicitly removed by the user or transformed by the decision procedures. Thus, the user must be careful to remember which of the antecedents or consequents is being disproved or proved, respectively, or else they may find themselves proving a subgoal twice in the same proof chain. This is particularly relevant after the decision procedures have been invoked since subgoals may be generated that have little relation to the original subgoal that they were invoked on.

For the most part, it is the responsibility of the user to keep track of the subgoal that is being proved. To assist the user, PVS uses curly brackets to denote formulas that were affected by the previous command and square brackets to denote formulas that were unchanged. To prevent excessive rewriting in the PVS strategies developed, whenever a command that performs rewriting, such as grind, is invoked and fails, the proof is always backtracked to a more readable sequent as mentioned

in section 10.1.2. Additionally, grind is only used when necessary. In all other cases, simpler decision procedures, which do not rewrite the sequent excessively, are used.

*10.1.4.7. Reckless Invocation of Decision Procedures*

The decision procedures of a theorem prover can take a significant amount of time to finish a provable subgoal. This time can be increased dramatically by invoking them on an unprovable subgoal. Thus, if the user invokes these procedures carelessly and without regard as to whether they will be able to complete the proof, all of the time used by the decision procedures can be wasted. Thus, it is important for the user to understand when the decision procedures can be invoked most effectively and to only invoke them in those instances.

In general, it is the user's responsibility to understand the capabilities of the decision procedures so that they are not invoked unnecessarily or prematurely. The PVS strategies that were developed make sure that the decision procedures are invoked only when enough information is present to reasonably expect the proof to be completed.

*10.1.4.8. Unnecessary Subgoal Information*

Even when the decision procedures can be invoked effectively, there is the potential that they will take longer than necessary. Specifically, the execution time of the decision procedures is directly related to the complexity of the information in the subgoal that they are invoked on. When a subgoal contains definitions that expand into complex quantifications or conditional expressions, the decision procedures can spend significant amounts of time attempting to instantiate the quantifiers or splitting the conditional expressions. When this information is unnecessary to the proof, however, the extra time that the decision procedures require to use this information is wasted. Thus, it is important to remove any information that cannot be used or to prevent definitions from being expanded needlessly.

In order to assure that the decision procedures will run as efficiently as possible, the PVS strategies developed that use the decision procedures first eliminate information that they cannot use effectively. Additionally, the definitions that are rewritten by the decision procedures are limited in case any additional expensive definitions are uncovered during rewriting.

*10.1.4.9. Abortion of Proof Attempts*

When an unprovable subgoal (i.e. an error) is discovered during the course of a proof, the user must abort the proof to fix the specification and then begin the proof again or rerun it up to the point at

which it was aborted. In PVS, when a proof is rerun, the complete proof must be rerun. That is, all completed subgoals must be rerun as well as incomplete subgoals. This means that time is wasted on already proved subgoals whenever an error is found in the specification. This is especially critical for ASTRAL specifications where invariants and schedules can be conjunctions of many different properties.

In order to counter the need to rerun completed subgoals on failed proof attempts, the various clauses of an ASTRAL specification are split into collections of simpler properties that infer the whole clause. For example, an invariant clause is split into a set of formulas s_Invariant(i), where each s_Invariant(i) corresponds to a split of the formula splitter. The proof of s_Invariant(i) is then performed separately from the proof of each s_Invariant(j), where $i \neq j$. When one part of the invariant is aborted, the parts that have already been proved do not have to be rerun on the next proof attempt. The proof may still need to be rerun to make sure that the changes made have not invalidated the proof, but it does not have to be rerun each time an error is found in another proof. It can be rerun once all proofs have been completed.

## 10.2. PVS Proofs of ASTRAL Properties

The last stage of analysis consists of proving the critical requirements of the current specification with a mechanical theorem prover. The "Prove" button in the SDE, shown in figure 5.1, translates the current specification into the PVS specification language and invokes the PVS theorem prover. At this point in the analysis, it is hoped that all the errors in the system have been identified by the previous stages, since finding errors in the prover stage is much more costly than in the earlier stages. Although it is not possible to eliminate the burden of learning the basic commands and use of the theorem prover, it is still possible to provide the user with techniques for performing the major steps of a proof sketch within the theorem prover. Most of the high-level reasoning about the proofs has already been done in the proof sketch stage by formulating the "strategies" to prove each property. The techniques provide "tactical" assistance for carrying out the developed strategies. Some of the techniques are hidden, such as lemmas and PVS strategies. The lemmas were developed based on the proofs of many example systems to capture the combinations of axioms and other lemmas that are applied together most often. The PVS strategies were also developed during the proofs of many systems and capture patterns of proof steps that can be automated. Thus, the lemmas and PVS strategies are essentially guidelines that could be encoded within the facilities of the prover.

The primary focus of the theorem proving techniques in this chapter is on simple single-threaded processes. Most of the techniques developed for these processes, however, are also applicable to iterative single-threaded processes and multi-threaded processes as well. Some iterative single-threaded processes and multi-threaded processes exhibit behavior that makes PVS proofs of their properties extremely complex as discussed in section 10.9. As discussed in section 8.2.3, however, simple single-threaded processes are significantly more common than the other two process types.

In the theorem proving stage, the proof sketches constructed in the previous stage are used as a high-level plan of how the theorem prover proofs should be discharged. Each step in the proof sketches will correspond to some set of theorem prover commands. In general, the types of proof sketches that the user can construct are too varied to be able to describe exactly how they can be translated into a proof in the theorem prover. Instead of supporting all of the individual constructs that can appear in a hand proof, techniques were developed to support the major reasoning steps of section 9.2. Thus, it is still necessary for the user to understand the commands and overall usage of the theorem prover.

It is possible to provide some general guidelines as to which PVS commands correspond to which steps in a hand proof. The prover commands that are used most often can be broken down into six functional classifications: structural manipulation, introduction of implicit information, instantiation, skolemization, case splitting, and decision procedure invocation. Structural manipulation refers to the process of rearranging, renaming, simplifying, removing, or revealing terms in a given sequent and is used for a variety of reasons such as allowing further manipulation by other commands, making the sequent more readable, and removing unneeded information to increase the efficiency of the decision procedures. The most commonly used commands of this type are flatten, name, replace, assert, delete, hide, and reveal. The *flatten* command separates conjunctions in the antecedents and disjunctions in the consequents into separate terms. The *name* command introduces a simple name for an expression that can be used in its place. The *replace* command substitutes the terms in an equality expression throughout the sequent. The *assert* command invokes the decision procedures to simplify terms in the sequent. The *delete* command removes given terms from the sequent. The *hide* command is a more cautious form of delete that allows terms to be revealed later in the proof with the *reveal* command.

The introduction of implicit information refers to the process of explicitly adding terms to the given sequent that are currently implicit. The most commonly used commands of this type are the lemma, expand, and typepred commands. The *lemma* command introduces axioms, lemmas, and theorems that are in the scope of the current obligation. The *expand* command expands the given definitions

in the sequent. The *typepred* command introduces type information about a given expression. These commands correspond to justifications in hand proofs such as "by the {name} axiom/lemma", "by the definition of {name}", and "by the type of {name}". Instantiation refers to the process of replacing a quantified variable of an existential quantifier of the consequent or a universal quantifier of the antecedent with a specific expression of the correct type. The *inst* command instantiates a specific quantifier with a given expression. Instantiation is usually implicit in hand proofs. Skolemization refers to the process of introducing skolem constants to represent arbitrary values in the universal quantifiers of the consequent and the existential quantifiers of the antecedent. The *skolem* command skolemizes a specific quantifier with a given name. This corresponds to statements in hand proofs such as "let x be an integer such that P(x)" where P is a quantifier predicate.

Case splitting refers to the process of splitting the current goal into several subgoals. The *split* command separates disjunctions in the antecedent and conjunctions in the consequent and produces a subgoal for each term in the expressions. The *case* command takes a boolean expression and splits the proof into a case such that the expression is true and a case such that it is false. Case splitting corresponds to statements in hand proofs such as "suppose P holds" where P is some predicate. Finally, decision procedure invocation refers to the process of finishing off a proof. The *assert* command is the basic decision procedure upon which other decision procedures are based and uses boolean and arithmetic reasoning to attempt to complete the proof of the current subgoal. The *grind* command is a heavy-duty decision procedure based on assert that performs rewriting, skolemization, and automatic quantifier instantiation. Decision procedure invocation corresponds to statements in hand proofs such as "which is a contradiction" and "which is trivially true".

The complete PVS proof of an ASTRAL property will consist of sequences of these basic prover commands interspersed with the PVS strategies that were developed to support the major reasoning steps of the proof sketch stage. A similar approach can be found in [AH 97], which is discussed in section 4.2.4. In this work, a number of lemmas and PVS strategies were developed to support reasoning about the Timed Automaton Model of section 3.2.1.2 in PVS. Several of the strategies correspond closely with the strategies developed for ASTRAL. Most notably, the last-event and first-event strategies have a function similar to the step-bw and step-fw strategies presented below. This indicates that such strategies are useful for many different real-time specification languages and not just ASTRAL. Although [AH 97] does provide several useful techniques for allowing the PVS proofs to correspond closely to hand proofs, what is lacking is any guidance on how the hand proof is to be constructed as is discussed for ASTRAL in section 9.2.

## 10.3. ASTRAL Lemmas

A number of ASTRAL lemmas have been developed that encapsulate many different axioms and lemmas that appear together often into a single form that can be used in place of the combination. These lemmas are valid regardless of process or property type and are discussed in the following subsections.

### 10.3.1. no_trans_fire

The *no_trans_fire* lemmas state that if no transition is the first to fire in an interval, then no transition fires in the interval. These lemmas allow it to be assumed that no transition fires in the interval up until the point at which the first transition fires, which is crucial to avoiding repetition of work when applying the vars_no_change axiom. There are several variations of this lemma depending on whether the interval includes or excludes the end of the interval, and whether or not the beginning of the interval is the end of some transition. Usually, it is necessary to show that a transition does not fire up until another time that some transition is to fire. When the interval is represented by a skolem time constant, however, the constant is usually already limited to be less than the end of the interval. In this case, the no_trans_fire interval should include the constant, otherwise a portion of the interval will be excluded. When the end of the interval is not to be included, the "_lt" variation should be used. For both of these variations, there are two variations depending on whether or not the beginning of the interval is the end of some transition. If it is, then it can be assumed that the variables do not change up until the point it is shown that no transition fires. When a transition ends at the beginning of the interval, the "_vnc" variation should be used. The no_trans_fire_lt and no_trans_fire_vnc variations are shown in figure 10.3.1.

The no_trans_fire lemmas hold because transitions have non-null duration, thus any interval of time can be broken down into subintervals of a length less than the smallest duration. In each subinterval, only a single transition can begin execution. If no transition fires in any subinterval, then no transition can fire in the complete interval. In the definition of the no_trans_fire lemmas, however, points in time are used in place of intervals of time. For most transitions, this is equivalent as there will be a specific time at which their entry assertions first hold or at which they have been invoked from the external environment. Since the time domain of ASTRAL is dense, however, transitions can be defined for which it is not possible to determine the exact time they fire. For example, a transition with an entry assertion "now > 1" may fire at any time after time one, but the exact time is unknown since there are infinitely many times in any interval of dense time. Since such a transition

does not fire at any specific time and the no_trans_fire lemmas use points in time in place of intervals, it may be that such a transition is overlooked by the lemma. Upon closer inspection, however, this is not possible because of the assumption that nothing fires up until the point that is proved. By this assumption and the trans_mutex axiom, there must be a time in the proof interval at which the process is idle. The example "now > 1" transition is enabled at every point after time one and the process is idle by the assumption that nothing fires. Thus, by the trans_fire axiom, the transition must fire, so every point after time one is unprovable. In this case, even though the contradiction is not achieved at the exact point that the transition could fire, it is nonetheless still achieved. Therefore, the no_trans_fire lemmas hold for all transitions.

```
no_trans_fire_lt: LEMMA
    (FORALL (t0: time, t3: time):
        t0 < t3 AND
        (FORALL (tr1: transition, t1: time):
            t0 ≤ t1 AND t1 < t3 AND
            (FORALL (tr2: transition, t2: time):
                t0 ≤ t2 AND t2 < t1 IMPLIES
                    NOT Fired(tr2, t2))
                IMPLIES
                NOT Fired(tr1, t1)) IMPLIES
            (FORALL (tr1: transition, t1: time):
                t0 ≤ t1 AND t1 < t3 IMPLIES
                    NOT Fired(tr1, t1)))
```

```
no_trans_fire_vnc: LEMMA
    (FORALL (t0: time, t3: time):
        t0 ≤ t3 AND
        (EXISTS (tr1: transition):
            t0 ≥ Duration(tr1) AND
            Fired(tr1, t0 - Duration(tr1))) AND
            (FORALL (tr1: transition, t1: time):
                t0 ≤ t1 AND t1 ≤ t3 AND
                (FORALL (t2: time):
                    t0 ≤ t2 AND t2 ≤ t1 IMPLIES
                        Vars_No_Change(t0, t2))
                    AND
                (FORALL (tr2: transition, t2: time):
                    t0 ≤ t2 AND t2 < t1 IMPLIES
                        NOT Fired(tr2, t2)) IMPLIES
                NOT Fired(tr1, t1)) IMPLIES
            (FORALL (tr1: transition, t1: time):
                t0 ≤ t1 AND t1 ≤ t3 IMPLIES
                    NOT Fired(tr1, t1)) AND
            (FORALL (t1: time):
                t0 ≤ t1 AND t1 ≤ t3 IMPLIES
                    Vars_No_Change(t0, t1)))
```

Figure 10.3.1: Variations of no_trans_fire

### 10.3.2. trans_mutex_end

The *trans_mutex_end* lemma states that if a transition fired, then no transition can end while it is executing. This lemma is a slight variation of the trans_mutex axiom that is useful in situations where trans_mutex would have to be applied twice to achieve the same effect. Namely, it is useful when the end of a transition tr1 occurs within the execution of another transition tr2, but the start of tr1 may occur either before or after the start of the tr2. If the start of tr1 occurs before the start of tr2,

then this lemma holds by the trans_mutex axiom applied to tr1. If the start of tr1 occurs after the start of tr2, then this lemma holds by trans_mutex applied to tr2.

```
trans_mutex_end: LEMMA
    (FORALL (tr1: transition, t1: time):
        Fired(tr1, t1) IMPLIES
            (FORALL (tr2: transition, t2: time):
                t1 < t2 + Duration(tr2) AND
                t2 + Duration(tr2) < t1 + Duration(tr1) IMPLIES
                    NOT Fired(tr2, t2)))
```

### 10.3.3. idle_or_firing

The *idle_or_firing* lemma states that at any time in the execution of a process, the process is either idling or executing some transition. This lemma is useful whenever it is necessary to break the proof into cases based on the state of the process at some skolem time constant. In particular, this lemma is useful for splitting the state of the process into cases as discussed in section 9.2.4.2.1. This lemma holds trivially since the two cases are negations of each other.

```
idle_or_firing: LEMMA
    (FORALL (t1: time):
        (FORALL (tr2: transition, t2: time):
            t1 - Duration(tr2) < t2 AND t2 < t1 IMPLIES
                NOT Fired(tr2, t2)) OR
        (EXISTS (tr2: transition, t2: time):
            t1 - Duration(tr2) < t2 AND t2 < t1 AND
            Fired(tr2, t2)))
```

### 10.3.4. var_changes

The *var_changes* lemma states that any time a variable of the current process level changes, some transition has ended at the same time. This lemma is useful for quickly deriving that a transition fired at a given time. This lemma holds because by the vars_no_change axiom, variables do not change value in any interval in which a transition does not end, thus it is not possible for a variable to change at any time in such an interval.

```
var_changes: LEMMA
    (FORALL (t1: time):
        Var_Changes(t1) IMPLIES
            (EXISTS (tr1: transition):
                t1 - Duration(tr1) ≥ 0 AND
                Fired(tr1, t1 - Duration(tr1))))
```

### 10.3.5. not_vnc_vc

The *not_vnc_vc* lemma states that if there is some variable that has two different values in some interval, then there is some time in the interval in which a variable has changed and all variables have kept the same value after that time. This lemma is useful for deriving Var_Changes, which can be used in var_changes to derive that a transition has ended. This lemma holds by the definitions of Vars_No_Change, Var_Changes, and Change1.

```
not_vnc_vc: LEMMA
    (FORALL (t0: time, t3: time):
        t0 < t3 AND
        NOT Vars_No_Change(t0, t3) IMPLIES
            (EXISTS (t1: time):
                t0 < t1 AND t1 ≤ t3 AND
                Var_Changes(t1) AND
                (FORALL (t2: time):
                    t1 ≤ t2 AND t2 ≤ t3 IMPLIES
                        Vars_No_Change(t2, t3))))
```

### 10.3.6. not_vc_vnc

The *not_vc_vnc* lemma is the converse of not_vnc_vc and states that if none of the variables have changed value at a particular time, then there is an earlier time after which the variables have the same values. This lemma is useful in the second branch of a case split on Var_Changes. This lemma holds by the definitions of Var_Changes, Vars_No_Change, and Change1.

```
not_vc_vnc: LEMMA
    (FORALL (t3: time):
        t3 > 0 AND
        NOT Var_Changes(t3) IMPLIES
            (EXISTS (t1: time):
                t1 < t3 AND
                (FORALL (t2: time):
                    t1 ≤ t2 AND t2 ≤ t3 IMPLIES
                        Vars_No_Change(t2, t1))))
```

### 10.3.7. ended_last_ended

The *ended_last_ended* lemma states that any time a transition has ended on the current process, there must be some transition that was the last to end. This lemma is useful for stepping backward to the last transition to end. Since no transition ends after the last transition, it is known that the variables cannot change value up until the given time. This lemma holds because only a finite number of transitions can end in any interval since all transitions have a non-null duration and are nonoverlapping.

```
ended_last_ended: LEMMA
    (FORALL (t3: time):
        (EXISTS (tr1: transition, t1: time):
            t1 + Duration(tr1) ≤ t3 AND
            Fired(tr1, t1)) IMPLIES
            (EXISTS (tr1: transition, t1: time):
                t1 + Duration(tr1) ≤ t3 AND
                Fired(tr1, t1) AND
                (FORALL (tr2: transition, t2: time):
                        t1 + Duration(tr1) < t2 + Duration(tr2) AND
                        t2 + Duration(tr2) ≤ t3 IMPLIES
                            NOT Fired(tr2, t2))))
```

### 10.3.8. first_change1

The *first_change1* lemma states that any time a timed predicate has changed in an interval, there must be a time when the predicate first changes in the interval. Note that in dense time, this lemma is not valid for arbitrary predicates. For example, it is not possible to find the first time that the predicate "now > 2" changes to true. This lemma is always valid, however, for changes to variables since variables can only change when some transition ends and there is always a first transition to end in an interval for similar reasons as in ended_last_ended. Thus, this lemma should only be instantiated with the Var_Changes or i_Var_Changes predicates. Thus, any time a variable changes in an interval, there must be a time when a variable first changes in the interval. This lemma is useful for finding the first time that a property can be violated by a change to a variable.

```
first_change1: LEMMA
    (FORALL (vc_av1: [time → T], t0: time, t3: time):
        (EXISTS (t2: time):
            t0 < t2 AND t2 ≤ t3 AND
            Change1(vc_av1, const(t2))(t2)) IMPLIES
            (EXISTS (t2: time):
                t0 < t2 AND t2 ≤ t3 AND
                Change1(vc_av1, const(t2))(t2) AND
                (FORALL (t1: time):
                    t0 < t1 AND t1 < t2 IMPLIES
                        NOT Change1(vc_av1, const(t1))(t1))))
```

### 10.3.9. exists_change1

The *exists_change1* lemma states that if a timed predicate has two different values in an interval, then there is a time that the predicate has changed. This lemma is useful for discharging TCC obligations that require a change to an expression at some time in the past without the need to expand the Change1 operator. These obligations usually result from applications of the Change1

operator without a time argument. This lemma holds by the definition of Change1. An exists_changen lemma is also available to derive Changen.

```
exists_change1: LEMMA
    (FORALL (av1: [time → T], t1: time, t3: time):
        t1 < t3 AND
        av1(t1) ≠ av1(t3) IMPLIES
            (EXISTS (t2: time):
                t1 < t2 AND t2 ≤ t3 AND
                av1(t2) = av1(t3) AND
                Change1(av1, const(t2))(t3)))
```

#### 10.3.10. exists_start1

The *exists_start1* lemma states that any time a transition has fired on the current process, there has been some Start1 of that transition at any time after that point. This lemma is useful for similar reasons as exists_change1 and holds by the definition of Start1. The lemmas exists_end1, exists_call1, exists_startn, exists_endn, and exists_calln are defined similarly, but state that there has been an End1, Call1, Startn, Endn, or Calln, respectively.

```
exists_start1: LEMMA
    (FORALL (tr1: transition, t1: time):
        Fired(tr1, t1) IMPLIES
            (FORALL (t3: time):
                t3 ≥ t1 IMPLIES
                    (EXISTS (t2: time):
                        t1 ≤ t2 AND t2 ≤ t3 AND
                        Start1(Base_Trans(tr1), const(t2))(t3))))
```

### 10.4. General Strategies

A number of PVS strategies have been developed that are applicable in many situations and are not specific to any property classification. All of the PVS strategies developed for ASTRAL can be found in appendix C.

#### 10.4.1. (case-trans tname)

The *case-trans* strategy takes the name of a skolemized transition constant and produces a case statement that can be split into a case for each possible transition value. For example, in the Gate process of the railroad crossing, if tr1 is a skolemized transition constant, (case-trans "tr1") performs the command (case "tr1 = lower OR tr1 = down OR tr1 = raise OR tr1 = up"), which can be split to produce four possible values for tr1. Since the names of the possible transitions depend on the process level associated with the current proof, this strategy is process-dependent, and is generated automatically by the SDE. The theory name of a level l1 of a process p1 was chosen to be p1_L_l1,

thus this name is compared to the *current-theory* variable of PVS to generate the appropriate cases. If case-trans is used in a global proof, it does nothing.

The case-trans strategy is used any time that it is necessary to give a specific value to a skolemized transition constant to continue the proof. Namely, it is used before expanding a transition-specific function such as an entry/exit assertion or a duration.

#### 10.4.2. (astral-expand (&optional (fnums *))

The *astral-expand* strategy takes an optional list of formula numbers in the current sequent and expands certain definitions within those formulas. If the list is not given, all formulas in the current sequent are expanded. The only definitions that are expanded are the simple functions such as boolean and arithmetic operators and simple transition functions (e.g. Exported, Duration, etc.). None of the complex functions such as timed operators (e.g. Start1, End1, etc.) or specification clauses (e.g. Invariant, Exit, etc.) are expanded.

The main use of the astral-expand strategy is to transform the sequent into a form that can be simplified by the built-in mechanisms of PVS. Simplification usually occurs when the Curried boolean and arithmetic operators are given a temporal context, thus become simple boolean and arithmetic operations that can be directly manipulated by PVS or when boolean transition functions are expanded and become simple true or false values. This strategy is most often used after a specification clause is expanded or when a skolem constant is given a specific value.

#### 10.4.3. (astral-expand-clause (&optional (fnums *))

The *astral-expand-clause* strategy is similar to astral-expand, but additional definitions are expanded. In this strategy, the specification clauses and a few additional functions are expanded. After these definitions are expanded, astral-expand is called. This strategy is used to provide more specific information to PVS. It is most often invoked to expand transition entry/exit assertions after a skolemized transition constant has been given a specific value or to expand requirement/assumption clauses.

#### 10.4.4. (astral-expand-all (&optional (fnums *))

The *astral-expand-all* strategy expands all the definitions of the ASTRAL-PVS encoding. This strategy is the least used of the three astral-expand strategies. It is available for whenever it is necessary to expand a set of formulas as much as possible.

### 10.4.5. (delete-bad)

The *delete-bad* strategy attempts to delete any formulas in the current sequent that reference definitions that are too complex for PVS to utilize effectively. These definitions include the ASTRAL timed operators, the "UNIQUE" quantifier, and the Mod and Div operators. In addition, delete-bad deletes any formulas in the current sequent that reference an if-then-else expression based on transition parameters. When such expressions are present in a sequent, PVS can spend a significant amount of time in its decision procedures splitting the proof into a large number of cases without being able to discharge them. By removing such formulas before the decision procedures are invoked, this time can be saved. This strategy is mainly intended for use by other strategies to control running time. Most notably, delete-bad is used in the definition of my-grind, which is the main strategy used by all of the other strategies to complete the complex cases. Note that delete-bad is only effective after the definitions of astral-expand-clause have been expanded and the sequent has been appropriately flattened. Otherwise, it will delete too little if not expanded, or too much if expanded but not flattened.

### 10.4.6. (my-grind (&optional (if-match NIL)))

The *my-grind* strategy is a modified version of the PVS grind strategy that uses delete-bad and grind parameters to decrease running time. First, my-grind uses astral-expand-clause to expand the ASTRAL definitions up to the clause level so that delete-bad will not miss expressions that are hidden in definitions. It then repeatedly tries flatten and assert until no more simplifications can be made. This is done so that delete-bad will not delete terms that are separable from the "bad" terms. In some cases, one of the repeated asserts will complete the proof without grind ever being invoked, which means the proof can be discharged very quickly. If assert does not complete the proof, delete-bad is executed followed by grind. Most of the operators that are removed by delete-bad are also excluded from rewriting by appropriate grind arguments. This is done so that definitions that are not expanded by astral-expand-clause (namely, definitions from the various define clauses of the specification) but that contain these operators, are not expanded by grind. In both the case of delete-bad and the rewrite exclusions, the definitions that are excluded are those that PVS would most likely not be able to use effectively such as the timed operators. When these definitions are expanded, PVS attempts to automatically instantiate quantifiers in the expansion, which increases running time. Since PVS cannot usually instantiate correctly in these situations, excluding the definitions saves significant time. The optional if-match argument corresponds to the if-match argument of grind and when given, it controls the situations in which PVS attempts automatic quantifier instantiation. If if-

match is not given, PVS does not attempt any instantiation. This strategy is used in place of grind in most of the strategies of appendix C.

## 10.5. Inductive Base Case

Most of the ASTRAL proof obligations are inductive on the time domain, thus each of these proof obligations has a base case. In the base case, each property must be shown to hold when the system is first initialized. The *try-base-case* strategy is used to discharge these obligations. The try-base-case strategy introduces the initial_state axiom and then invokes grind. In this case, grind is used in favor of my-grind as it is often possible for PVS to use the timed operator information at time zero given the simplifications that can be performed at that time.

The try-base-case strategy is sufficient in most cases to discharge the base case obligations automatically. Table 10.5 lists the number of base case obligations in each testbed system and the number that were automatically proved using the try-base-case strategy.

| System | Total Base Cases | Proved Base Cases |
|---|---|---|
| Bakery Algorithm | 2 | 2 |
| Cruise Control | 2 | 2 |
| Elevator | 5 | 4 |
| Olympic Boxing | 4 | 3 |
| Phone | 4 | 3 |
| Production Cell | 9 | 8 |
| Railroad Crossing | 3 | 1 |
| Stoplight | 1 | 0 |
| Total | 30 | 23 |

Table 10.5: Results of try-base-case on testbed system properties

There are three main cases in which this strategy fails. The first case is when imported variables are referenced in the property. In this case, it may be necessary to introduce information about the initial state of the other properties with the i_initial_state axiom. Then, the quantified formulas must be instantiated with the correct process types. The second case is when an immediate response is required such as "Call(tr1, now) → Start(tr1, now)". In this case, it is necessary for the user to prove that the required response will occur. This is almost always simpler in the base case than in the inductive cases, since at time zero all processes are idle and the state is completely known. Finally, try-base-case can fail when the base case obligation contains complex definitions or quantifications that cannot be resolved by grind.

## 10.6. Global and Imported Variable Properties

As discussed in section 9.2.3, the proofs of global and imported variable properties can usually be discharged simply by instantiating the appropriate inductive assumptions correctly. Once the proof sketch for such a property has been constructed, the corresponding theorem prover obligation can be proved in almost exactly the same manner. For example, consider the proof sketch of the global schedule of the bakery algorithm discussed in section 9.2.3. The first step in the proof sketch is to suppose that two different processes are in their critical sections at the same time. To set up this situation within the prover, 10 prover commands are needed. These commands consist mostly of skolemizing, expanding, and simplifying various clauses. The next step in the proof sketch is to show that the number of neither of the processes is zero. This step consists of 8 prover commands, which consist of expanding the exported portion of the Proc invariant, instantiating it with both of the processes, and removing unneeded information. At this point, the main goal of the PVS proof looks like the following.

```
[-1]    1 ≤ V1!2
[-2]    V1!2 ≤ n_procs
[-3]    1 ≤ V1!1
[-4]    V1!1 ≤ n_procs
[-5]    i_Schedule(T1!1)
[-6]    T0 < T1!1
{-7}    T1!1 < DELTA + T0
[-8]    i_proc__in_critical(procs(V1!1))(T1!1)
[-9]    i_proc__in_critical(procs(V1!2))(T1!1)
  |-------
{1}     i_proc__choosing(procs(V1!2))(T1!1)
{2}     i_proc__number(procs(V1!2))(T1!1) = 0
{3}     i_proc__choosing(procs(V1!1))(T1!1)
{4}     i_proc__number(procs(V1!1))(T1!1) = 0
[5]     V1!1 = V1!2
```

In this sequent, the two processes, procs(V1!1) and procs(V1!2), are both in their critical sections from antecedents -8 and -9, and neither of their numbers are zero from consequents 2 and 4. The exported portion of the Proc schedule is still in unexpanded form in antecedent -5.

The next step in the proof sketch is to assume that both processes have the same number. Instead of introducing this case explicitly, it is introduced by expanding the schedule, instantiating it with both processes, and then splitting the resulting expressions appropriately. This takes 10 prover commands to accomplish. The final step of the proof sketch is to achieve a contradiction for the case when the numbers are equal and when they are not equal. Each of these is completed with a single prover

command. Thus, 30 prover commands are necessary to complete the PVS proof. This proof is shown in appendix D.

## 10.7. Untimed Properties

### 10.7.1. Transition Entry/Exit Analysis

Two strategies have been developed that correspond to the transition entry/exit analysis techniques of section 9.2.4.1.1. The try-untimed strategy is used for invariant and schedule properties, while try-untimed-con is used for constraint properties. These strategies are based on the general proof obligations rather than using proof obligations specifically written for untimed formulas. Although this possibility was mentioned in section 4.2.5, it was determined that the advantages of using untimed-specific obligations could be achieved more easily by using the general obligations and developing an appropriate untimed strategy. This avoids any possibility for unsoundness that could result from the user applying untimed obligations to timed properties.

*10.7.1.1. (try-untimed (fnum_i fnum_d &optional (do_grind T)))*

The *try-untimed* strategy is the embodiment of the techniques discussed in section 9.2.4.1.1. The basis of the try-untimed strategy is that in the interval T0 to T0 + Δ of the proof obligations, the state variables either stay the same or one or more of them change. If the variables stay the same, then by the inductive hypothesis, the property holds at all times in the interval. If a variable changes during the interval, then by the semantics of ASTRAL, a transition ended at the time of the change. Furthermore, since transitions are nonoverlapping and Δ has been limited to a constant less than the duration of any transition as discussed in section 6.3.3, only a single transition end can occur within the interval. Figure 10.7.1.1 depicts this situation. Let T1 be the time of such an end. Since no transition ended in the interval [T0, T1), the state variables must have stayed the same during that time period, thus the property holds by the inductive hypothesis. Similarly, since no transition ended in the interval (T1, T0 + Δ], the variables are unchanged in that region, thus the property holds in that region if it holds at T1. The bulk of the strategy is thus devoted to proving that the property holds at T1.

To prove this, it must be shown that all transition exit assertions preserve the property, thus the proof is split into a case for each transition and the transition's entry and exit clauses are asserted. In the proof obligations, the start of the transition occurred before T0 by the limitation on Δ, thus the

property held at the start of the transition. From this point, my-grind is invoked to finish the proof, if possible.

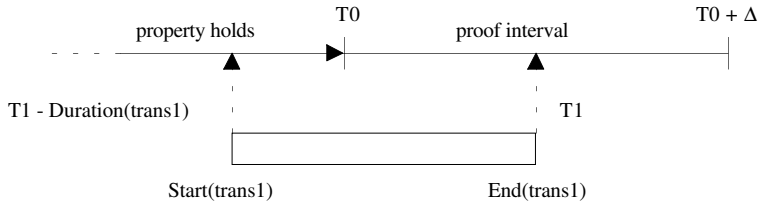



Figure 10.7.1.1: Proof interval

The fnum_i argument is the number of the formula in the sequent that contains the inductive assumption of the proof obligation ("FORALL (T1: time): T1 ≤ T0 IMPLIES Invariant(T1)" or similarly for a schedule property). The fnum_d argument is the number of the formula that contains the limitation on Δ ("FORALL (tr1: transition): Δ < Duration(tr1)"). If do_grind is given as NIL, try-untimed will set up the proof but will not attempt to carry it out.

Table 10.7.1.1 shows the results of using the try-untimed strategy to attempt the proofs of applicable invariant and schedule properties in the testbed systems. In this case, "applicable" means that the property meets the conditions discussed in section 9.2.4.1.1, or in other words, that the property is untimed and only references local state variables. The table shows that over half of the properties that are applicable were automatically discharged by the try-untimed strategy.

| System | Applicable Properties | Proved Properties |
|---|---|---|
| Bakery Algorithm | 5 | 3 |
| Cruise Control | 5 | 5 |
| Elevator | 2 | 2 |
| Olympic Boxing | 1 | 1 |
| Phone | 17 | 10 |
| Production Cell | 14 | 8 |
| Railroad Crossing | 0 | 0 |
| Stoplight | 11 | 0 |
| Total | 55 | 29 |

Table 10.7.1.1: Results of try-untimed on testbed system properties

*10.7.1.2. (try-untimed-con (&optional (do_grind T)))*

Untimed constraints are similar to untimed invariants except that they must only hold at the times a transition ends. Thus, the *try-untimed-con* strategy is similar to the try-untimed strategy except that there is no need to reason about the proof interval. Instead, it can be assumed that a transition ends

at some time T1 and then prove that the entry and exit assertion of each transition preserves the constraint property. Table 10.7.1.2 shows the results of using the try-untimed-con strategy to attempt the proofs of applicable constraint properties in the testbed systems. In this case, all of the constraints in the testbed systems are applicable. The table shows that half of the constraints were discharged automatically by the try-untimed-con strategy.

| System | Applicable Properties | Proved Properties |
|---|---|---|
| Bakery Algorithm | 0 | 0 |
| Cruise Control | 0 | 0 |
| Elevator | 2 | 2 |
| Olympic Boxing | 1 | 1 |
| Phone | 3 | 3 |
| Production Cell | 0 | 0 |
| Railroad Crossing | 0 | 0 |
| Stoplight | 6 | 0 |
| Total | 12 | 6 |

Table 10.7.1.2: Results of try-untimed-con on testbed system properties

### 10.7.2. Transition Sequence Analysis

A side benefit of the try-untimed strategy is that even when it fails, it is still advantageous for the user to run it because usually only very difficult cases will be left for the user to prove. When the strategy fails, it is due to one of three reasons. The first reason is that the user invoked the strategy on a timed property or one that involves imported variables. In this case, it is likely that most of the cases will fail, since try-untimed was not intended to deal with these types of properties. The second reason is that one or more transitions do not preserve the property. In this case, the user knows the exact transitions that failed since PVS will require further interaction to complete those cases. The user can correct the specification before continuing with other proofs. The last reason, which will be the most likely, is that it failed because there was not enough information in the entry assertion of a transition to prove the property. Usually, this occurs when the value of a variable in the formula to be proved is not explicitly stated in the entry assertion of the transition, but instead is implied by the sequences preceding that particular transition. The cases that are left consist of the transitions for which transition sequence analysis is necessary.

*10.7.2.1. (step-bw-indeterminate (t_from &optional (fnum_i NIL) (fnum_d NIL) (do_grind T)))*

The *step-bw-indeterminate* strategy is the embodiment of the techniques of section 9.2.4.1.2. This strategy takes a time t_from and performs the necessary proof steps to derive the transitions that

could have ended prior to this time as shown in figure 10.7.2.1. The optional arguments are identical to those of try-untimed. It is first shown that there is a transition that ended before t_from. The strategy attempts to discharge this subgoal by achieving a contradiction between the initial state and the state at t_from. This is possible because if no transition ends before t_from, then the variables could not have changed value since the initial state. The strategy invokes my-grind, which in most cases will be sufficient to finish the proof. In the cases that it is not sufficient, the user must complete the proof by expanding timed operators or introducing relevant assumptions that require some transition to end between the initial state and t_from. The proof sketch for the current property will have which timed operators and/or assumptions were used to complete the proof.

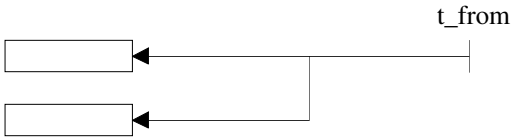



Figure 10.7.2.1: An indeterminate backward step

Since there is a transition that ended before t_from, there is a transition that ends last by ended_last_ended. After it has been determined that some transition has ended last, the strategy then attempts to eliminate as many of the possible predecessors as possible by achieving a contradiction between the entry/exit of those transitions and the state at t_from. This step is performed in a similar manner to proving the sequence generator obligations and fails for similar reasons. In this case, however, more information, such as the inductive invariant/schedule, is available to PVS, which makes this step more likely to succeed. When it fails, however, the user must prove the contradictions manually by expanding timed operators and/or stepping backward appropriately.

*10.7.2.2. Is_Predecessor*

In many cases, step-bw-indeterminate may have to be used more than once from the start of the same transition. Rather than using step-bw-indeterminate every time and potentially repeating the same cases that fail in the strategy, it is oftentimes worthwhile to prove that a particular transition is the predecessor of another once and then use it in all the proofs. Is_Predecessor is given a time, t_succ, two transitions, pred and succ, and a boolean value, is_first. Is_Predecessor states that if succ fires at t_succ, then pred is the last transition to fire, meaning that the variables do not change between the time it ends and the start of succ. The value of is_first indicates whether succ can be the first transition to fire after the initial state. If this is possible, then it must be shown that some transition has fired before t_succ in order for pred to be the last transition to fire.

```
Is_Predecessor(t_succ: astral_basic.time, pred: transition,
        succ: transition, is_first: bool): bool =
    Fired(succ, t_succ) AND
    (is_first IMPLIES
        (EXISTS (tr1: transition, t1: time):
            t1 + Duration(tr1) ≤ t_succ AND
            Fired(tr1, t1))) IMPLIES
        (EXISTS (t1: time):
            t1 + Duration(pred) ≤ t_succ AND
            Fired(pred, t1) AND
            (FORALL (t2: time):
                t1 + Duration(pred) ≤ t2 AND
                t2 < t_succ + Duration(succ) IMPLIES
                    Vars_No_Change(t1 + Duration(pred), t2)) AND
            (FORALL (tr2: transition, t2: time):
                t1 < t2 AND t2 < t_succ IMPLIES
                    NOT Fired(tr2, t2)))
```

For example, in the P_Robot process of the production cell, the user might make the following declaration before the invariant obligations.

```
is_pred_arm2_retracted: THEOREM
    (FORALL (t1: time):
        Is_Predecessor(t1, retract_arm2, arm2_retracted, FALSE))
```

This theorem states that retract_arm2 is the predecessor of arm2_retracted. It may be that a transition is the predecessor of another only when certain conditions hold. These conditions may include invariant properties, state variable restrictions, etc. In that case, the conditions can be added to the user definition. For example, the predecessor of retract_arm2 depends on the value of arm2_has_object. In the following definition, arm2_drop is the predecessor of retract_arm2 when arm2_has_object is false at the start of arm2_retracted.

```
is_pred_retract_arm2: THEOREM
    (FORALL (t1: time):
        NOT arm2_has_object(t1) IMPLIES
            Is_Predecessor(t1, arm2_drop, retract_arm2, FALSE))
```

*10.7.2.3. (is-pred-indeterminate (tr_from t_from))*

By defining Is_Predecessor in a standard form, it is possible to define a strategy that can be used to discharge Is_Predecessor obligations. The *is-pred-indeterminate* strategy is an example of such a strategy. This strategy must be given a transition, tr_from, and the time it fired, t_from. It then uses step-bw-indeterminate to find the transition that fires immediately before the given transition. The other clauses of the Is_Predecessor definition are trivially derived from the information produced by step-bw-indeterminate.

*10.7.2.4. (expand-is-pred-indeterminate (fnum t_from tr_to &optional (f_hide T)))*

Once the appropriate Is_Predecessor declarations have been made, the user can utilize these declarations within other proofs. In most cases, a similar sequence of steps will be performed whenever an Is_Predecessor definition is expanded. Namely, the Vars_No_Change expression is instantiated with the time succ fired, and the entry/exit information of pred is introduced. The *expand-is-pred-indeterminate* strategy is an example of how the Is_Predecessor information can be expanded automatically. This strategy takes a formula number in the current sequent, fnum, that has an Is_Predecessor declaration, the time that the declaration is instantiated with, t_from, and the transition that is the pred transition of the declaration, tr_to. In addition, an optional argument, f_hide, can be given, which when true hides the fact that nothing fires between succ and pred. The strategy instantiates the Vars_No_Change expression with t_from and introduces the entry/exit information of tr_to.

*10.7.2.5. PVS Transition Sequence Analysis Proof*

As an example of using the proof sketch of an untimed property to perform the corresponding PVS proof, consider the proof sketch of the Phone property of the phone system discussed in section 9.2.4.1.2. The first step in the proof sketch is to show that Start_Busytone and Start_Ringback are the only transitions that assert Busytone and Ringback, respectively. This is accomplished using 12 prover commands. The key command in this sequence is the try-untimed command, which eliminates all the transition cases except the Start_Busytone and Start_Ringback cases. The other commands consist of skolemizing, expanding, and simplifying various clauses. In the following, only the Start_Busytone case will be considered. The next step in the proof sketch is to determine the transitions that can precede Start_Busytone. This step takes 3 prover commands, the most important of which is the step-bw-indeterminate command, which eliminates all the transition cases besides the Enter_Digit and Start_Ringback cases. After step-bw-indeterminate is applied, the main goal of the PVS proof is shown in figure 10.7.2.5.

In this sequent, t1!1 - Duration(start_busytone) is the time at which Start_Busytone fires as shown in antecedent -12 while t1!2 is the time at which Enter_Digit fires as shown in antecedent -5. At the start of Start_Busytone, the variables keep the values they had at the end of Enter_Digit by antecedent -3. Similarly, at T1!1, the variables keep the values they had at the end of Start_Busytone by antecedent -15. Since Start_Busytone does not change the value of Ringback, the goal that ~Ringback holds at T1!1 in antecedent -22 holds if ~Ringback holds at the end of Enter_Digit.

```
{-1}   Exit(enter_digit, Duration(enter_digit) + t1!2)
{-2}   Entry(enter_digit, t1!2)
{-3}   Vars_No_Change(Duration(enter_digit) + t1!2, t1!1 - Duration(start_busytone))
[-4]   Duration(enter_digit) + t1!2 ≤ t1!1 - Duration(start_busytone)
[-5]   Fired(enter_digit, t1!2)
[-6]   FORALL (tr2: transition, t2: time):
           Duration(enter_digit) + t1!2 < t2 + Duration(tr2)
           AND t2 + Duration(tr2) ≤ t1!1 - Duration(start_busytone)
           IMPLIES NOT Fired(tr2, t2)
[-7]   Schedule(Duration(enter_digit) + t1!2) AND Invariant(Duration(enter_digit) + t1!2)
[-8]   Schedule(t1!2) AND Invariant(t1!2)
[-9]   Exit(start_busytone, t1!1)
[-10]  Entry(start_busytone, t1!1 - Duration(start_busytone))
[-11]  t1!1 - Duration(start_busytone) ≥ 0
[-12]  Fired(start_busytone, t1!1 - Duration(start_busytone))
[-13]  T0 < t1!1
[-14]  t1!1 ≤ T1!1
[-15]  Vars_No_Change(t1!1, T1!1)
[-16]  Schedule(t1!1 - Duration(start_busytone))
[-17]  Invariant(t1!1 - Duration(start_busytone))
[-18]  DELTA < Duration(start_busytone)
[-19]  T0 < T1!1
[-20]  T1!1 < DELTA + T0
[-21]  busytone(T1!1)
[-22]  ringback(T1!1)
  |-------
```

Figure 10.7.2.5: Main goal after step-bw-indeterminate

If Enter_Digit fires before Start_Ringback, there are two cases in the proof sketch. If Phone_State(P) is Ready_To_Dial, then Dialtone holds, which implies ~Ringback holds from the schedule. This is achieved using 4 prover commands. The main command in this step is my-grind. If Phone_State(P) is Dialing, then the previous value of Phone_State(P) was Ready_To_Dial and Enter_Digit has executed at some time between the change to Ready_To_Dial and the change to Dialing from the imported variable clause. This information is introduced into the proof using 22 prover commands. The bulk of these commands consist of skolemizing, expanding, instantiating, and simplifying various clauses. To show that ~Ringback holds at the time Enter_Digit fires, 6 prover commands are used, which consist of expanding the schedule and simplifying.

The next step in the proof sketch of the Enter_Digit case is to show that the only way Ringback could change to true is if Start_Ringback fires. This is achieved using 12 prover commands. The key to this step is to assume that Ringback is true at some time after the start of the earlier Enter_Digit and then use the exists_change1 lemma to derive that a change of Ringback occurred. After this information is present, change-fire is invoked to find the transitions that can bring about such a

change, which in this case is only the Start_Ringback transition. The final step in the proof sketch of the Enter_Digit case is to show that Start_Ringback could not have fired after the start of the earlier Enter_Digit since Phone_State(P) was only Ready_To_Dial and Busy after that time. This is achieved using 22 prover commands. The bulk of these commands consist of skolemizing, expanding, instantiating, and simplifying various clauses.

In the case that Start_Ringback fires before Start_Busytone, it is known that the previous value of Phone_State(P) was Dialing by the imported variable clause since the value of Phone_State(P) at the time Start_Busytone fires is Busy from its entry assertion. This information is obtained using 17 prover commands in a similar manner as the related step above. The final step in the proof sketch is to show that Start_Ringback cannot be a predecessor of Start_Busytone because of the entry assertion of Start_Ringback, which requires that Phone_State(P) is Waiting. This is achieved using 33 prover commands in a similar manner as the related step above. Thus, 131 prover commands are necessary to complete the PVS proof of the Start_Busytone case. This proof is shown in appendix D. A similar number of commands could be used to complete the Start_Ringback case.

### 10.7.3. Timed Operator Analysis

An example of a strategy developed for dealing with untimed properties that contain timed operators as discussed in section 9.2.4.1.3 is the change-fire strategy. This strategy is used for properties that reference the change operator in the antecedent.

#### *10.7.3.1. (change-fire (fnum_c vname_c time_c &optional (do_grind T)))*

The *change-fire* strategy is used to determine the transitions that could produce a given change to a variable. This strategy takes a formula number in the current sequent, fnum_c, that contains a Change1 expression, the name of the variable that changed, vname_c, and the time at which it changed, time_c. It is first shown that some transition ended at time_c using var_changes. The strategy then attempts to eliminate as many transitions as possible by achieving a contradiction between the entry/exit of those transitions and the variable change that occurred. Although the transitions that do not reference the variable unprimed in their exit assertions can almost always be eliminated, it is not always possible to eliminate the transitions that change the variable, but not to the desired value. For example, in the proof of the property "Change(number, now) & number = 0 $\rightarrow$ ~in_critical" of the Proc process of the bakery algorithm, the strategy cannot eliminate the set_number transition due to the quantification used in the exit assertion to constrain the new value of number. In these cases, the user must eliminate the possibilities manually.

*10.7.3.2. PVS Timed Operator Analysis Proof*

Consider the proof sketch of the property of the Proc process of the bakery algorithm discussed in section 9.2.4.1.3. The first step in the proof sketch is to find the transitions that can the change the number variable appropriately. This step requires 6 prover commands. Besides commands for skolemizing, expanding, and simplifying various clauses, the main command of this step is the change-fire command. The only transition that could not be eliminated by change-fire is the set_number transition. After change-fire is applied, the main goal of the PVS proof looks like the following.

```
{-1}   Exit(set_number, T1!1)
{-2}   Entry(set_number, T1!1 - Duration(set_number))
[-3]   number(T1!1 - Duration(set_number)) ≠ number(T1!1)
[-4]   T1!1 - Duration(set_number) ≥ 0
[-5]   Fired(set_number, T1!1 - Duration(set_number))
[-6]   T0 < T1!1
{-7}   T1!1 < DELTA + T0
[-8]   t1!1 < T1!1
[-9]   FORALL (t3: time):
           t1!1 ≤ t3 AND t3 < T1!1 IMPLIES NOT number(t3) = number(T1!1)
[-10]  FORALL (t2: time):
           T1!1 ≤ t2 AND t2 ≤ T1!1 IMPLIES number(t2) = number(T1!1)
  |-------
[1]    number(T1!1) = 0
[2]    number(T1!1) ≥ 1 + i_proc__number(procs(V1!1))(T1!1 - exec_time)
```

In this sequent, the entry and exit assertions of set_number appear in antecedents -2 and -1, respectively. The change operator has been expanded into antecedents -8, -9, and -10. The requirement portion of the property is in consequents 1 and 2.

The final step of the proof sketch is to show that the exit assertion of set_number satisfies the given requirement. This takes 3 prover commands, which consist of expanding the exit assertion of set_number, flattening it, then instantiating it with the correct procs instance. Thus, 9 prover commands are necessary to complete the PVS proof. This proof is shown in appendix D.

The change-fire strategy is an example of how timed operator analysis can be automated within the prover. For example, the above property of the Proc process was fully proved for all transitions except set_number using the change-fire strategy. In general, this strategy can be applied in any proof to find the transition that brought about the desired change.

## 10.8. Timed Properties

The proof sketches of timed properties consist of using forward and backward steps to obtain enough information to prove the desired property. Thus, PVS strategies were developed that correspond closely to the techniques used during the construction of the proof sketch of a property. It is the user's job to introduce the appropriate assumptions at the appropriate times as given by the proofs sketches of the various properties.

### 10.8.1. Forward Properties

*10.8.1.1. (step-fw-delay (tr_from t_from tr_to t_to &optional (add_entry NIL) (fnum_i NIL) (do_grind T)))*

The *step-fw-delay* strategy is the embodiment of the timed forward steps of section 9.2.2 that have nonzero delay. This strategy takes a "source" transition, tr_from, the time it fired, t_from, a "destination" transition, tr_to, and the time it is to fire, t_to. The optional arguments are similar to those of try-untimed except for the add_entry argument, which introduces the entry assertion of tr_from into the original sequent. The strategy performs the necessary proof steps to show that tr_to is the next transition to fire and that it fires at t_to as shown in figure 10.8.1.1. In order to show this, four subgoals must be proved.

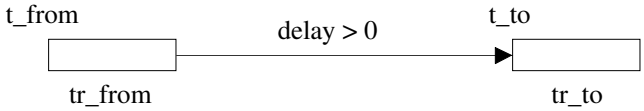



Figure 10.8.1.1: A delayed forward step

1. t_to - (t_from + Duration(tr_from)) > 0

This strategy is only meant to be used when there is a delay between the end of tr_from and the start of tr_to. If there is no delay, then the step-fw-immediate strategy should be used instead since more of it can be fully automated. The strategy attempts to discharge this subgoal with the assert command. This may or may not finish the subgoal depending on the forms of t_from and t_to. If either of these expressions has a complex form, it may be necessary for the user to complete this proof by introducing type predicates for the terms in each expression.

2. no transition fires from t_from + Duration(tr_from) until t_to

In order to show that no transition can fire at any given point in this interval, a contradiction must be achieved between the variable values at that point and the entry assertion of each transition. In order

to determine the variable values, however, it must be shown that no transition fires before that point, so that the vars_no_change axiom can be applied. This results in a circular situation that can only be resolved with complex case analysis. To avoid this situation, the no_trans_fire_vnc lemma is used. This lemma relies on the fact that if no transition is the first to fire in the interval, then no transition fires in the interval. By using this lemma, the proof is reduced to proving that no transition is enabled if the variable values do not change after the beginning of the interval. It is the user's responsibility to complete the cases that are not discharged with my-grind by expanding timed operators and/or invoking the appropriate assumptions.

3. tr_to is enabled at t_to

Since no transition fires from t_from + Duration(tr_from) until t_to, the variables remain unchanged in this interval. Thus, the entry/exit information of tr_from can be used to show that tr_to is enabled. Since tr_to is delayed, however, its entry assertion depends on either the current time, the other processes in the system, or the external environment, thus there is almost no chance that PVS would be able to discharge this subgoal. It is therefore the user's responsibility to complete this proof by expanding timed operators and/or invoking the appropriate assumptions.

4. no other transition is enabled at t_to

The strategy attempts to discharge this subgoal by showing that if a transition besides tr_to is enabled at t_to, a contradiction can be achieved between the exit assertion of tr_from and the entry assertion of the transition. This step is performed in a similar manner to proving the sequence generator obligations and fails for similar reasons. In this case, however, more information, such as the inductive invariant/schedule, is available to PVS, which makes this step more likely to succeed. When it fails, however, the user must prove the contradictions manually by expanding timed operators and/or stepping backwards appropriately.

These four subgoals are sufficient to show that tr_to fires at t_to. The first subgoal shows that there is a nonzero delay between the end of tr_from and the start of tr_to. The last three subgoals are used to invoke the trans_fire axiom, which states that if the process is idle and some transition is enabled, then some transition fires. The second subgoal shows that the process is idle at t_to. It is also used to show that the variables do not change value between the end of tr_from and t_to. The third subgoal shows that some transition is enabled at t_to. Finally, the last subgoal is needed to show that the transition that actually fires at t_to is tr_to.

*10.8.1.2. (step-fw-immediate (tr_from t_from tr_to &optional (add_entry NIL) (fnum_i NIL) (do_grind T)))*

The *step-fw-immediate* strategy is the embodiment of the timed forward steps of section 9.2.2 that have zero delay. Like step-fw-delay, this strategy takes a "source" transition, tr_from, the time it fired, t_from, a "destination" transition, tr_to, and identical optional arguments. The time that tr_to fires is t_from + Duration(tr_from) since this strategy only considers the cases with zero delay. This strategy performs the necessary proof steps to show that tr_to is the next transition to fire and that it fires at t_from + Duration(tr_from) as shown in figure 10.8.1.2. In order to show this, two subgoals must be proved, which correspond to the last two subgoals of step-fw-delay.

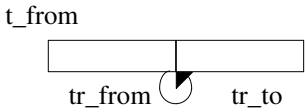



Figure 10.8.1.2: An immediate forward step

Since tr_to is to fire immediately after tr_from, tr_to is most likely an L-type transition. This fact simplifies the proof considerably and allows step-fw-immediate to be automated more fully than step-fw-delay. Of the two subgoals attempted by this strategy, the only one that may require user interaction is the proof that no other transition is enabled at t_to. Like its step-fw-delay counterpart, this step may be discharged by expanding timed operators and/or stepping backward appropriately.

*10.8.1.3. Is_Successor*

Like step-bw-indeterminate, a delayed successor can be declared once as a theorem similar to the declarations in section 10.7.2.2. In this case, however, the Is_Successor definition requires an exact delay to be given and does not require an is_first argument since this is not an issue in forward steps.

```
Is_Successor(t_pred: astral_basic.time, pred: transition,
        succ: transition, delay: nonneg_real): bool =
    Fired(pred, t_pred) IMPLIES
        Fired(succ, t_pred + Duration(pred) + delay) AND
        (FORALL (t2: time):
            t_pred + Duration(pred) ≤ t2 AND
            t2 < t_pred + Duration(pred) + delay + Duration(succ) IMPLIES
                Vars_No_Change(t_pred + Duration(pred), t2)) AND
        (FORALL (tr2: transition, t2: time):
            t_pred < t2 AND
            t2 < t_pred + Duration(pred) + delay IMPLIES
                NOT Fired(tr2, t2))
```

### 10.8.2. Backward Properties

For forward steps, it is possible to directly infer that a transition fires after the given transition. For backward steps, however, this information must be indirectly inferred by the fact that the given transition has already fired after the desired transition.

*10.8.2.1. (step-bw-delay (tr_from t_from tr_to t_to &optional (add_entry NIL) (fnum_i NIL) (do_grind T)))*

The *step-bw-delay* strategy is the embodiment of the timed backward steps of section 9.2.2 that have nonzero delay. This strategy takes a "source" transition, tr_from, the time it fired, t_from, a "destination" transition, tr_to, the time it is to end, t_to, and optional arguments identical to those of step-fw-delay. It then performs the necessary proof steps to show that tr_to is the last transition to end and that it ends at t_to as shown in figure 10.8.2.1. In order to show this, five subgoals must be proved.

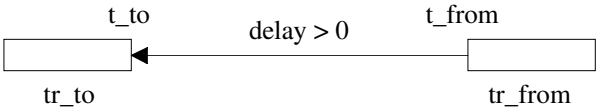



Figure 10.8.2.1: A delayed backward step

1. t_from - t_to > 0

This strategy is only meant to be used when there is a delay between the start of tr_from and the end of tr_to. If there is no delay, then the step-bw-immediate strategy should be used instead since more of it can be fully automated. The strategy attempts to discharge this subgoal with the assert command. This may or may not finish the proof depending on the forms of t_from and t_to. If either of these expressions has a complex form, it may be necessary for the user to complete this proof by introducing type predicates for the terms in each expression.

2. some transition ended before t_from

The strategy attempts to discharge this subgoal by achieving a contradiction between the initial state and the state at t_from. This is possible because if no transition ends before t_from, then the variables could not have changed value since the initial state. The strategy invokes my-grind, which in most cases will be sufficient to finish the proof. In the cases where it is not sufficient the user must complete the proof by expanding timed operators or introducing relevant assumptions that require some transition to end between the initial state and t_from. The proof sketch for the associated property will have which timed operators and/or assumptions were used to complete the proof.

3. the transition that ended last was tr_to

Since there is a transition that ended before t_from, there is a transition that ends last by ended_last_ended. The strategy attempts to discharge this subgoal by showing that the transitions besides tr_to could not have been the last to end or else a contradiction could be achieved between the entry/exit of those transitions and the entry of tr_from. This step is performed in a similar manner to proving the sequence generator obligations and fails for similar reasons. In this case, however, more information, such as the inductive invariant/schedule, is available to PVS, which makes this step more likely to succeed. When it fails, however, the user must prove the contradictions manually by expanding timed operators and/or stepping backwards appropriately.

4. tr_to did not end before t_to

Once it is shown that tr_to was the last transition to end, it must be shown that tr_to ended at t_to. The strategy attempts to show that if tr_to ended earlier than t_to, then tr_from would fire earlier than t_from. In this case, if tr_from ended before t_from, then a contradiction is achieved with the fact that nothing ended between the end of tr_to and t_from. If tr_from did not end before t_from, then by trans_mutex, tr_from could not fire at t_from. The main thing the user must prove in this step is that tr_from is enabled after the given delay elapses from the end of tr_to. This will usually require expanding timed operators in the entry assertion of tr_from.

5. tr_to did not end after t_to

The strategy attempts to discharge this subgoal in a similar manner to the previous step. In this case, it must be shown that if tr_to ends later than t_to, then tr_from could not be enabled (hence fire) at t_from. Since the entry assertion of tr_from is most likely dependent on timed operators, the user must expand these operators appropriately.

If tr_from is not delayed due to timed operators, then it must be delayed by other processes or the external environment. In these cases, it must be shown that the change to the operating environment was delayed in response to some change made by tr_to. Otherwise, it will not be possible to prove this subgoal because the operating environment must have changed at t_from, which means that tr_from will still be enabled.

These five subgoals are sufficient to show that tr_to ends at t_to. The first subgoal shows that there is a nonzero delay between the end of tr_to and the start of tr_from. The second subgoal shows that there has been some transition that has fired in the execution history of the process. If no transition has fired, then tr_to cannot possibly have fired before tr_from. The third subgoal shows that the last

transition to end was tr_to. Finally, the last two subgoals show that tr_to did not fire too early or too late, respectively.

*10.8.2.2. (step-bw-immediate (tr_from t_from tr_to &optional (add_entry NIL) (fnum_i NIL) (do_grind T)))*

The *step-bw-immediate* strategy is the embodiment of the timed backward steps of section 9.2.2 that have zero delay. Like step-bw-delay, this strategy takes a "source" transition, tr_from, the time it fired, t_from, a "destination" transition, tr_to, and identical optional arguments. The time that tr_to ends is t_from since this strategy only considers the cases with zero delay. This strategy performs the necessary proof steps to show that tr_to is the last transition to end and that it ends at t_from as shown in figure 10.8.2.2. In order to show this, four subgoals must be proved, which correspond to the last four subgoals of step-bw-delay.

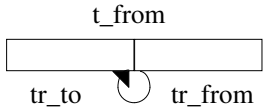



Figure 10.8.2.2: An immediate backward step

Since tr_from is to fire immediately after tr_to, tr_from is most likely an L-type transition. This fact simplifies the proof considerably and allows step-bw-immediate to be automated more fully than step-bw-delay. Of the four subgoals attempted by this strategy, the only one that may require user interaction is the proof that tr_to is the last transition to end. Like its step-bw-delay counterpart, this step may be discharged by expanding timed operators and/or stepping backward appropriately.

*10.8.2.3. Is_Predecessor*

Is_Predecessor is the backward equivalent of Is_Successor. In this case, it is necessary to give an is_first argument as in the indeterminate version of Is_Predecessor as described in section 10.7.2.2

```
Is_Predecessor(t_succ: astral_basic.time, pred: transition,
        succ: transition, delay: nonneg_real, is_first: bool): bool =
    Fired(succ, t_succ) AND
    (is_first IMPLIES
        (EXISTS (tr1: transition, t1: time):
            t1 + Duration(tr1) ≤ t_succ AND
            Fired(tr1, t1))) IMPLIES
        t_succ - delay - Duration(pred) ≥ 0 AND
        Fired(pred, t_succ - delay - Duration(pred)) AND
        (FORALL (t2: time):
            t_succ - delay ≤ t2 AND
            t2 < t_succ + Duration(succ) IMPLIES
                Vars_No_Change(t_succ - delay, t2)) AND
```

```
(FORALL (tr2: transition, t2: time):
    t_succ - delay - Duration(pred) < t2 AND
    t2 < t_succ IMPLIES
        NOT Fired(tr2, t2))
```

*10.8.2.4. (is-pred-immediate (tr_from t_from tr_to))*

The *is-pred-immediate* strategy is an example of a strategy that can be used to discharge Is_Predecessor obligations. This strategy must be given two transitions tr_from and tr_to, which correspond to succ and pred, respectively, and the time tr_from fired, which corresponds to t_succ. It then uses step-bw-immediate to show that tr_to is the immediate predecessor of tr_from. The other clauses of the Is_Predecessor definition are trivially derived from the information produced by step-bw-immediate. Strategies to discharge the other timed variants of Is_Predecessor and Is_Successor can be defined in a similar manner using step-bw-delay, step-fw-immediate, and step-fw-delay.

*10.8.2.5. PVS Liveness Property Proof*

As an example of using the proof sketch of a liveness property to perform the corresponding PVS proof, consider the proof sketch of the Gate property of the railroad crossing discussed in section 9.2.4.2.1. The first step in the proof sketch is to split the proof into cases based on the operating environment and the local state. This step takes 43 commands to perform. The bulk of these commands deal with transforming the change expression Change(s.train_in_R) to the more usable form Change(s.train_in_R, t), where t is the time of the last change. The idle_or_firing lemma is used to split the proof at the time that the change occurs. Then the firing case is split into a case for each transition using case-trans. The other commands consist of skolemizing, expanding, and simplifying various clauses.

Consider the case of the up transition. The next step in the proof sketch of the up case is to show that lower fires immediately at the end of up. This is accomplished with 16 prover commands, the most important of which is the step-fw-immediate command. The other commands are used to prove the TCC resulting from the application of step-fw-immediate. The next step in the proof sketch is to show that down fires lower_time after the end of lower. This is achieved with 137 prover commands. The key to this step is the use of the step-fw-delay command. The rest of the commands are used to prove the subgoals that are generated from this strategy. These subgoals consist of showing that raise is not enabled when down fires, that lower is enabled, and that neither lower nor raise is enabled before lower_time after the end of up. After step-fw-delay is applied, the main goal of the proof looks like the following.

```
{-1}   Entry(down, Duration(lower) + lower_time + t2!1 + up_dur)
{-2}   Fired(down, Duration(lower) + lower_time + t2!1 + up_dur)
[-3]   FORALL (tr1: transition, t1: time):
           Duration(lower) + t2!1 + up_dur ≤ t1
           AND t1 < Duration(lower) + lower_time + t2!1 + up_dur
               IMPLIES NOT Fired(tr1, t1)
{-4}   Vars_No_Change(Duration(lower) + t2!1 + up_dur,
           Duration(lower) + lower_time + t2!1 + up_dur)
[-5]   lower_time > 0
[-6]   lower_dur > 0
[-7]   up_dur > 0
[-8]   down_dur > 0
[-9]   lower_time > 0
[-10]  dist_r_to_i / max_speed
           ≥ down_dur + lower_dur + lower_time + response_time + up_dur
[-11]  Exit(lower, Duration(lower) + t2!1 + up_dur)
[-12]  Entry(lower, t2!1 + up_dur)
[-13]  Fired(lower, t2!1 + up_dur)
[-14]  Exit(up, (t2!1 + up_dur))
[-15]  Entry(up, t2!1)
[-16]  ct - up_dur < t2!1
[-17]  t2!1 < ct
[-18]  Fired(up, t2!1)
[-19]  ct ≥ 0
[-20]  ct ≤ T1!1
[-21]  Change1(i_sensor__train_in_r(V1!1), const(ct))(T1!1)
[-22]  i_sensor__train_in_r(V1!1)(T1!1)
[-23]  T1!1 - ct ≥ dist_r_to_i / max_speed - response_time
  |-------
[1]    position(T1!1) = lowered
```

In this sequent, down has fired at Duration(lower) + lower_time + t2!1 + up_dur in antecedent -2, which is lower_time after the end of lower, which occurs at Duration(lower) + t2!1 + up_dur in antecedent -13. To complete this subgoal, it must be shown that the variables do not change between the end of down and T1!1, so that position is lowered at that time in consequent 1.

The final step in the proof sketch is to show that nothing fires from the end of down until now. This is shown using 45 prover commands. The key to this step is the use of the no_trans_fire_vnc_lt lemma to show that the variables do not change after the end of down. The bulk of the proof consists of achieving a contradiction between the entry assertion of each transition and the state of the process when the transition is to fire to show that no transition can fire in this interval. Thus, 241 commands are necessary to complete the PVS proof of the up case. This proof is shown in appendix D.

*10.8.2.6. PVS Safety Property Proof*

As an example of using the proof sketch of a safety property to perform the corresponding PVS proof, consider the proof sketch of the Sensor property of the railroad crossing discussed in section 9.2.4.2.2. The first step in the proof sketch is to show that exit_I fired at now - exit_dur. This is accomplished using 8 prover commands. Besides commands for skolemizing, expanding, and simplifying various clauses, the main command of this step is the change-fire command. The next step in the proof sketch is to assume that there is a time in the interval [now - (dist_R_to_I + dist_I_to_out) / max_speed + response_time, now) such that train_in_R is false and thus enter_R must have fired. This step takes 12 prover commands to perform. The main portions of this step consist of using the not_vnc_vc lemma to produce a change to train_in_R and then using change-fire to show that enter_R is the transition that fired. After change-fire is applied, the main goal of the PVS proof looks like the following.

```
{-1}   Exit(enter_r, t1!2)
{-2}   Entry(enter_r, t1!2 - Duration(enter_r))
[-3]   train_in_r(t1!2 - Duration(enter_r)) ≠ train_in_r(t1!2)
[-4]   t1!2 - Duration(enter_r) ≥ 0
[-5]   Fired(enter_r, t1!2 - Duration(enter_r))
[-6]   t1!2 ≥ 0
[-7]   V1!1 < t1!2
[-8]   t1!2 ≤ T1!1 - exit_dur
[-9]   t1!3 < t1!2
[-10]  FORALL (t3: time):
           t1!3 ≤ t3 AND t3 < t1!2 IMPLIES NOT train_in_r(t3) = train_in_r(t1!2)
[-11]  FORALL (t2: time):
           t1!2 ≤ t2 AND t2 ≤ t1!2 IMPLIES train_in_r(t2) = train_in_r(t1!2)
[-12]  train_in_r(t1!2)
[-13]  ts ≥ 0
[-14]  ts ≤ (T1!1 - exit_dur)
[-15]  Start1(enter_r, const(ts))(T1!1 - exit_dur)
[-16]  train_in_r(T1!1 - exit_dur)
[-17]  T1!1 - exit_dur - ts ≥ (dist_i_to_out + dist_r_to_i) / min_speed - exit_dur
[-18]  T1!1 - exit_dur ≥ 0)
[-19]  Fired(exit_i, T1!1 - exit_dur)
[-20]  T0 < T1!1
[-21]  T1!1 < DELTA + T0
[-22]  T1!1 - (dist_i_to_out + dist_r_to_i) / max_speed + response_time ≤ V1!1
[-23]  V1!1 < T1!1
  |-------
[1]    V1!1 ≥ T1!1 - exit_dur
[2]    train_in_r(T1!1)
[3]    train_in_r(V1!1)
```

In this sequent, V1!1 is the time at which train_in_R was false in the interval as shown in consequent 3. T1!1 is the time at which the property must hold, thus antecedent -19 shows that exit_I fired at now - exit_dur. The fact that enter_R ended at some time in the appropriate interval is shown by the antecedents -5, -7, and -8.

The remainder of the steps in the proof sketch consist of showing that a contradiction can be achieved between the fact the enter_R fired in the interval and the entry assertion of exit_I. In the sequent above, this essentially consists of achieving a contradiction between antecedents -5 and -15 by using various inequalities, the most important of which are in antecedents -17 and -22. This step takes 94 prover commands. These commands consist of expanding the entry assertion of exit_I, retrieving the fact that there was a time at which enter_R started with its appropriate limits, splitting the proof into cases based on the times that various events occurred, and manipulating the resulting inequalities to achieve a contradiction. Thus, 114 prover commands are necessary to complete the PVS proof. This proof is shown in appendix D.

## 10.9. Theorem Proving Results

Table 10.9 shows the results of using PVS to prove the proof obligations of the testbed systems. As can be seen, approximately half of the total number of proof obligations were completely discharged using the prover. All of the obligations of the cruise control system and the Olympic boxing scoring system specifications were completely discharged with the exception of the global schedule of the scoring system. This schedule, however, is not provable due to a flaw in the scoring system itself and not in the specification. Namely, it is possible for a boxer to obtain more total points and yet still lose the fight.

| System | Total Obligations | Attempted Obligations | Completed Obligations | Prover Commands |
|---|---|---|---|---|
| Bakery Algorithm | 21 | 18 | 17 | 466 |
| Cruise Control | 9 | 9 | 9 | 535 |
| Elevator | 33 | 13 | 12 | 234 |
| Olympic Boxing | 18 | 17 | 17 | 1073 |
| Phone | 51 | 25 | 16 | 172 |
| Production Cell | 69 | 37 | 31 | 903 |
| Railroad Crossing | 14 | 10 | 8 | 1367 |
| Stoplight | 24 | 18 | 0 | 29 |
| Total | 239 | 147 | 110 | 4779 |

Table 10.9: Results of theorem proving on testbed systems

In table 10.9, the number of proof obligations attempted indicates how many proofs were started, but not completed. The meaning of this number varies from system to system. In some cases, such as the railroad crossing, a significant portion of the proofs that were not completed were performed. For example, in the proof of the Gate liveness property of section 10.8.2.5, one of the two worst cases was proved, which demonstrated how all of the others could be proved. In the case of the obligations of the stoplight control system, however, only a small number of approaches were tried. The number of prover commands gives an estimate of the effort associated with each system. These numbers only include the latest attempt of each obligation and do not include earlier attempts or backtracking, which would make the numbers significantly higher.

The systems besides the cruise control system and the scoring system were not completely proved due to a number of factors. The foremost reason is that as more and more of the obligations were discharged, it became evident that most of the proofs had similar themes and could be proved using the same techniques as earlier proofs. Thus, once enough mechanisms were developed to deal with the most common themes, it became less critical to actually complete every proof. This was the case for the bakery algorithm, the production cell, and the railroad crossing specifications. The other factor is that some of the processes exhibit behavior that is extremely non-trivial to reason about within a theorem prover. This type of behavior is most prevalent in iterative single-threaded processes and multi-threaded processes.

One of the central themes of the proofs of properties in iterative single-threaded processes is finding the maximum number of full iterations as discussed in section 9.2.5.3. The main difficulty arises when it must be proved that this number is actually the worst case and that the other cases are subsumed. In many instances, each case depends on the behavior of the operating environment. Thus, there may be infinitely many other cases that the worst case must be shown to subsume. In addition, each case may be complex to reason about by itself. For example, in the elevator control system, a case consists of a set of times at which the buttons are pushed. It is impossible to reason explicitly about all such cases, especially since the worst case can only be expressed symbolically. In order to deal with this issue, new theorem prover techniques must be developed to support “worst case” reasoning.

One of the central themes of the proofs of properties in multi-threaded processes is finding the maximum number of threads that can be enabled at a given time as discussed in section 9.2.6.2.3. This is equivalent the finding the cardinality of the set of threads that are enabled at a given time. In a hand proof, such a cardinality can be found fairly quickly based mostly on human ingenuity and

"hand waving". In PVS, however, cardinality proofs are extremely complex and become even more so when the set predicate is non-trivial. For the set of enabled threads, the set predicate (i.e. is a specific thread enabled at the given time) is highly non-trivial as it depends on the arbitrary first-order logic expressions of the transition entry assertion associated with the thread as well as the execution history of the process and the behavior of the operating environment. Thus, determining the cardinality of this set within PVS becomes intractable. Further research is necessary to make such a proof feasible.

Although additional theorem prover techniques are needed for iterative single-threaded and multi-threaded processes, these process types compromise only a small fraction of the testbed systems. Only 4 of 25 processes are iterative single-threaded and only 2 of 25 are multi-threaded. Of these six processes, only two of them, the Elevator process and the Central_Control process, suffer from the problems mentioned above. Given that the testbed systems are a random sample taken from existing literature, it is likely that simple single-threaded processes compromise the significant majority of real-world systems as well. This is also a reasonable assumption because every iterative single-threaded or multi-threaded process is inevitably surrounded by a number of simple single-threaded processes such as buttons, sensors, and other input/output processes that support it. This means that the techniques described in this chapter will be directly applicable to most real-time systems.

# Chapter 11

# Conclusion

## 11.1. Summary

In this research, the ASTRAL real-time specification language was augmented to meet the five requirements presented in chapter one, which had not previously been met by a single real-time specification language. ASTRAL has met these requirements as shown below.

- ASTRAL was formally defined within the PVS theorem prover and rigorously reasoned about during the analysis of a set of testbed systems.
- ASTRAL is based on transition systems and first-order logic, which already allowed simple systems to be described in a simple and intuitive manner and did not require any augmentation.
- ASTRAL is very expressive and has facilities for designing large and complex systems such as a modular proof system as well as composition and refinement capabilities, which were revised and expanded to enhance correctness, expressiveness, and usability.
- ASTRAL was furnished with an integrated software development environment that provides support for design, analysis, and reuse.
- ASTRAL was furnished with a systematic analysis methodology in which each analysis tool of the development environment was provided with guidelines for effective usage and for complementing the results obtained by other tools.

Chapters five, six, and seven present the tools and techniques developed for designing real-time systems in ASTRAL. Chapter five describes the design portions of the ASTRAL software development environment, which include capabilities for editing, formatting, validating, and composing ASTRAL specifications. This chapter also presents a specification manager, which guides the user during design and analysis. Chapter six discusses the problems in the original ASTRAL semantics and proof obligations and presents the revised and expanded versions. Chapter

seven discusses the problems in the original ASTRAL sequential refinement mechanism and proposes a new parallel mechanism that increases the expressiveness of the language.

Chapters eight, nine, and ten describe the tools and techniques developed for analyzing real-time specifications in ASTRAL. Chapter eight presents the set of classification schemes that were used as the basis for the systematic analysis methodology. It also describes querying mechanisms that can be used to obtain the classification information as well as other information required during analysis. Chapter nine presents a methodology for systematically determining model checker test cases to guarantee that a property will be adequately tested. In addition, this chapter presents a methodology for systematically proving the requirements of a system by hand based on process and property classifications. Chapter ten discusses how the proof of a property can be carried out within a mechanical theorem prover in a similar manner as the proof by hand.

## 11.2. Conclusions

The above qualities of ASTRAL demonstrate that it is possible to develop a specification language that makes the design and analysis of real-time systems more practical and increases the likelihood that the language will be used. While providing ASTRAL with these qualities, this research has also provided a general framework through which other specification languages can be augmented to meet the requirements of chapter one.

1. Choosing the appropriate specification language

This research was based on the ASTRAL real-time specification language. ASTRAL was well-suited for this research given its intuitive style and expressiveness as well as its modularity, refinement, and composition facilities. In general, not every specification language is suitable for augmentation. It is necessary to begin with a language with qualities similar to ASTRAL.

2. Solidifying the foundation of the language

When research began, the ASTRAL language had a number of errors and omissions in its definition. By encoding ASTRAL into the language of a theorem prover, it forced every aspect of the language to be examined in minute detail and allowed the definition to be rigorously reasoned about during formal proofs within the prover. In general, this process has the same benefits for any specification language.

3. Building an analysis hierarchy

The ASTRAL analysis hierarchy consists of a model checking stage, a proof sketch stage, and a theorem proving stage. In this hierarchy, analysis moves from cheap, fully automated testing to expensive, partially automated verification. In general, a similar hierarchy can be used for any specification language. By using tools that are more heavily automated and that require less expertise earlier in the analysis process, a significant amount of time is saved in the later stages where errors are costly to fix.

4. Identifying where guidance is needed in each stage of the hierarchy

In the ASTRAL hierarchy, guidance was needed in the form of test cases in the model checking stage, proof ordering and how to perform each proof in the proof sketch stage, and proof ordering, a general plan of attack, and how the plan can be carried out in the theorem proving stage. In general, the guidance that is needed in a stage depends on the characteristics of the tool associated with that stage. For example, the ASTRAL model checker requires the user to input test cases consisting of concrete values for each constant in the system. The model checkers for other languages may not require test cases, but may require other input such as choosing an appropriate abstraction to limit the state space.

5. Identifying classification schemes on which guidance can be based

For ASTRAL, classifications based on transitions, processes, and properties were identified, which affected the style of proof. Although these classifications worked well for ASTRAL, they will not necessarily work well for other specification languages. In general, it is necessary to specify and verify a number of systems to identify the classification schemes of most benefit. In identifying these schemes, it is necessary to make sure that each is statically recognizable. That is, even though some classification schemes may be beneficial, they are of no use if they cannot be recognized before performing dynamic analysis.

6. Providing the appropriate guidance for each stage based on the classification schemes

In ASTRAL, guidance was provided based on two principles. First, properties that were simpler in nature were separated from other properties so that they could be more fully automated than the whole. For example, untimed properties were separated from timed properties, which allowed the use of frame axioms to automate their proofs. Second, complex reasoning was decomposed into more fundamental reasoning units that could be attacked in an identical manner. For example, in timed proofs the process was decomposed into forward and backward transition steps that could repeatedly

be taken to reach the given requirement. In general, these same techniques can also be used in other languages with suitable modifications. For example, instead of using transition steps, a language would use steps based on its own event types.

7. Providing tool support for guidance

In ASTRAL, the guidance of each stage is supported by a number of different tools. For example, the classifiers are used to determine the classifications of the current specification. The formula splitter is used to separate simpler properties from the others. The sequence generator is used to find the possible forward and backward steps. The specification manager is used to direct the flow of design and analysis. In general, the same types of assistance are useful for other languages, again with suitable modifications for language constructs.

## 11.3. Future Work

As discussed in section 10.9, there is a need for additional theorem prover techniques for iterative single-threaded and multi-threaded processes. For iterative single-threaded processes, it is necessary to support "worst case" reasoning and to allow the other cases to be implicitly subsumed. For multi-threaded processes, it is necessary to support reasoning about the cardinality of complex sets. To provide the necessary support for these processes types, a more in-depth study of the capabilities of the theorem prover is needed.

The analysis tools and techniques of this dissertation focus on the intra-level proof obligations, which are the most fundamental type of obligation in ASTRAL. Most of the techniques developed for intra-level proofs are directly applicable to the inter-level and composition proofs. There may be additional techniques, however, that are unique to these other obligation types. These techniques can be developed in a similar manner as the techniques developed for intra-level obligations. That is, a set of refinements and compositions can be specified and then their respective proof obligations can be performed and analyzed to determine the techniques that are most effective for each. These techniques should be integrated into an analysis methodology and be provided with tool support such as decision procedures, querying mechanisms, and a theorem prover encoding, as was done for the intra-level obligations.

An additional area of research is to investigate new analysis stages to be added into the analysis hierarchy. Currently, the hierarchy consists of a model checking stage, a proof sketch stage, and a theorem proving stage. There are several stages that could be performed between the model checking and proof sketch stages. For example, a mutation analysis stage could be added to test the adequacy

of the test cases generated for the model checker by using mutation techniques similar to those developed for programming languages. A symbolic model checking stage could be added to test the specification by using symbolic constants instead of using test cases composed of concrete values. A symbolic executor stage could be added to allow the user to explicitly test the scenarios that are most likely to contain errors. There are also stages that could be performed between proof sketch construction and theorem proving. For example, instead of describing proof sketches in an *ad hoc* fashion, a proof language could be developed that allows the most common types of reasoning in hand proofs to be described in a regular manner. This would allow portions of proof sketches to be automatically translated into a sequence of theorem prover commands. With each of these additional stages also comes the need for developing the appropriate guidance and tool support.

# Bibliography


[Aal 93] van der Aalst, W.M.P. "Interval timed coloured Petri nets and their analysis". *Proceedings of the 14th International Conference on Applications and Theory of Petri Nets*, Chicago, IL, USA, 21-25 June 1993. Edited by: Marsan, M.A. Berlin, Germany: Springer-Verlag, 1993. p. 453-72.

[ACD 90] Alur, R.; Courcoubetis, C.; Dill, D. "Model-checking for real-time systems". *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, Philadelphia, PA, USA, 4-7 June 1990. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1990. p. 414-25.

[AD 90] Alur, R.; Dill, D. "Automata for modeling real-time systems". *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, Coventry, UK, 16-20 July 1990. Edited by: Paterson, M.S. Berlin, Germany: Springer-Verlag, 1990. p. 322-35.

[AG 93] Atlee, J.M.; Gannon, J. "State-based model checking of event-driven system requirements". *IEEE Transactions on Software Engineering*, Jan. 1993, vol. 19, (no. 1): 24-40.

[AGM 97] Alborghetti, A.; Gargantini, A.; Morzenti, A. "Providing automated support to deductive analysis of time critical systems". *Proceedings of the 6th European Software Engineering Conference*, Zurich, Switzerland, 22-25 Sept. 1997. Software Engineering Notes, Nov. 1997, vol. 22, (no. 6): 211-26.

[AH 94] Alur, R.; Henzinger, T.A. "A really temporal logic". *Journal of the Association for Computing Machinery*, Jan. 1994, vol. 41, (no. 1): 181-204.

[AH 96] Archer, M.; Heitmeyer, C. "Mechanical verification of timed automata: a case study". *Proceedings of the 2nd Real-Time Technology and Applications Symposium*, Brookline, MA, USA, 10-12 June 1996. Edited by: Jeffay, K.; Zhao, W. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1996. p. 192-203.

[AH 97] Archer, M.; Heitmeyer, C. "Human-style theorem proving using PVS". *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*, Murray Hill, NJ, USA, 19-22 Aug. 1997. Edited by: Gunter, E.L.; Felty, A. Berlin, Germany: Springer-Verlag, 1997. p. 33-48.

[AK 85] Auernheimer, B.; Kemmerer, R.A. ASLAN user's manual. Technical Report TRCS84-10, Department of Computer Science, University of California, Santa Barbara, March 1985.

[AK 86a] Auernheimer, B.; Kemmerer, R.A. "Procedural and nonprocedural semantics of the ASLAN formal specification language". *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, 1986.

[AK 86b] Auernheimer, B.; Kemmerer, R.A. "RT ASLAN: a specification language for real-time systems". *IEEE Transactions on Software Engineering*, Sept. 1986, vol. SE-12, (no. 9): 879-89.

[AL 91] Abadi, M.; Lamport, L. "An old-fashioned recipe for real time". *Proceedings of the REX Workshop on Real-Time Theory in Practice*, Mook, Netherlands, 3-7 June 1991. Edited by: de Bakker, J.W.; Huizing, C.; de Roever, W.P.; Rozenberg, G. Berlin, Germany: Springer-Verlag, 1992. p. 1-27.

[AL 93] Abadi, M.; Lamport, L. Conjoining specifications. Research Report 118, Digital Equipment Corporation, Systems Research Center, Dec. 1993.

[And 91] Andrews, G.R. "Paradigms for process interaction in distributed programs". C*omputing Surveys*, March 1991, vol. 23, (no. 1): 49-90.

[BD 91] Berthomieu, B.; Diaz, M. "Modeling and verification of time dependent systems using time Petri nets". *IEEE Transactions on Software Engineering*, March 1991, vol. 17, (no. 3): 259-73.

[BMR 95] Borgida, A.; Mylopoulos, J.; Reiter, R. "On the frame problem in procedure specifications". *IEEE Transactions on Software Engineering*, Oct. 1995, vol. 21, (no. 10): 785-98.

[BS 91] Boussinot, F.; de Simone, R. "The ESTEREL language". *Proceedings of the IEEE*, Sept. 1991, vol. 79, (no. 9): 1293-304.

[Bun 96] Bun, L. "Checking properties of ASTRAL specifications with PVS". *Proceedings of the 2nd Annual Conference of the Advanced School for Computing and Imaging*, Lommel, Belgium, June 1996, p. 102-7.

[Bun 97] Bun, L. "Embedding Astral in PVS". *Proceedings of the 3rd Annual Conference of the Advanced School for Computing and Imaging*, Heijen, The Netherlands, June 1997, p. 130-6.

[CES 86] Clarke, E.M.; Emerson, E.A.; Sistla, A.P. "Automatic verification of finite-state concurrent systems using temporal logic specifications". *ACM Transactions on Programming Languages and Systems*, April 1986, vol. 8, (no. 2): 244-63.

[CGK 97] Coen-Porisini, A.; Ghezzi, C.; Kemmerer, R.A. "Specification of realtime systems using ASTRAL". *IEEE Transactions on Software Engineering*, Sept. 1997, vol. 23, (no. 9): 572-98.

[CK 93] Coen-Porisini, A.; Kemmerer, R.A. "The composability of ASTRAL realtime specifications". *Proceedings of the 7th International Symposium on Software Testing and Analysis*, Cambridge, MA, USA, 28-30 June 1993. SIGSOFT Software Engineering Notes, July 1993, vol. 18, (no. 3): 128-38.

[CKM 94] Coen-Porisini, A.; Kemmerer, R.A.; Mandrioli, D. "A formal framework for ASTRAL intralevel proof obligations". *IEEE Transactions on Software Engineering*, Aug. 1994, vol. 20, (no. 8): 548-61.

[CKM 95] Coen-Porisini, A.; Kemmerer, R.A.; Mandrioli, D. "A formal framework for ASTRAL inter-level proof obligations". *Proceedings of the 5th European Software Engineering Conference*, Sitges, Spain, 25-28 Sept. 1995. Berlin, Germany: Springer-Verlag, 1995. p. 90-108.

[COR 95] Crow, J.; Owre, S.; Rushby, J.; Shankar, N.; Srivas, M. "A tutorial introduction to PVS". *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, Apr. 1995.

[CSK 94] Coen-Porisini, A.; San Pietro, P.; Kemmerer, R.A. "Formal semantics definition for ASTRAL". Unpublished technical report, Department of Computer Science, University of California, Santa Barbara, 1994.

[DC 95] Dean, T.R.; Cordy, J.R. "A syntactic theory of software architecture". *IEEE Transactions on Software Engineering*, April 1995, vol. 21, (no. 4): 302-13.

[DiM 99] Di Marzo Serugendo, G. Stepwise refinement of formal specifications based on logical formulae: from CO-OPN/2 specifications to java programs. Ph.D. Thesis no. 1931, EPFL, Lausanne, 1999.

[DJR 91] Davies, J.; Jackson, D.M.; Reed, G.M.; Reed, J.N.; and others. "Timed CSP: theory and practice". *Proceedings of the REX Workshop on Real-Time Theory in Practice*, Mook, Netherlands, 3-7 June 1991. Edited by: de Bakker, J.W.; Huizing, C.; de Roever, W.P.; Rozenberg, G. Berlin, Germany: Springer-Verlag, 1992. p. 640-75.

[DK 97] Dang, Z.; Kemmerer, R.A. "Using the ASTRAL model checker for cryptographic protocol analysis", *Proceedings of the DIMACS Workshop on the Design and Formal Verification of Security Protocols*, Rutgers University, 1997.

[FF 84] Filman, R.E.; Friedman, D.P. Coordinated computing: tools and techniques for distributed software. New York: McGraw-Hill, 1984.

[FG 89] Franklin, M.K.; Gabrielian, A. "A transformational method for verifying safety properties in real-time systems". *Proceedings of the 10th Real Time Systems Symposium*, Santa Monica, CA, USA, 5-7 Dec. 1989. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1989. p. 112-23.

[FGM 98] Felder, M.; Gargantini, A.; Morzenti, A. "A theory of implementation and refinement in timed Petri nets". *Theoretical Computer Science*, July 1998, vol. 202, (no. 1-2): 127-61.

[FMM 94] Felder, M.; Mandrioli, D.; Morzenti, A. "Proving properties of real-time systems through logical specifications and Petri net models". *IEEE Transactions on Software Engineering*, Feb. 1994, vol. 20, (no. 2): 127-41.

[GF 88] Gabrielian, A.; Franklin, M.K. "State-based specification of complex real-time systems". *Proceedings of the 9th Real-Time Systems Symposium*, Huntsville, AL, USA, 6-8 Dec. 1988. Washington, DC, USA: IEEE Comput. Soc. Press, 1988. p. 2-11.

[GF 90] Gabrielian, A.; Franklin, M.K. "Multi-level specification and verification of real-time software". *Proceedings of the 12th International Conference on Software Engineering*, Nice, France, 26-30 March 1990. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1990. p. 52-62.

[GK 91a] Ghezzi, C.; Kemmerer, R.A. "ASTRAL: an assertion language for specifying realtime systems". *Proceedings of the 3rd European Software Engineering Conference*, Milan, Italy, 21-24 Oct. 1991. Edited by: van Lamsweerde, A.; Fugetta, A. Berlin, Germany: Springer-Verlag, 1991. p. 122-40.

[GK 91b] Ghezzi, C.; Kemmerer, R.A. "Executing formal specifications: the ASTRAL to TRIO translation approach". *Proceedings of the 6th International Symposium on Software Testing and Analysis*, Victoria, B.C., Canada, Oct. 1991.

[GL 90] Gerber, R.; Lee, I. "A proof system for communicating shared resources". *Proceedings of the 11th Real-Time Systems Symposium*, Lake Buena Vista, FL, USA, 5-7 Dec. 1990. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1990. p. 288-99.

[GL 92] Gerber, R.; Lee, I. "A layered approach to automating the verification of real-time systems". *IEEE Transactions on Software Engineering*, Sept. 1992, vol. 18, (no. 9): 768-84.

[GM 93] Gordon, M.J.C.; Melham, T.F.: editors. Introduction to HOL: a theorem proving environment for higher order logic. New York: Cambridge University Press, 1993.

[GMM 90] Ghezzi, C.; Mandrioli, D.; Morzenti, A. "TRIO: a logic language for executable specifications of real-time systems". *Journal of Systems and Software*, May 1990, vol. 12, (no. 2): 107-23.

[Gor 95] Gordon, M. "Notes on PVS from a HOL perspective". Available at http://www.cl.cam.ac.uk/users/mjcg/PVS.html, Aug. 1995.

[Har 87] Harel, D. "Statecharts: a visual formalism for complex system". *Science of Computer Programming*, June 1987, vol. 8, (no. 3): 231-74.

[HCH 93] Hale, R.; Cardell-Oliver, R.; Herbert, J. "An embedding of timed transition systems in HOL". *Formal Methods in System Design*, Aug. 1993, vol.3, (no.1-2):151-74.

[HCR 91] Halbwachs, N.; Caspi, P.; Raymond, P.; Pilaud, D. "The synchronous data flow programming language LUSTRE". *Proceedings of the IEEE*, Sept. 1991, vol. 79, (no. 9): 1305-20.

[HL 94] Heitmeyer, C.; Lynch, N. "The generalized railroad crossing: a case study in formal verification of real-time systems". *Proceedings of the 15th Real-Time Systems Symposium*, San Juan, Puerto Rico, 7-9 Dec. 1994. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1994. p. 120-31.

[HLN 90] Harel, D.; Lachover, H.; Naamad, A.; Pnueli, A.; and others. "STATEMATE: a working environment for the development of complex reactive systems". *IEEE Transactions on Software Engineering*, April 1990, vol. 16, (no. 4): 403-14.

[HM 85] Hennessy, A.; Milner, R. "Algebraic laws for nondeterminism and concurrency". *Journal of the ACM*, Jan. 1985, vol. 32, (no. 1): 137-61.

[HM 96] Heitmeyer, C.; Mandrioli, D.: editors. Formal methods for real-time computing. New York: John Wiley, 1996.

[HMP 94] Henzinger, T.A.; Manna, Z.; Pnueli, A. "Temporal proof methodologies for timed transition systems". *Information and Computation*, 1 Aug. 1994, vol. 112, (no. 2): 273-337.

[Hoa 69] Hoare, C.A.R. "An axiomatic basis for computer programming". *Communications of the ACM*, Oct. 1969, vol. 12, (no. 10): 576-80, 583.

[Hoa 85] Hoare, C.A.R. Communicating sequential processes. London, UK: Prentice-Hall, 1985.

[Hoo 94] Hooman, J. "Extending Hoare logic to real-time". *Formal Aspects of Computing*, 1994, vol. 6, (no. 6A): 801-25.

[HU 79] Hopcroft, J.E.; Ullman, J.D. Introduction to automata theory, languages, and computation. Reading, MA: Addison-Wesley, 1979.

[i-L 91a] The languages of STATEMATE. i-Logix Inc., Burlington, MA, Technical Report, Jan 1991.

[i-L 91b] The semantics of statecharts. i-Logix Inc., Burlington, MA, Technical Report, 1991.

[JM 86] Jahanian, F.; Mok, A.K. "Safety analysis of timing properties in real-time systems". *IEEE Transactions on Software Engineering*, Sept. 1986, vol. SE-12, (no. 9): 890-904.

[JM 94] Jahanian, F.; Mok, A.K. "Modechart: a specification language for real-time systems". *IEEE Transactions on Software Engineering*, Dec. 1994, vol. 20, (no. 12): 933-47.

[JS 88] Jahanian, F.; Stuart, D.A. "A method for verifying properties of Modechart specifications". *Proceedings of the 9th Real-Time Systems Symposium*, Huntsville, AL, USA, 6-8 Dec. 1988. Washington, DC, USA: IEEE Comput. Soc. Press, 1988. p. 12-21.

[KDK 99] Kolano, P.Z.; Dang, Z.; Kemmerer, R.A. "The design and analysis of real-time systems using the ASTRAL software development environment". *Annals of Software Engineering*, vol. 7, 1999.

[KM 96] Kaufmann, M.; Strother Moore, J. "ACL2: an industrial strength version of Nqthm". *Proceedings of the 11th Annual Conference on Computer Assurance*, Gaithersburg, MD, USA, 17-21 June 1996. New York, NY, USA: IEEE, 1996. p. 23-34.

[Lam 74] Lamport, L. "A new solution of Dijkstra's concurrent programming problem". *Communications of the ACM*, Aug. 1974, vol. 17, (no. 8): 453-5.

[Lam 91] Lamport, L. "The temporal logic of actions". Research Report 79, Digital Equipment Corporation, Systems Research Center, Dec. 1991.

[Lei 69] Leisenring, A.C. Mathematical logic and Hilbert's ε-symbol. New York, Gordon and Breach, 1969.

[Lib 97] Libes, D. "Automation and testing of character-graphic programs". *Software - Practice and Experience*, Feb. 1997, vol. 27, (no. 2): 123-37.

[LL 95] Lewerentz, C.; Lindner, T.: editors. Formal development of reactive systems: case study production cell. New York: Springer-Verlag, 1995.

[LV 91] Lynch, N.; Vaandrager, F. "Forward and backward simulations for timing-based systems". *Proceedings of the REX Workshop on Real-Time Theory in Practice*, Mook, Netherlands, 3-7 June 1991. Edited by: de Bakker, J.W.; Huizing, C.; de Roever, W.P.; Rozenberg, G. Berlin, Germany: Springer-Verlag, 1992. p. 397-446.

[MF 76] Merlin, P.M.; Farber, D.J. "Recoverability of communication protocols-implications of a theoretical study". *IEEE Transactions on Communications*, Sept. 1976, vol. COM-24, (no. 9): 1036-43.

[Mil 80] Milner, R. A calculus of communicating systems. New York: Springer-Verlag, 1980.

[MP 91] Mall, R.; Patnaik, L.M. "Formal timing analysis of distributed systems". *International Journal of Parallel Programming*, April 1991, vol. 20, (no. 2): 75-94.

[MP 92] Manna, Z.; Pnueli, A. The temporal logic of reactive and concurrent systems. New York: Springer-Verlag, 1992.

[MS 96] Mok, A.K.; Stuart, D. "Simulation vs. verification: getting the best of both worlds". *Proceedings of the 11th Annual Conference on Computer Assurance*, Gaithersburg, MD, USA, 17-21 June 1996. New York, NY, USA: IEEE, 1996. p. 12-22.

[Mus 93] The ASTRAL specification language for real-time systems: user's manual. Technical Report, CEFRIEL, 1993.

[Oly 96] Official 1996 Olympic Web Site, <http://www.atlanta.olympic.org>.

[OSR 98a] Owre, S.; Shankar, N.; Rushby, J.M.; Stringer-Calvert, D.W.J. PVS language reference. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1998.

[OSR 98b] Owre, S.; Shankar, N.; Rushby, J.M.; Stringer-Calvert, D.W.J. PVS system guide. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1998.

[Ost 89] Ostroff, J.S. Temporal logic for real-time systems. Taunton, UK: Research Studies Press, 1989.

[Ost 92] Ostroff, J.S. "Formal methods for the specification and design of real-time safety critical systems". *Journal of Systems and Software*, April 1992, vol. 18, (no. 1): 33-60.

[Ost 99] Ostroff, J.S. "Composition and refinement of discrete real-time systems". *ACM Transactions on Software Engineering and Methodology*, Jan. 1999, vol. 8, (no. 1): 1-48.

[OW 90] Ostroff, J.S.; Wonham, W.M. "A framework for real-time discrete event control". *IEEE Transactions on Automatic Control*, April 1990, vol. 35, (no. 4): 386-97.

[Pet 62] Petri, C.A. Kommunikationen mit automaten. Ph.D. thesis, University of Bonn, Bonn, Germany, 1962.

[PW 84] Pinter, S.S.; Wolper, P. "A temporal logic for reasoning about partially ordered computations". *Proceedings of the 3rd ACM Principles of Distributed Computing*, Vancouver, B.C. (1984), p. 28-37.

[RMM 93] Ramakrishna, Y.S.; Melliar-Smith, P.M.; Moser, L.E.; Dillon, L.K.; and others. "Really visual temporal reasoning". *Proceedings of the 13th Real-Time Systems Symposium*, Raleigh Durham, NC, USA, 1-3 Dec. 1993. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1993. p. 262-73.

[RMM 96] Ramakrishna, Y.S.; Melliar-Smith, P.M.; Moser, L.E.; Dillon, L.K.; and others. "Interval logics and their decision procedures. Part II: a real-time interval logic". *Theoretical Computer Science*, 15 Dec. 1996, vol. 170, (no. 1-2): 1-46.

[SG 96] Shaw, M.; Garlan, D. Software architecture: perspectives on an emerging discipline. Upper Saddle River, NJ: Prentice Hall, 1996.

[Sha 89] Shaw, A.C. "Reasoning about time in higher-level language software". *IEEE Transactions on Software Engineering*, July 1989, vol. 15, (no. 7): 875-89.

[Sin 93] Singh, A.K. "Program refinement in fair transition systems". *Acta Informatica*, 1993, vol. 30, (no. 6): 503-35.

[SMV 83] Schwartz, R.L.; Melliar-Smith, P.M.; Vogt, F. "An interval logic for higher-level temporal reasoning". *Proceedings of the 2nd ACM Principles of Distributed Computing*, 1983, p. 173-186.

[SOR 98] Shankar, N.; Owre, S.; Rushby, J.M.; Stringer-Calvert, D.W.J. PVS prover guide. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1998.

[Spi 90] Spivey, J.M. "Specifying a real-time kernel". *IEEE Software*, Sept. 1990, vol. 7, (no. 5): 21-8.

[SS 94] Skakkebaek, J.U.; Shankar, N. "Towards a duration calculus proof assistant in PVS". *Proceedings of the 3rd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lubeck, Germany, 19-23 Sept. 1994. Edited by: Langmaack, H.; de Roever, W.-P.; Vytopil, J. Berlin, Germany: Springer-Verlag, 1994. p. 660-79.

[Stu 90] Stuart, D.A. "Implementing a verifier for real-time systems". *Proceedings of the 11th Real-Time Systems Symposium*, Lake Buena Vista, FL, USA, 5-7 Dec. 1990. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1990. p. 62-71.

[Tan 92] Tanenbaum, A.S. Modern operating systems. Englewood Cliffs, NJ: Prentice Hall, 1992.

[WM 85] Ward, P.T.; Mellor, S.J. Structured development for real-time systems. New York: Yourdon Press, 1985.

[You 96] Young, W.D. "Comparing verification systems: interactive consistency in ACL2". *Proceedings of 11th Annual Conference on Computer Assurance*, Gaithersburg, MD, USA, 17-21 June 1996. New York, NY, USA: IEEE, 1996. p. 35-45.

[YMS 95] Jin Yang; Mok, A.K.; Stuart, D. "A new generation modechart verifier". *Proceedings of the 2nd Real-Time Technology and Applications Symposium*, Chicago, IL, USA, 15-17 May 1995. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1995. p. 116-25.

[Zav 82] Zave, P. "An operational approach to requirements specification for embedded systems". *IEEE Transactions on Software Engineering*, May 1982, vol. SE-8, (no. 3): 250-69.

[ZHR 91] Zhou Chaochen; Hoare, C.A.R.; Ravn, A.P. "A calculus of durations". *Information Processing Letters*, 13 Dec. 1991, vol. 40, (no. 5): 269-76.

# Appendix A

# ASTRAL Specifications of Testbed Systems

## A.1. Bakery Algorithm

```
SPECIFICATION Bakery_Algorithm
  GLOBAL SPECIFICATION Bakery_Algorithm
    PROCESSES
              procs: array [ n_procs ]  of Proc
    TYPE
              procs_int: TYPEDEF i: integer ( 1 <= i
                                              & i <= n_procs ) ,
              nonneg_int: TYPEDEF i: integer ( i >= 0 ) ,
              pos_int: TYPEDEF i: integer ( i > 0 ) ,
              nonneg_real: TYPEDEF r: real ( r >= 0 ) ,
              pos_real: TYPEDEF r: real ( r > 0 )
    CONSTANT
              n_procs: pos_int,
              exec_time: pos_real
    SCHEDULE
      /* only a single proc can be in its critical region at any given time */
               ( FORALL i, j: procs_int
                         ( procs [ i ] .in_critical
                         & procs [ j ] .in_critical
                      ->   i = j )  )
  END Bakery_Algorithm
  PROCESS SPECIFICATION Proc
    LEVEL Top_Level
      IMPORT
                pos_int, nonneg_int, nonneg_real, n_procs, exec_time, procs_int, procs, procs.choosing,
                procs.number, procs.in_critical
      EXPORT
                choosing, number, in_critical
      VARIABLE
                next_i: pos_int,
                choosing: boolean,
                number: nonneg_int,
                in_critical: boolean,
                delay: nonneg_real
      INITIAL
                next_i = 1
              & ~choosing
              & number = 0
              & ~in_critical
      INVARIANT
        /* must have finished loop to be in critical */
                 ( in_critical
              ->   next_i > n_procs )
        /* when in loop, have already chosen a non-zero number */
              &  ( next_i > 1
              ->   ~choosing
                 & number ~= 0 )
        /* proc must have chosen a non-zero number to enter its critical region */
              &  ( in_critical
              ->   ~choosing
                 & number ~= 0 )
        /* when number is changed to a non-zero value, the new value must be greater than or equal to all
           other numbers at the time that number "started changing" */
              &  ( Change ( number, now )
                 & number ~= 0
              ->   FORALL i: procs_int
                           ( number >= past ( procs [ i ] .number + 1, now - exec_time )  )  )
        /* number can only change to zero when the proc has been in its critical region */
              &  ( Change ( number, now )
                 & number = 0
```

```
        ->   ~in_critical
           & EXISTS t: time
                      ( Change [ 2 ]  ( number )  < t
                      & t < now
                      & past ( Change ( in_critical, t ) , t )
                      & past ( in_critical, t )  )  )
SCHEDULE
  /* when a proc is in its critical region, its number and id must be the lowest of all procs with
     non-zero numbers */
          ( in_critical
        ->   FORALL i, j: procs_int
                      ( procs [ j ]  = self
                    ->   procs [ i ] .number = 0
                      |  number < procs [ i ] .number
                      |  number = procs [ i ] .number
                      & j < i )  )
  /* inductive loop invariant used to prove above */
        &  ( Start ( for_loop, now )
        ->   FORALL i, j: procs_int
                      ( procs [ j ]  = self
                      & i < next_i
                    ->   procs [ i ] .number = 0
                      |  number < procs [ i ] .number
                      |  number = procs [ i ] .number
                      & j <= i )  )
IMPORTED VARIABLE
  /* proc must have chosen a non-zero number to enter its critical region */
          ( FORALL i: procs_int
                      ( procs [ i ] .in_critical
                    ->   ~procs [ i ] .choosing
                      & procs [ i ] .number ~= 0 )  )
  /* when number is changed to a non-zero value, the new value must be greater than or equal to all
     other numbers at the time that number "started changing" */
        &  ( FORALL i, j: procs_int
                      ( Change ( procs [ i ] .number, now )
                      & procs [ i ] .number ~= 0
                    ->   procs [ i ] .number >= past ( procs [ j ] .number + 1, now - exec_time )  )  )
  /* number can only change to zero when the proc has been in its critical region */
        &  ( FORALL i: procs_int
                      ( Change ( procs [ i ] .number, now )
                      & procs [ i ] .number = 0
                    ->   ~procs [ i ] .in_critical
                      & EXISTS t: time
                                  ( Change [ 2 ]  ( procs [ i ] .number )  < t
                                  & t < now
                                  & past ( Change ( procs [ i ] .in_critical, t ) , t )
                                  & past ( procs [ i ] .in_critical, t )  )  )  )
TRANSITION set_choose
  ENTRY           [ TIME : exec_time ]
            now >= delay
          & ~choosing
          & number = 0
  EXIT
            choosing
TRANSITION set_number
  ENTRY           [ TIME : exec_time ]
            choosing
          & FORALL t: time
                      ( Change ( number, t )
                    ->   t < Change ( choosing )  )
  EXIT
            FORALL i: procs_int
                      ( number >= procs [ i ] .number + 1 )
          & EXISTS i: procs_int
                      ( number = procs [ i ] .number + 1 )
TRANSITION reset_choose
  ENTRY           [ TIME : exec_time ]
            choosing
          & FORALL t: time
                      ( Change ( number, t )
                    ->   t > Change ( choosing )  )
  EXIT
            ~choosing
TRANSITION for_loop
  ENTRY           [ TIME : exec_time ]
            next_i <= n_procs
          & ~choosing
          & number ~= 0
          & ~procs [ next_i ] .choosing
          &  ( procs [ next_i ] .number = 0
            |  number < procs [ next_i ] .number
            |  number = procs [ next_i ] .number
             & FORALL j: procs_int
                         ( procs [ j ]  = self
                       ->   j <= next_i )  )
```

```
        EXIT
                  next_i = next_i' + 1
      TRANSITION start_critical
        ENTRY           [ TIME : exec_time ]
                  next_i > n_procs
                & ~in_critical
        EXIT
                  in_critical
      TRANSITION end_critical
        ENTRY           [ TIME : exec_time ]
                  in_critical
        EXIT
                  ~in_critical
                & next_i = 1
                & number = 0
                & delay >= now
    END Top_Level
  END Proc
END Bakery_Algorithm
```

## A.2. Cruise Control

```
SPECIFICATION Cruise_Control
  GLOBAL SPECIFICATION Cruise_Control
    PROCESSES
              the_speed_control: Speed_Control,
              the_accelerometer: Accelerometer,
              the_speedometer: Speedometer,
              the_tire_sensor: Tire_Sensor
    TYPE
              nonneg_real: TYPEDEF r: real ( r >= 0 ) ,
              pos_real: TYPEDEF r: real ( r > 0 ) ,
              nonneg_int: TYPEDEF i: integer ( i >= 0 )
    CONSTANT
              tire_circumference: pos_real,
              sample_time: pos_real
  END Cruise_Control
  PROCESS SPECIFICATION Speed_Control
    LEVEL Top_Level
      IMPORT
                nonneg_real, pos_real, the_speedometer, the_speedometer.speed, the_accelerometer,
                the_accelerometer.acceleration
      EXPORT
                set_gas_pedal, set_brake_pedal, enable_cruise, disable_cruise, maintain_speed, resume_speed,
                begin_speed_increase, end_speed_increase
      CONSTANT
                input_dur, control_dur: pos_real,
                desired_acceleration: pos_real,
                full_throttle: pos_real,
                throttle_step: pos_real,
                speed_step: pos_real,
                increase_delay: pos_real
      VARIABLE
                throttle: nonneg_real,
                brake: nonneg_real,
                desired_speed: nonneg_real,
                cruise_on: boolean,
                maintaining_speed: boolean,
                increasing_speed: boolean,
                cruise_throttle: nonneg_real,
                foot_throttle: nonneg_real
      INITIAL
                ~cruise_on
              & ~maintaining_speed
              & ~increasing_speed
              & throttle = 0
              & brake = 0
              & foot_throttle = 0
      INVARIANT
        /* cruise control must be on to maintain speed */
                ( maintaining_speed
              ->   cruise_on )
        /* must be maintaining speed to increase desired speed */
              &  ( increasing_speed
              ->   maintaining_speed )
        /* when maintaining speed, throttle will be the higher of the pedal and the cruise throttle */
              &  ( maintaining_speed
                 & foot_throttle < cruise_throttle
              ->   throttle = cruise_throttle )
              &  ( maintaining_speed
                 & foot_throttle > cruise_throttle
              ->   throttle = foot_throttle )
```

```
  /* when not maintaining speed, throttle is equal to foot throttle */
        &  ( ~maintaining_speed
        ->   throttle = foot_throttle )
SCHEDULE
  /* cruise control will stop maintaining speed as quickly as possible when the brake is applied */
           ( control_dur <= input_dur
           & now >= input_dur + input_dur
           & past ( maintaining_speed, now - input_dur - input_dur )
           & Call ( set_brake_pedal, now - input_dur - input_dur )
        ->   EXISTS t: time
                     ( now - input_dur - input_dur <= t
                     & t <= now
                     & ~past ( maintaining_speed, t )  )  )
        &  ( input_dur <= control_dur
           & now >= input_dur + control_dur
           & past ( maintaining_speed, now - input_dur - control_dur )
           & Call ( set_brake_pedal, now - input_dur - control_dur )
        ->   EXISTS t: time
                     ( now - input_dur - control_dur <= t
                     & t <= now
                     & ~past ( maintaining_speed, t )  )  )
FURTHER ASSUMPTIONS #1
  FURTHER PROCESS ASSUMPTIONS
    TRANSITION SELECTION
              enabled_transitions CONTAINS  { set_brake_pedal }
           & TRUE
         ->   eligible_transitions =  { set_brake_pedal }
TRANSITION set_gas_pedal ( v: pos_real )
  ENTRY           [ TIME : input_dur ]
            TRUE
  EXIT
            foot_throttle = v
          & IF
                    maintaining_speed'
                  & cruise_throttle' > foot_throttle
            THEN
                    throttle = cruise_throttle'
            ELSE
                    throttle = foot_throttle
            FI
TRANSITION set_brake_pedal ( v: pos_real )
  ENTRY           [ TIME : input_dur ]
            TRUE
  EXIT
            ~maintaining_speed
          & ~increasing_speed
          & throttle = foot_throttle'
          & brake = v
TRANSITION enable_cruise
  ENTRY           [ TIME : input_dur ]
            ~cruise_on
  EXIT
            cruise_on
TRANSITION disable_cruise
  ENTRY           [ TIME : input_dur ]
            cruise_on
  EXIT
            ~cruise_on
          & ~maintaining_speed
          & ~increasing_speed
          & throttle = foot_throttle'
TRANSITION maintain_speed
  ENTRY           [ TIME : input_dur ]
            cruise_on
          & ~maintaining_speed
  EXIT
            cruise_throttle = throttle'
          & desired_speed = the_speedometer.speed
          & maintaining_speed
TRANSITION resume_speed
  ENTRY           [ TIME : input_dur ]
            cruise_on
          & ~maintaining_speed
  EXIT
            maintaining_speed
          & IF
                    cruise_throttle' > foot_throttle'
            THEN
                    throttle = cruise_throttle'
            ELSE
                    throttle = foot_throttle'
            FI
TRANSITION begin_speed_increase
  ENTRY           [ TIME : input_dur ]
            maintaining_speed
          & ~increasing_speed
```

```
      EXIT
                increasing_speed
              & desired_speed = desired_speed' + speed_step
    TRANSITION end_speed_increase
      ENTRY           [ TIME : input_dur ]
                increasing_speed
      EXIT
                ~increasing_speed
    TRANSITION increase_speed
      ENTRY           [ TIME : control_dur ]
                increasing_speed
              & now - Change ( desired_speed )  >= increase_delay
      EXIT
                desired_speed = desired_speed' + speed_step
    TRANSITION increase_throttle
      ENTRY           [ TIME : control_dur ]
                maintaining_speed
              &  ( desired_speed > the_speedometer.speed
                 & the_accelerometer.acceleration < desired_acceleration
                |  desired_speed < the_speedometer.speed
                 & the_accelerometer.acceleration <  - desired_acceleration )
      EXIT
                IF
                        cruise_throttle' + throttle_step > full_throttle
                THEN
                        cruise_throttle = full_throttle
                ELSE
                        cruise_throttle = cruise_throttle' + throttle_step
                FI
              & IF
                        cruise_throttle > foot_throttle'
                THEN
                        throttle = cruise_throttle
                ELSE
                        throttle = foot_throttle'
                FI
    TRANSITION decrease_throttle
      ENTRY           [ TIME : control_dur ]
                maintaining_speed
              &  ( desired_speed > the_speedometer.speed
                 & the_accelerometer.acceleration > desired_acceleration
                |  desired_speed < the_speedometer.speed
                 & the_accelerometer.acceleration >  - desired_acceleration )
      EXIT
                IF
                        cruise_throttle' - throttle_step < 0
                THEN
                        cruise_throttle = 0
                ELSE
                        cruise_throttle = cruise_throttle' - throttle_step
                FI
              & IF
                        cruise_throttle > foot_throttle'
                THEN
                        throttle = cruise_throttle
                ELSE
                        throttle = foot_throttle'
                FI
  END Top_Level
END Speed_Control
PROCESS SPECIFICATION Accelerometer
  LEVEL Top_Level
    IMPORT
              sample_time, the_speedometer, the_speedometer.speed
    EXPORT
              acceleration
    VARIABLE
              acceleration: real
    INITIAL
              acceleration = 0
    TRANSITION sample_acceleration
      ENTRY           [ TIME : sample_time ]
                TRUE
      EXIT
                IF
                        now < 2 * sample_time
                THEN
                        acceleration = the_speedometer.speed / sample_time
                ELSE
                        acceleration =  ( the_speedometer.speed - Past ( the_speedometer.speed, now - 2 *
                                                                         sample_time )  )  / sample_time
                FI
  END Top_Level
END Accelerometer
```

```
  PROCESS SPECIFICATION Speedometer
    LEVEL Top_Level
      IMPORT
                nonneg_real, sample_time, tire_circumference, the_tire_sensor, the_tire_sensor.rotations
      EXPORT
                speed
      VARIABLE
                speed: nonneg_real
      INITIAL
                speed = 0
      TRANSITION sample_speed
        ENTRY           [ TIME : sample_time ]
                  TRUE
        EXIT
                  IF
                          now < 2 * sample_time
                  THEN
                          speed = the_tire_sensor.rotations * tire_circumference / sample_time
                  ELSE
                          speed =  ( the_tire_sensor.rotations - Past ( the_tire_sensor.rotations, now -
                                  2 * sample_time )  )  * tire_circumference / sample_time
                  FI
    END Top_Level
  END Speedometer
  PROCESS SPECIFICATION Tire_Sensor
    LEVEL Top_Level
      IMPORT
                nonneg_int, pos_real
      EXPORT
                rotate, rotations
      CONSTANT
                sense_time: pos_real
      VARIABLE
                rotations: nonneg_int
      INITIAL
                rotations = 0
      TRANSITION rotate
        ENTRY           [ TIME : sense_time ]
                  TRUE
        EXIT
                  rotations = rotations' + 1
    END Top_Level
  END Tire_Sensor
END Cruise_Control
```

## A.3. Elevator Control System

```
SPECIFICATION Elevator_System
  GLOBAL SPECIFICATION Elevator_System
    PROCESSES
              the_elevator: Elevator,
              the_elevator_buttons: Elevator_Button_Panel,
              the_floor_buttons: array [ 1..n_floors ]  of Floor_Button_Panel
    TYPE
              pos_integer: TYPEDEF i: integer ( i > 0 ) ,
              pos_real: TYPEDEF r: real ( r > 0 ) ,
              floor: TYPEDEF i: pos_integer ( i <= n_floors )
    CONSTANT
              n_floors: pos_integer,
              request_dur, clear_dur: pos_real,
              t_service_request, t_move, t_stop, t_move_door: pos_real
    AXIOM
      /* clear_request must be able to fire no matter how many requests are made
         while the elevator door is opening */
               ( clear_dur + n_floors * request_dur < t_move_door )
      /* must be at least 2 floors in the building */
            &  ( n_floors >= 2 )
    SCHEDULE
      /* any request must be serviced within time t_service_request */
              FORALL f: floor
                      ( the_elevator_buttons.Call ( request_floor ( f ) , now - t_service_request )
                    ->   EXISTS t: time
                                ( now - t_service_request < t
                                & t <= now
                                & past ( the_elevator.position, t )  = f
                                & past ( Change ( the_elevator.door_open, t ) , t )
                                & past ( the_elevator.door_open, t )  )  )
            & FORALL f: floor
                      ( f ~= n_floors
                      & the_floor_buttons [ f ] .Call ( request_up, now - t_service_request )
                    ->   EXISTS t: time
                                ( now - t_service_request < t
                                & t <= now
```

```
                                & past ( the_elevator.position, t )  = f
                                & past ( Change ( the_elevator.door_open, t ) , t )
                                & past ( the_elevator.door_open, t )
                                & past ( the_elevator.going_up, t )  )  )
          & FORALL f: floor
                     ( f ~= 1
                     & the_floor_buttons [ f ] .Call ( request_down, now - t_service_request )
                   ->   EXISTS t: time
                                ( now - t_service_request < t
                                & t <= now
                                & past ( the_elevator.position, t )  = f
                                & past ( Change ( the_elevator.door_open, t ) , t )
                                & past ( the_elevator.door_open, t )
                                & ~past ( the_elevator.going_up, t )  )  )
END Elevator_System
PROCESS SPECIFICATION Elevator
  LEVEL Top_Level
    IMPORT
              pos_real, floor, request_dur, the_elevator_buttons, the_floor_buttons,
              the_elevator_buttons.floor_requested, the_elevator_buttons.request_floor,
              the_floor_buttons.up_requested, the_floor_buttons.down_requested,
              the_floor_buttons.request_up, the_floor_buttons.request_down, t_stop, t_move, t_move_door,
              t_service_request, n_floors
    EXPORT
              position, going_up, door_open, moving, door_moving
    CONSTANT
              move_dur, arrive_dur, open_dur, close_dur, door_stop_dur: pos_real
    VARIABLE
              position: floor,
              going_up, door_open, moving, door_moving: boolean
    AXIOM
      /* t_service_request must be big enough to handle the worst case. One instance of the worst case
         is when the elevator is moving up from floor 1 to 2 and 2 has not been requested on the elevator
         panel nor has any request been made on 2's button panel. Let t_arrive be the next time such that
         End(arrive, t_arrive). up_request and down_request are simultaneously called on floor 2 an "instant"
         after t_arrive - 2 * request_dur and down_request fires first. In addition, every floor in the
         building (besides 2) has up_requested (except the top floor) and down_requested (except the bottom
         floor). Thus, the up request is not posted in time for the elevator to service it and the elevator
         must stop and open the door at every floor up to the top, back down to the bottom, and back up
         to 2. */
               ( t_service_request >= 2 * request_dur + move_dur + t_move + arrive_dur +  ( 2 * n_floors -
                                                                                            3 )  *
                  ( open_dur + t_move_door + door_stop_dur + t_stop + close_dur + t_move_door + door_stop_dur +
                     request_dur + move_dur + t_move + arrive_dur )  + open_dur + t_move_door + door_stop_dur )
    DEFINE
              request_above ( f0: floor ) : boolean ==
                      EXISTS f: floor
                               ( f > f0
                               &  ( the_elevator_buttons.floor_requested ( f )
                                 |  the_floor_buttons [ f ] .up_requested
                                 |  the_floor_buttons [ f ] .down_requested )  ) ,
              request_below ( f0: floor ) : boolean ==
                      EXISTS f: floor
                               ( f < f0
                               &  ( the_elevator_buttons.floor_requested ( f )
                                 |  the_floor_buttons [ f ] .up_requested
                                 |  the_floor_buttons [ f ] .down_requested )  )
    INITIAL
              position = 1
            & going_up
            & ~door_open
            & ~moving
            & ~door_moving
    INVARIANT
      /* the elevator door must stay closed while the elevator is moving */
               ( moving
              ->   ~door_open
                 & ~door_moving )
    CONSTRAINT
      /* if the elevator changes direction, there cannot be an outstanding request in the old direction */
               ( going_up
               & ~going_up'
              ->   ~request_below' ( position' )  )
            &  ( ~going_up
               & going_up'
              ->   ~request_above' ( position' )  )
    SCHEDULE
      /* if the elevator is moving in some direction, there must be an outstanding request in that direction */
               ( moving
               & going_up
              ->   request_above ( position )  )
            &  ( moving
               & ~going_up
              ->   request_below ( position )  )
```

```
  /* any request must be serviced within time t_service_request */
        &  ( FORALL f: floor
                     ( the_elevator_buttons.Call ( request_floor ( f ) , now - t_service_request )
                  ->   EXISTS t: time
                               ( now - t_service_request < t
                               & t <= now
                               & past ( position, t )  = f
                               & past ( Change ( door_open, t ) , t )
                               & past ( door_open, t )  )  )  )
        &  ( FORALL f: floor
                     ( f ~= n_floors
                     & the_floor_buttons [ f ] .Call ( request_up, now - t_service_request )
                  ->   EXISTS t: time
                               ( now - t_service_request < t
                               & t <= now
                               & past ( position, t )  = f
                               & past ( Change ( door_open, t ) , t )
                               & past ( door_open, t )
                               & past ( going_up, t )  )  )
           & FORALL f: floor
                     ( f ~= 1
                     & the_floor_buttons [ f ] .Call ( request_down, now - t_service_request )
                  ->   EXISTS t: time
                               ( now - t_service_request < t
                               & t <= now
                               & past ( position, t )  = f
                               & past ( Change ( door_open, t ) , t )
                               & past ( door_open, t )
                               & ~past ( going_up, t )  )  )  )
IMPORTED VARIABLE
  /* buttons only clear after elevator has arrived and started opening the doors */
           ( FORALL f: floor
                     ( Change ( the_elevator_buttons.floor_requested ( f ) , now )
                     & ~the_elevator_buttons.floor_requested ( f )
                  ->   EXISTS t: time
                               ( Change [ 2 ]  ( the_elevator_buttons.floor_requested ( f )  )  <
                                   t
                               & t <= now
                               & past ( position, t )  = f
                               & ~past ( door_open, t )
                               & past ( door_moving, t )  )  )  )
        &  ( FORALL f: floor
                     ( f ~= n_floors
                     & Change ( the_floor_buttons [ f ] .up_requested, now )
                     & ~the_floor_buttons [ f ] .up_requested
                  ->   EXISTS t: time
                               ( Change [ 2 ]  ( the_floor_buttons [ f ] .up_requested )  < t
                               & t <= now
                               & past ( position, t )  = f
                               & ~past ( door_open, t )
                               & past ( door_moving, t )
                               & past ( going_up, t )  )  )  )
        &  ( FORALL f: floor
                     ( f ~= 1
                     & Change ( the_floor_buttons [ f ] .down_requested, now )
                     & ~the_floor_buttons [ f ] .down_requested
                  ->   EXISTS t: time
                               ( Change [ 2 ]  ( the_floor_buttons [ f ] .down_requested )  < t
                               & t <= now
                               & past ( position, t )  = f
                               & ~past ( door_open, t )
                               & past ( door_moving, t )
                               & ~past ( going_up, t )  )  )  )
  /* the top floor never has an up request and the bottom floor never has a down request */
        &  ( ~the_floor_buttons [ n_floors ] .up_requested )
        &  ( ~the_floor_buttons [ 1 ] .down_requested )
  /* requests cannot be made of the elevator to stop at a floor between when the door starts opening
     on that floor until when it starts closing */
        &  ( Change ( door_moving, now )
           & door_moving
           & door_open
        ->   FORALL t: time
                     ( t >= Change [ 2 ]  ( door_moving )
                  ->   ~the_elevator_buttons.Call ( request_floor ( position ) , t )  )  )
  /* requests cannot be made of the elevator to stop at a floor between when the door starts opening
     on that floor until when it starts closing */
        &  ( Change ( door_moving, now )
           & door_moving
           & door_open
        ->   FORALL t: time
                     ( t >= Change [ 2 ]  ( door_moving )
                  ->    ( past ( going_up, t )
                        ->   ~the_floor_buttons [ position ] .Call ( request_up, t )  )
                        &  ( past ( ~going_up, t )
                        ->   ~the_floor_buttons [ position ] .Call ( request_down, t )  )  )  )
```

```
TRANSITION move_up
  ENTRY           [ TIME : move_dur ]
            ~door_open
          & ~door_moving
          & request_above ( position )
          &  ( going_up
            |  ~going_up
             & ~request_below ( position )
             & ~the_floor_buttons [ position ] .up_requested )
          &  ( End ( arrive, now )
             & ~the_elevator_buttons.floor_requested ( position )
             & ~the_floor_buttons [ position ] .up_requested
            |  FORALL t, t1: time
                        ( Change ( moving, t )
                        & Change ( door_open, t1 )
                    ->   t < t1
                        & now >= t1 + request_dur )  )
  EXIT
            moving
          & going_up
TRANSITION move_down
  ENTRY           [ TIME : move_dur ]
            ~door_open
          & ~door_moving
          & request_below ( position )
          &  ( ~going_up
            |  going_up
             & ~request_above ( position )
             & ~the_floor_buttons [ position ] .down_requested )
          &  ( End ( arrive, now )
             & ~the_elevator_buttons.floor_requested ( position )
             & ~the_floor_buttons [ position ] .down_requested
            |  FORALL t, t1: time
                        ( Change ( moving, t )
                        & Change ( door_open, t1 )
                    ->   t < t1
                        & now >= t1 + request_dur )  )
  EXIT
            moving
          & ~going_up
TRANSITION arrive
  ENTRY           [ TIME : arrive_dur ]
            moving
          & FORALL t: time
                     ( t <= now
                     &  ( End ( move_down, t )
                       |  End ( move_up, t )  )
                   ->   now - t_move >= t )
          & FORALL t, t1: time
                     ( t <= now
                     & End ( arrive, t )
                     &  ( End ( move_up, t1 )
                       |  End ( move_down, t1 )  )
                   ->   t < t1 )
  EXIT
            IF
                    going_up'
            THEN
                    position = position' + 1
            ELSE
                    position = position' - 1
            FI
TRANSITION open_door
  ENTRY           [ TIME : open_dur ]
            ~door_open
          & ~door_moving
          &  ( ~moving
            |  moving
             & EXISTS t: time
                        ( Change ( position, t )
                        & t > Change ( moving )  )  )
          &  ( the_elevator_buttons.floor_requested ( position )
            |  going_up
             &  ( the_floor_buttons [ position ] .up_requested
               |  ~request_above ( position )
                & the_floor_buttons [ position ] .down_requested )
            |  ~going_up
             &  ( the_floor_buttons [ position ] .down_requested
               |  ~request_below ( position )
                & the_floor_buttons [ position ] .up_requested )  )
  EXIT
            ~moving
          & door_moving
          & going_up =  ( going_up'
                        &  ( request_above' ( position' )
                          |  the_floor_buttons [ position' ] .up_requested' )
```

```
                          |  ~request_below' ( position' )
                           & ~the_floor_buttons [ position' ] .down_requested' )
    TRANSITION close_door
      ENTRY           [ TIME : close_dur ]
                door_open
              & ~door_moving
              & now - t_stop >= Change ( door_open )
      EXIT
                door_moving
    TRANSITION door_stop
      ENTRY           [ TIME : door_stop_dur ]
                door_moving
              & now - t_move_door >= Change ( door_moving )
      EXIT
                ~door_moving
              & door_open = ~door_open'
  END Top_Level
END Elevator
PROCESS SPECIFICATION Elevator_Button_Panel
  LEVEL Top_Level
    IMPORT
              floor, request_dur, clear_dur, the_elevator, the_elevator.position, the_elevator.door_open,
              the_elevator.door_moving
    EXPORT
              floor_requested, request_floor
    VARIABLE
              floor_requested ( floor ) : boolean
    ENVIRONMENT
      /* multiple button pushes should have no effect */
               ( FORALL f: floor
                         ( Change ( floor_requested ( f ) , now )
                         & ~floor_requested ( f )
                      ->   FORALL t: time
                                   ( Start ( request_floor ( f )  )  <= t
                                   & t <= now
                                ->   ~Call ( request_floor ( f ) , t )  )  )  )
      /* requests cannot be made of the elevator to stop at a floor between when the door starts opening
         on that floor until when it starts closing */
            &  ( Change ( the_elevator.door_moving, now )
               & the_elevator.door_moving
               & the_elevator.door_open
            ->   FORALL t: time
                         ( t >= Change [ 2 ]  ( the_elevator.door_moving )
                      ->   ~Call ( request_floor ( the_elevator.position ) , t )  )  )
    INITIAL
              FORALL f: floor
                        ( ~floor_requested ( f )  )
    INVARIANT
      /* buttons only clear after elevator has arrived and started opening the doors */
               ( FORALL f: floor
                         ( Change ( floor_requested ( f ) , now )
                         & ~floor_requested ( f )
                      ->   EXISTS t: time
                                   ( Change [ 2 ]  ( floor_requested ( f )  )  < t
                                   & t <= now
                                   & past ( the_elevator.position, t )  = f
                                   & ~past ( the_elevator.door_open, t )
                                   & past ( the_elevator.door_moving, t )  )  )  )
    TRANSITION request_floor ( f: floor )
      ENTRY           [ TIME : request_dur ]
                ~floor_requested ( f )
      EXIT
                floor_requested ( f )
    TRANSITION clear_floor_request
      ENTRY           [ TIME : clear_dur ]
                floor_requested ( the_elevator.position )
              & ~the_elevator.door_open
              & the_elevator.door_moving
      EXIT
                ~floor_requested ( the_elevator.position )
  END Top_Level
END Elevator_Button_Panel
PROCESS SPECIFICATION Floor_Button_Panel
  LEVEL Top_Level
    IMPORT
              request_dur, clear_dur, the_floor_buttons, the_elevator, the_elevator.position,
              the_elevator.door_open, the_elevator.going_up, the_elevator.door_moving, n_floors
    EXPORT
              up_requested, down_requested, request_up, request_down
    VARIABLE
              up_requested, down_requested: boolean
    ENVIRONMENT
      /* multiple button pushes should have no effect */
               ( Change ( up_requested, now )
               & ~up_requested
```

```
        ->   FORALL t: time
                     ( Start ( request_up )  <= t
                     & t <= now
                  ->   ~Call ( request_up, t )  )  )
        &  ( Change ( down_requested, now )
           & ~down_requested
        ->   FORALL t: time
                     ( Start ( request_down )  <= t
                     & t <= now
                  ->   ~Call ( request_down, t )  )  )
  /* requests cannot be made of the elevator to stop at a floor between when the door starts opening
     on that floor until when it starts closing */
        &  ( Change ( the_elevator.door_moving, now )
           & the_elevator.door_moving
           & the_elevator.door_open
           & the_floor_buttons [ the_elevator.position ]  = Self
        ->   FORALL t: time
                     ( t >= Change [ 2 ]  ( the_elevator.door_moving )
                  ->    ( past ( the_elevator.going_up, t )
                        ->   ~Call ( request_up, t )  )
                        &  ( past ( ~the_elevator.going_up, t )
                        ->   ~Call ( request_down, t )  )  )  )
INITIAL
          ~up_requested
        & ~down_requested
INVARIANT
  /* buttons only clear after elevator has arrived and started opening the doors */
          ( Change ( up_requested, now )
          & ~up_requested
        ->   EXISTS t: time
                     ( Change [ 2 ]  ( up_requested )  < t
                     & t <= now
                     & the_floor_buttons [ past ( the_elevator.position, t )  ]  = Self
                     & ~past ( the_elevator.door_open, t )
                     & past ( the_elevator.door_moving, t )
                     & past ( the_elevator.going_up, t )  )  )
        &  ( Change ( down_requested, now )
           & ~down_requested
        ->   EXISTS t: time
                     ( Change [ 2 ]  ( down_requested )  < t
                     & t <= now
                     & the_floor_buttons [ past ( the_elevator.position, t )  ]  = Self
                     & ~past ( the_elevator.door_open, t )
                     & past ( the_elevator.door_moving, t )
                     & ~past ( the_elevator.going_up, t )  )  )
  /* the top floor never has an up request and the bottom floor never has a down request */
        &  ( the_floor_buttons [ n_floors ]  = Self
        ->   ~up_requested )
        &  ( the_floor_buttons [ 1 ]  = Self
        ->   ~down_requested )
SCHEDULE
  /* calls will be posted within 2 * request_dur time */
          ( Call ( request_up, now - 2 * request_dur )
        ->   EXISTS t: time
                     ( now - 2 * request_dur < t
                     & t <= now
                     & past ( Change ( up_requested, t ) , t )
                     & past ( up_requested, t )  )  )
        &  ( Call ( request_down, now - 2 * request_dur )
        ->   EXISTS t: time
                     ( now - 2 * request_dur < t
                     & t <= now
                     & past ( Change ( down_requested, t ) , t )
                     & past ( down_requested, t )  )  )
TRANSITION request_up
  ENTRY           [ TIME : request_dur ]
            ~up_requested
          & the_floor_buttons [ n_floors ]  ~= Self
  EXIT
            up_requested
TRANSITION request_down
  ENTRY           [ TIME : request_dur ]
            ~down_requested
          & the_floor_buttons [ 1 ]  ~= Self
  EXIT
            down_requested
TRANSITION clear_up_request
  ENTRY           [ TIME : clear_dur ]
            up_requested
          & the_floor_buttons [ the_elevator.position ]  = Self
          & the_elevator.going_up
          & ~the_elevator.door_open
          & the_elevator.door_moving
  EXIT
            ~up_requested
```

```
      TRANSITION clear_down_request
        ENTRY           [ TIME : clear_dur ]
                  down_requested
                & the_floor_buttons [ the_elevator.position ]  = Self
                & ~the_elevator.going_up
                & ~the_elevator.door_open
                & the_elevator.door_moving
        EXIT
                  ~down_requested
    END Top_Level
  END Floor_Button_Panel
END Elevator_System
```

## A.4. Olympic Boxing Scoring System

```
SPECIFICATION Olympic_Boxing
  GLOBAL SPECIFICATION Olympic_Boxing
    PROCESSES
              Time_Keeper: Timer,
              Judges: array [ 1..5 ]  of Judge,
              Scorer: Tabulate
    TYPE
              Pos_Integer: TYPEDEF i: Integer ( i > 0 ) ,
              Non_Negative: TYPEDEF i: Integer ( i >= 0 ) ,
              Pos_Real: TYPEDEF r: Real ( r > 0 ) ,
              Name:  ( Fighter1, Fighter2, None ) ,
              Boxer: TYPEDEF n: Name ( n ~= None ) ,
              Judge_ID: TYPEDEF i: Pos_Integer ( i <= 5 ) ,
              Set_Of_Judge_ID: SET OF Judge_ID,
              Decision:  ( In_Progress, Win, Draw )
    CONSTANT
              Num_Rounds: Pos_Integer, /* 3 */
              Window: Pos_Real /* 1 second */
    ENVIRONMENT
              FORALL t: Time, j: Judge_ID, B: Boxer
                       ( t <= Now - Window
                       & past ( Judges [ j ] .Call ( Score ( B ) , t ) , t )
                       & FORALL t1: Time, i: Judge_ID
                                 ( t1 >= t - Window
                                 & t1 < t
                                ->   ~past ( Judges [ i ] .Call ( Score, t1 ) , t1 )  )
                             ->   EXISTS S: Set_Of_Judge_ID
                                 ( SET_SIZE  ( S )  >= 3
                                 & FORALL i: Judge_ID
                                            ( i ISIN S
                                          <->    EXISTS t1: Time
                                                        ( t1 >= t
                                                        & t1 < t + Window
                                                        & past ( Judges [ i ] .Call ( Score ( B ) , t1 ) ,
                                                                  t1 )  )  )  )
                       & FORALL t1: Time, i: Judge_ID
                                 ( t1 >= t + Window
                                 & t1 < t + 2 * Window
                                ->   ~past ( Judges [ i ] .Call ( Score, t1 ) , t1 )  )  )
    SCHEDULE
               ( Scorer.Outcome ~= In_Progress
            ->    ( Time_Keeper.Round_Number = Num_Rounds
                  & ~Time_Keeper.In_Round )  )
            &  ( Scorer.Outcome = Win
            ->   EXISTS S: Set_Of_Judge_ID
                        ( SET_SIZE  ( S )  >= 3
                        & FORALL j: Judge_ID
                                   ( j ISIN S
                                ->   FORALL B: Boxer
                                            ( Judges [ j ] .Score_Card ( Scorer.Winner )  >=
                                                Judges [ j ] .Score_Card ( B )  )  )  )  )
  END Olympic_Boxing
  PROCESS SPECIFICATION Timer
    LEVEL Top_Level
      IMPORT
                Pos_Real, Non_Negative, Num_Rounds
      EXPORT
                In_Round, Round_Number
      CONSTANT
                Begin_Dur, End_Dur: Pos_Real,
                Round_Length: Pos_Real, /* 3 minutes */
                Between_Rounds: Pos_Real /* 1 minute */
      VARIABLE
                Round_Number: Non_Negative,
                In_Round: Boolean
      INITIAL
                Round_Number = 0
              & ~In_Round
```

```
    INVARIANT
              FORALL t: Time
                       ( t <= Now
                       & past ( Round_Number, t )  = Num_Rounds
                       & ~past ( In_Round, t )
                    ->   FORALL t1: Time
                                ( t1 > t
                                & t1 <= Now
                             ->   past ( Round_Number, t1 )  = past ( Round_Number, t )  )
                       & FORALL t1: Time
                                ( t1 > t
                                & t1 <= Now
                             ->   past ( In_Round, t1 )  = past ( In_Round, t )  )  )
    TRANSITION Begin_Round
      ENTRY           [ TIME : Begin_Dur ]
                 ( Round_Number = 0
                 |  Now - Start ( End_Round )  >= Between_Rounds )
              & Round_Number < Num_Rounds
              & ~In_Round
      EXIT
                Round_Number = Round_Number' + 1
              & In_Round
    TRANSITION End_Round
      ENTRY           [ TIME : End_Dur ]
                Now - Start ( Begin_Round )  >= Round_Length
              & In_Round
      EXIT
                ~In_Round
  END Top_Level
END Timer
PROCESS SPECIFICATION Tabulate
  LEVEL Top_Level
    IMPORT
              Pos_Real, Name, Boxer, Judges, Judge_ID, Set_Of_Judge_ID, Non_Negative, Num_Rounds, Decision,
              Window, Time_Keeper.Round_Number, Judges.Score, Time_Keeper, Time_Keeper.In_Round
    EXPORT
              Winner, Outcome
    CONSTANT
              Final_Dur: Pos_Real,
              Update_Dur: Pos_Real
    VARIABLE
              Points ( Boxer ) : Non_Negative,
              Outcome: Decision,
              Winner: Name
    INITIAL
              FORALL B: Boxer
                       ( Points ( B )  = 0 )
            & Outcome = In_Progress
            & Winner = None
    INVARIANT
               ( Outcome = Win
            ->   Winner ~= None )
            &  ( Outcome ~= In_Progress
            ->   EXISTS t: Time
                          ( t <= Now
                          & End ( Final_Decision, t )  )  )
    CONSTRAINT
              Winner ~= Winner'
         ->   Outcome ~= In_Progress
    SCHEDULE
              Outcome ~= In_Progress
         ->    ( Time_Keeper.Round_Number = Num_Rounds
               & ~Time_Keeper.In_Round )
    IMPORTED VARIABLE
              FORALL t: Time
                       ( t <= Now
                       & past ( Time_Keeper.Round_Number, t )  = Num_Rounds
                       & ~past ( Time_Keeper.In_Round, t )
                    ->   FORALL t1: Time
                                ( t1 > t
                                & t1 <= Now
                             ->   past ( Time_Keeper.Round_Number, t1 )  = past ( Time_Keeper.Round_Number,
                                                                                  t )  )
                       & FORALL t1: Time
                                ( t1 > t
                                & t1 <= Now
                             ->   past ( Time_Keeper.In_Round, t1 )  = past ( Time_Keeper.In_Round,
                                                                              t )  )  )
    TRANSITION Update ( B: Boxer )
      ENTRY           [ TIME : Update_Dur ]
                 EXISTS S: Set_Of_Judge_ID
                           ( SET_SIZE  ( S )  >= 3
                           & FORALL j: Judge_ID
                                     ( j ISIN S
                                  <->    Now - Judges [ j ] .Start ( Score ( B )  )  <= Window )  )
```

```
              & Now - Start ( Update )  >= Window
              & Outcome = In_Progress
       EXIT
              Points ( B )  BECOMES Points' ( B )  + 1
     TRANSITION Final_Decision
       ENTRY          [ TIME : Final_Dur ]
              Time_Keeper.Round_Number = Num_Rounds
              & ~Time_Keeper.In_Round
              & Points ( Fighter1 )  ~= Points ( Fighter2 )
              & Outcome = In_Progress
       EXIT
              EXISTS B: Boxer
                       ( FORALL B1: Boxer
                                 ( Points' ( B )  > Points' ( B1 )
                                |  B = B1 )
                       & Winner = B
                       & Outcome = Win )
       EXCEPT         [ TIME : Final_Dur ]
              Time_Keeper.Round_Number = Num_Rounds
              & ~Time_Keeper.In_Round
              & Points ( Fighter1 )  = Points ( Fighter2 )
              & Outcome = In_Progress
       EXIT
              Outcome = Draw
   END Top_Level
  END Tabulate
  PROCESS SPECIFICATION Judge
    LEVEL Top_Level
      IMPORT
              Pos_Real, Boxer, Time_Keeper.In_Round, Non_Negative, Window, Time_Keeper
      EXPORT
              Score, Score_Card
      CONSTANT
              Score_Dur: Pos_Real
      VARIABLE
              Score_Card ( Boxer ) : Non_Negative
      AXIOM
              Score_Dur < Window
      ENVIRONMENT
               ( EXISTS t: Time
                         ( t <= Now
                         & Call [ 2 ]  ( Score, t )  )
            ->   Call ( Score )  - Call [ 2 ]  ( Score )  >= 2 * Window )
            &  ( Call ( Score, now )
            ->   Time_Keeper.In_Round )
      INITIAL
              FORALL B: Boxer
                       ( Score_Card ( B )  = 0 )
      SCHEDULE
              Call ( Score, now )
          ->   Start ( Score, now )
      TRANSITION Score ( B: Boxer )
       ENTRY          [ TIME : Score_Dur ]
              Time_Keeper.In_Round
              & FORALL t: Time
                       ( t < Now
                       & past ( Start ( Score, t ) , t )
                     ->   Now - t > Window )
       EXIT
              Score_Card ( B )  BECOMES Score_Card' ( B )  + 1
   END Top_Level
  END Judge
END Olympic_Boxing
```

## A.5. Phone System

```
SPECIFICATION Phone_System
  GLOBAL SPECIFICATION Phone_System
    PROCESSES
            Phones: array [ 1 .. Num_Phone ] of Phone,
            Centrals: array [ 1 .. Num_Area ] of Central_Control
    TYPE
            Positive_Integer IS TYPEDEF p: Integer ( p > 0 ) ,
            Digit IS TYPEDEF d: Integer ( d >= 0
                                        & d <= 9 ) ,
            Digit_List IS LIST OF Digit,
            Connection IS STRUCTURE OF  ( From_Area, From_Number, To_Area, To_Number: Digit_List ) ,
            Phone_ID IS TYPEDEF pid: ID ( IDTYPE  ( pid )  = Phone ) ,
            Central_Control_ID IS TYPEDEF pid: ID ( IDTYPE  ( pid )  = Central_Control ) ,
            Enabled_State IS  ( Idle, Ready_To_Dial, Dialing, Ringing, Waiting, Talk, Calling, Disconnecting,
                                Busy, Alarm ) ,
            Connection_Status IS  ( Available, In_Progress, Disconnect, Connect, Talking )
```

```
  CONSTANT
           In_Area ( Phone_ID ) : Central_Control_ID,
           Max_Cust, Num_Phone, Num_Area: Positive_Integer,
           LD_Timeout: Time
  DEFINE
           Plug ( L1, L2: Connection ) : Boolean ==
                  L1 [ From_Area ]  = L2 [ To_Area ]
                & L1 [ From_Number ]  = L2 [ To_Number ]
                & L1 [ To_Area ]  = L2 [ From_Area ]
                & L1 [ To_Number ]  = L2 [ From_Number ]
  ENVIRONMENT
           FORALL C: Central_Control_ID
                    ( SET_SIZE (  { SETDEF P: Phone_ID ( In_Area ( P )  = C
                                                       & Now - 2 <= P.Call ( Pickup )
                                                       & P.Call ( Pickup )  <= Now )  }  )  <= Max_Cust )
  SCHEDULE
           FORALL P: Phone_ID, t, t1, t2: Time
                    ( t <= t1
                    & t1 < t2
                    & Change [ 2 ]  ( In_Area ( P ) .Phone_State ( P ) , t )
                    & past ( In_Area ( P ) .Phone_State ( P ) , t )  = Idle
                    & P.End ( Pickup, t1 )
                    & P.Offhook
                    & Change ( In_Area ( P ) .Phone_State ( P ) , t2 )
                  ->   past ( In_Area ( P ) .Phone_State ( P ) , t2 )  = Ringing
                    |  past ( In_Area ( P ) .Phone_State ( P ) , t2 )  = Ready_To_Dial
                    & t2 <= t1 + 2 )
END Phone_System
PROCESS SPECIFICATION Phone
  LEVEL Top_Level
    IMPORT
             Digit, Phone_ID, Central_Control_ID, Enabled_State, In_Area, Centrals.Phone_State,
             Centrals.Enabled_Ring_Pulse, Centrals.Enabled_Ringback_Pulse
    EXPORT
             Offhook, Next_Digit, Pickup, Enter_Digit, Hangup
    CONSTANT
             T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11: Time
    VARIABLE
             Offhook, Dialtone, Ring, Ringback, Busytone: Boolean,
             Next_Digit: Digit
    DEFINE
             My_Central: Central_Control_ID ==
                     In_Area ( Self )
    ENVIRONMENT
             FORALL t: Time
                      ( Call ( Pickup, t )
                    ->   ~past ( Offhook, t )  )
           & FORALL t: Time
                      ( Call ( Hangup, t )
                    ->   past ( Offhook, t )  )
           & FORALL t: Time
                      ( Call ( Enter_Digit, t )
                    ->    ( past ( Dialtone, t )
                          |  EXISTS t1: Time, n: Integer, D: Digit
                                    ( 2 <= n
                                    & Call [ n ]  ( Enter_Digit ( D ) , t1 )
                                    & past ( Dialtone, t1 )
                                    &  ( n <= 7
                                       & D ~= 1
                                     |  n <= 11
                                       & D = 1 )
                                    & FORALL t2: Time
                                              ( t1 <= t2
                                              & t2 <= t
                                            ->   past ( Offhook, t2 )  )  )  )  )
           & FORALL t: Time
                      ( Call [ 2 ]  ( Pickup, t )
                    ->   Call ( Pickup )  - Call [ 2 ]  ( Pickup )  >= 1 )
    INITIAL
             ~Offhook
           & ~Dialtone
           & ~Busytone
           & ~Ring
           & ~Ringback
    INVARIANT
              ( Dialtone
           ->   Offhook )
           &  ( Ringback
           ->   Offhook )
           &  ( Busytone
           ->   Offhook )
           &  ( Ring
           ->   ~Offhook )
```

```
        &  ( Ring
        ->   ~Dialtone
           & ~Ringback
           & ~Busytone )
SCHEDULE
           ( Dialtone
        ->   ~Ring
           & ~Ringback
           & ~Busytone )
        &  ( Ringback
        ->   ~Dialtone
           & ~Ring
           & ~Busytone )
        &  ( Busytone
        ->   ~Dialtone
           & ~Ring
           & ~Ringback )
IMPORTED VARIABLE
           ( My_Central.Phone_State ( Self )  = Busy
        ->   EXISTS t1, t2, t3, t4: time
                     ( t1 <= t2
                     & t2 < t3
                     & t3 < t4
                     & t4 <= now
                     & FORALL t: time
                               ( t1 <= t
                               & t < t4
                              ->   past ( My_Central.Phone_State ( Self ) , t )  = Dialing )
                     & FORALL t: time
                               ( t4 <= t
                               & t <= now
                             ->   past ( My_Central.Phone_State ( Self ) , t )  = Busy )
                     & past ( Start ( Enter_Digit, t2 ) , t2 )
                     & past ( End ( Enter_Digit, t3 ) , t3 )
                     & past ( My_Central.Phone_State ( Self ) , t4 )  = Busy )  )
        &  ( My_Central.Phone_State ( Self )  = Waiting
        ->   EXISTS t1, t2, t3: time
                     ( t1 <= t2
                     & t2 < t3
                     & t3 < Change ( My_Central.Phone_State ( Self )  )
                     & Change [ 2 ]  ( My_Central.Phone_State ( Self ) , t1 )
                     & past ( My_Central.Phone_State ( Self ) , t1 )  = Dialing
                     & past ( Start ( Enter_Digit, t2 ) , t2 )
                     & past ( End ( Enter_Digit, t3 ) , t3 )  )  )
        &  ( My_Central.Phone_State ( Self )  = Dialing
        ->   EXISTS t1, t2, t3, t4: time
                     ( t1 <= t2
                     & t2 < t3
                     & t3 < t4
                     & t4 <= now
                     & FORALL t: time
                               ( t1 <= t
                               & t < t4
                              ->   past ( My_Central.Phone_State ( Self ) , t )  = Ready_To_Dial )
                     & FORALL t: time
                               ( t4 <= t
                               & t <= now
                              ->   past ( My_Central.Phone_State ( Self ) , t )  = Dialing )
                     & past ( Start ( Enter_Digit, t2 ) , t2 )
                     & past ( End ( Enter_Digit, t3 ) , t3 )
                     & past ( My_Central.Phone_State ( Self ) , t4 )  = Dialing )  )
        &  ( My_Central.Phone_State ( Self )  = Ready_To_Dial
        ->   EXISTS t1: time
                     ( t1 < Change ( My_Central.Phone_State ( Self )  )
                     & Change [ 2 ]  ( My_Central.Phone_State ( Self ) , t1 )
                     & past ( My_Central.Phone_State ( Self ) , t1 )  = Idle )  )
        &  ( EXISTS t1: time
                     ( past ( My_Central.Phone_State ( Self ) , t1 )  = Idle
                     & FORALL t2: time
                               ( t1 <= t2
                               & t2 <= Now
                              ->   past ( My_Central.Phone_State ( Self ) , t2 )  ~= Waiting )  )
        ->   ~My_Central.Enabled_Ringback_Pulse ( Self )  )
FURTHER ASSUMPTIONS #1
  FURTHER PROCESS ASSUMPTIONS
    TRANSITION SELECTION
              enabled_transitions CONTAINS any_subset  (  { Stop_Ringback, Stop_Busytone }  )
            & TRUE
         ->   eligible_transitions =  { Stop_Ringback, Stop_Busytone }  INTERSECT enabled_transitions
TRANSITION Pickup
  ENTRY          [ TIME : T1 ]
            ~Offhook
```

```
    EXIT
              Offhook
            & ~Dialtone
            & ~Ring
            & ~Ringback
            & ~Busytone
  TRANSITION Start_Tone
    ENTRY           [ TIME : T2 ]
              Offhook
            & My_Central.Phone_State ( Self )  = Ready_To_Dial
            & ~Dialtone
            & FORALL t: time
                    ( Change ( Dialtone, t )
                  ->   t < Change ( Offhook )  )
    EXIT
              Dialtone
  TRANSITION Enter_Digit ( D: Digit )
    ENTRY           [ TIME : T4 ]
              Offhook
            &  ( My_Central.Phone_State ( Self )  = Ready_To_Dial
               & Dialtone
              |  My_Central.Phone_State ( Self )  = Dialing )
    EXIT
              Next_Digit = D
            & ~Dialtone
  TRANSITION Start_Ring
    ENTRY           [ TIME : T5 ]
              ~Offhook
            & My_Central.Phone_State ( Self )  = Ringing
            & My_Central.Enabled_Ring_Pulse ( Self )
            & ~Ring
    EXIT
              Ring
  TRANSITION Stop_Ring
    ENTRY           [ TIME : T6 ]
              Ring
            & ~My_Central.Enabled_Ring_Pulse ( Self )
    EXIT
              ~Ring
  TRANSITION Start_Ringback
    ENTRY           [ TIME : T7 ]
              Offhook
            & ~Ringback
            & My_Central.Phone_State ( Self )  = Waiting
            & My_Central.Enabled_Ringback_Pulse ( Self )
    EXIT
              Ringback
  TRANSITION Stop_Ringback
    ENTRY           [ TIME : T8 ]
              Ringback
            & ~My_Central.Enabled_Ringback_Pulse ( Self )
    EXIT
              ~Ringback
  TRANSITION Start_Busytone
    ENTRY           [ TIME : T9 ]
              Offhook
            & My_Central.Phone_State ( Self )  = Busy
            & ~Busytone
    EXIT
              Busytone
  TRANSITION Stop_Busytone
    ENTRY           [ TIME : T10 ]
              Busytone
            & My_Central.Phone_State ( Self )  ~= Busy
    EXIT
              ~Busytone
  TRANSITION Hangup
    ENTRY           [ TIME : T11 ]
              Offhook
    EXIT
              ~Offhook
            & ~Dialtone
            & ~Ring
            & ~Ringback
            & ~Busytone
  END Top_Level
END Phone
PROCESS SPECIFICATION Central_Control
  LEVEL Top_Level
    IMPORT
              LD_Timeout, Digit, Digit_List, Connection, Phone_ID, Central_Control_ID, Enabled_State,
              Connection_Status, In_Area, Max_Cust, Plug, Phones.Offhook, Phones.Next_Digit, Phones.Pickup,
              Phones.Enter_Digit
    EXPORT
              Phone_State, Enabled_Ring_Pulse, Enabled_Ringback_Pulse, LDOut_Line, LDOut_Status, Receive_LD,
              Start_LD, Start_Talk_2, Terminate_LD_2
```

```
TYPE
          Area_Phone IS TYPEDEF p: Phone_ID ( In_Area ( p )  = Self )
CONSTANT
          Uptime_Ring, Downtime_Ring, Uptime_Ringback, Downtime_Ringback, LD_Timeout, Delta: Time,
          Tim1, Tim2, Tim3, Tim4, Tim5, Tim6, Tim7, Tim8, Tim9, Tim10, Tim11, Tim12, Tim13, Tim14, Tim15,
            Tim16: Time,
          Get_ID ( Digit_List ) : Area_Phone,
          Get_Number ( Area_Phone ) , Get_Area ( Central_Control_ID ) : Digit_List,
          Pick_Area ( Digit_List ) , Pick_Number ( Digit_list ) : Digit_List,
          MAX ( Time, Time, Time, Time, Time, Time, Time, Time, Time, Time, Time, Time, Time, Time,
                  Time, Time ) : Time
VARIABLE
          Phone_State ( Area_Phone ) : Enabled_State,
          Long_Distance ( Area_Phone ) : Boolean,
          Enabled_Ring_Pulse ( Area_Phone ) , Enabled_Ringback_Pulse ( Area_Phone ) : Boolean,
          Connected_To ( Area_Phone ) : Area_Phone,
          Number ( Area_Phone ) : Digit_List,
          LDOut_Line ( Area_Phone ) : Connection,
          LDOut_Status ( Area_Phone ) : Connection_Status
AXIOM
          FORALL d: Digit_List
                   ( LIST_LEN  ( Pick_Number ( d )  )  = 7
                   & LIST_LEN  ( Pick_Area ( d )  )  = 3 )
DEFINE
          Count ( P: Area_Phone ) : Integer ==
                  LIST_LEN  ( Number ( P )  ) ,
          Calling_Out ( P: Area_Phone, L: Connection ) : Boolean ==
                  P.Offhook
                & Long_Distance ( P )
                & Get_Area ( Self )  = L [ To_Area ]
                & Get_Number ( P )  = L [ To_Number ]
                & Plug ( LDOut_Line ( P ) , L )
ENVIRONMENT
          FORALL t: Time, L: Connection
                   ( Call ( Terminate_LD_2 ( L ) , t )
                 ->   EXISTS t1: Time
                             ( t1 < t
                             &  ( Call ( Receive_LD ( L ) , t1 )
                               |  Call ( Start_LD ( L ) , t1 )  )  )  )
        & FORALL t: Time, L: Connection
                   ( Call ( Start_Talk_2 ( L ) , t )
                 ->   EXISTS t1: Time
                             ( t1 < t
                             & Call ( Start_LD ( L ) , t1 )  )  )
        & FORALL t: Time, L: Connection
                   ( Call ( Start_LD ( L ) , t )
                 ->   EXISTS t1: Time, P: Area_Phone
                             ( t1 < t
                             & past ( Phone_State ( P ) , t1 )  = Calling
                             & past ( Plug ( LDOut_Line ( P ) , L ) , t )  )  )
        & FORALL t: Time
                   ( Call [ 2 ]  ( Receive_LD, t )
                 ->   Call ( Receive_LD )  - t > LD_Timeout )
INITIAL
          FORALL P: Area_Phone ( Phone_State ( P )  = Idle
                               & Number ( P )  = NIL
                               & ~Enabled_Ring_Pulse ( P )
                               & ~Enabled_Ringback_Pulse ( P )
                               & ~Long_Distance ( P )
                               & LDOut_Status ( P )  = Available )
INVARIANT
          FORALL P: Area_Phone
                   (  ( Long_Distance ( P )
                   ->   Count ( P )  >= 0
                      & Count ( P )  <= 11 )
                   &  ( ~Long_Distance ( P )
                   ->    ( Count ( P )  >= 0
                         & Count ( P )  <= 7
                         &  ( Phone_State ( P )  = Waiting
                         ->   Phone_State ( Connected_To ( P )  )  = Ringing )
                         &  ( Phone_State ( P )  = Ringing
                         ->   Phone_State ( Connected_To ( P )  )  = Waiting )
                         &  ( Phone_State ( P )  = Talk
                         ->   Phone_State ( Connected_To ( P )  )  = Talk )  )  )  )
CONSTRAINT
          FORALL P: Area_Phone (  ( Phone_State' ( P )  = Busy
                                  |  Phone_State' ( P )  = Alarm
                                  |  Phone_State' ( P )  = Disconnecting )
                                & Phone_State ( P )  ~= Phone_State' ( P )
                              ->   Phone_State ( P )  = Idle )
SCHEDULE
          FORALL P: Area_Phone, t, t1, t2: Time
                   ( t <= t1
                   & t1 < t2
                   & Change [ 2 ]  ( Phone_State ( P ) , t )
                   & past ( Phone_State ( P ) , t )  = Idle
```

```
                & P.End ( Pickup, t1 )
                & P.Offhook
                & Change ( Phone_State ( P ) , t2 )
             ->   past ( Phone_State ( P ) , t2 )  = Ringing
               |  past ( Phone_State ( P ) , t2 )  = Ready_To_Dial
                & t2 <= t1 + 2 )
       & FORALL P: Area_Phone
                ( Phone_State ( P )  = Ringing
                & Now - Change ( Phone_State ( P )  )  >= Downtime_Ring
             ->   EXISTS n: Integer
                          ( End [ n ]  ( Enable_Ring ( P )  )  > Change ( Phone_State ( P )  )
                          & End [ n ]  ( Enable_Ring ( P )  )  <= Change ( Phone_State ( P )  )  +
                             Downtime_Ring )  )
       & FORALL P: Area_Phone
                ( Phone_State ( P )  = Ringing
                & End ( Enable_Ring ( P )  )  > Change ( Phone_State ( P )  )
                & Now >= End ( Enable_Ring ( P )  )  + Uptime_Ring + Delta
             ->    ( End ( Disable_Ring_Pulse ( P )  )  >= End ( Enable_Ring ( P )  )  + Uptime_Ring
                   & End ( Disable_Ring_Pulse ( P )  )  <= End ( Enable_Ring ( P )  )  + Uptime_Ring +
                      Delta )  )
       & FORALL P: Area_Phone
                ( Phone_State ( P )  = Ringing
                & End ( Disable_Ring_Pulse ( P )  )  > Change ( Phone_State ( P )  )
                & Now >= End ( Disable_Ring_Pulse ( P )  )  + Downtime_Ring + Delta
             ->    ( End ( Enable_Ring ( P )  )  >= End ( Disable_Ring_Pulse ( P )  )  + Downtime_Ring
                   & End ( Enable_Ring ( P )  )  <= End ( Disable_Ring_Pulse ( P )  )  + Downtime_Ring +
                      Delta )  )
       & FORALL P: Area_Phone
                ( ~Long_Distance ( P )
                & Phone_State ( P )  = Waiting
                & Now - End ( Process_Local_Call ( P )  )  >= Downtime_Ring
             ->   EXISTS n, m: Integer
                          ( End [ n ]  ( Enable_Ring ( Connected_To ( P )  )  )  >
                             End ( Process_Local_Call ( P )  )
                          & End [ n ]  ( Enable_Ring ( Connected_To ( P )  )  )  <=
                             End ( Process_Local_Call ( P )  )  + Downtime_Ring
                          & End [ m ]  ( Enable_Ringback ( P )  )  > End ( Process_Local_Call ( P )  )
                          & End [ m ]  ( Enable_Ringback ( P )  )  <=
                             End [ n ]  ( Enable_Ring ( Connected_To ( P )  )  )  + 0.5 )  )
       & FORALL P: Area_Phone
                ( Phone_State ( P )  = Waiting
                & End ( Enable_Ringback ( P )  )  > Change ( Phone_State ( P )  )
                & Now >= End ( Enable_Ringback ( P )  )  + Uptime_Ringback + Delta
             ->    ( End ( Disable_Ringback_Pulse ( P )  )  >= End ( Enable_Ringback ( P )  )  +
                      Uptime_Ringback
                   & End ( Disable_Ringback_Pulse ( P )  )  <= End ( Enable_Ringback ( P )  )  +
                      Uptime_Ringback + Delta )  )
       & FORALL P: Area_Phone
                ( Phone_State ( P )  = Waiting
                & End ( Disable_Ringback_Pulse ( P )  )  > Change ( Phone_State ( P )  )
                & Now >= End ( Disable_Ringback_Pulse ( P )  )  + Downtime_Ringback + Delta
             ->    ( End ( Enable_Ringback ( P )  )  >= End ( Disable_Ringback_Pulse ( P )  )  +
                      Downtime_Ringback
                   & End ( Enable_Ringback ( P )  )  <= End ( Disable_Ringback_Pulse ( P )  )  +
                      Downtime_Ringback + Delta )  )
IMPORTED VARIABLE
         SET_SIZE (  { SETDEF P: Area_Phone ( Now - 2 <= P.Start ( Pickup )
                                          & P.Start ( Pickup )  <= Now )  }  )  <= Max_Cust
FURTHER ASSUMPTIONS #1
  FURTHER PROCESS ASSUMPTIONS
    TRANSITION SELECTION
            enabled_transitions CONTAINS  { Give_Dial_Tone }
          & TRUE
        ->   eligible_transitions =  { Give_Dial_Tone }
    CONSTANT REFINEMENT
            2 > MAX ( Tim1, Tim2, Tim3, Tim4, Tim5, Tim6, Tim7, Tim8, Tim9, Tim10, Tim11, Tim12, Tim13,
                      Tim14, Tim15, Tim16 )  +  ( Max_Cust + 1 )  * Tim1
TRANSITION Give_Dial_Tone ( P: Area_Phone )
  ENTRY          [ TIME : Tim1 ]
           P.Offhook
         & Phone_State ( P )  = Idle
  EXIT
           Phone_State ( P )  BECOMES Ready_To_Dial
TRANSITION Process_Digit ( P: Area_Phone )
  ENTRY          [ TIME : Tim2 ]
           P.Offhook
         &  (  ( Long_Distance ( P )
               & Count ( P )  < 11 )
           |   ( ~Long_Distance ( P )
               & Count ( P )  < 7 )  )
         &  (  ( Phone_State ( P )  = Ready_To_Dial
               & P.End ( Enter_Digit )  > End ( Give_Dial_Tone ( P )  )  )
           |   ( Phone_State ( P )  = Dialing )
            & P.End ( Enter_Digit )  > End ( Process_Digit ( P )  )  )
```

```
  EXIT
            IF
                    Phone_State' ( P )  = Ready_To_Dial
            THEN
                    IF
                            P.Next_Digit' = 1
                    THEN
                            Long_Distance ( P )  BECOMES True
                    ELSE
                            Long_Distance ( P )  BECOMES False
                    FI
                  & Phone_State ( P )  BECOMES Dialing
                  & Number ( P )  BECOMES LISTDEF  ( P.Next_Digit' )
            ELSE
                    Number ( P )  BECOMES Number' ( P )  CONCAT LISTDEF  ( P.Next_Digit' )
            FI
TRANSITION Process_Local_Call ( P: Area_Phone )
  ENTRY           [ TIME : Tim3 ]
            P.Offhook
          & ~Long_Distance ( P )
          & Count ( P )  = 7
          & Phone_State ( P )  = Dialing
          & ~Get_ID ( Number ( P )  ) .Offhook
          & Phone_State ( Get_ID ( Number ( P )  )  )  = Idle
  EXIT
            Phone_State ( Get_ID ( Number' ( P )  )  )  = Ringing
          & Phone_State ( P )  = Waiting
          & ~Long_Distance ( Get_ID ( Number' ( P )  )  )
          & Connected_To ( P )  = Get_ID ( Number' ( P )  )
          & Connected_To ( Get_ID ( Number' ( P )  )  )  = P
          & FORALL P1: Area_Phone
                     ( P1 ~= P
                     & P1 ~= Get_ID ( Number' ( P )  )
                   ->   NOCHANGE  ( Phone_State ( P1 )  )
                     & NOCHANGE  ( Connected_To ( P1 )  )  )
  EXCEPT          [ TIME : Tim3 ]
            P.Offhook
          & ~Long_Distance ( P )
          & Count ( P )  = 7
          & Phone_State ( P )  = Dialing
          &  ( Get_ID ( Number ( P )  ) .Offhook
            |  Phone_State ( Get_ID ( Number ( P )  )  )  ~= Idle )
  EXIT
            Phone_State ( P ) BECOMES Busy
TRANSITION Connect_Long_Distance ( P: Area_Phone )
  ENTRY           [ TIME : Tim4 ]
            P.Offhook
          & Count ( P )  = 11
          & Long_Distance ( P )
          & Phone_State ( P )  = Dialing
          & Pick_Area ( Number ( P )  )  ~= Get_Area ( Self )
  EXIT
            LDOut_Line ( P )  [ From_Area ]  = Get_Area ( Self )
          & LDOut_Line ( P )  [ From_Number ]  = Get_Number ( P )
          & LDOut_Line ( P )  [ To_Area ]  = Pick_Area ( Number' ( P )  )
          & LDOut_Line ( P )  [ To_Number ]  = Pick_Number ( Number' ( P )  )
          & LDOut_Status ( P ) BECOMES In_Progress
          & Phone_State ( P ) BECOMES Calling
          & FORALL P1: Area_Phone ( P1 ~= P
                                  ->   NOCHANGE ( LDOut_Line ( P1 )  )  )
  EXCEPT          [ TIME : Tim4 ]
            P.Offhook
          & Count ( P )  = 11
          & Long_Distance ( P )
          & Phone_State ( P )  = Dialing
          & Pick_Area ( Number ( P )  )  = Get_Area ( Self )
  EXIT
            Long_Distance ( P ) BECOMES False
          & Number ( P ) BECOMES Pick_Number ( Number' ( P )  )
TRANSITION Enable_Ring ( P: Area_Phone )
  ENTRY           [ TIME : Tim5 ]
            ~P.Offhook
          & Phone_State ( P )  = Ringing
          & ~Enabled_Ring_Pulse ( P )
          & FORALL t: Time ( End ( Disable_Ring_Pulse ( P ) , t )
                           & FORALL t1: Time ( t <= t1
                                             & t1 <= Now
                                           ->   past ( Phone_State ( P ) , t1 )  = Ringing )
                       ->   Now - t >= Downtime_Ring )
  EXIT
            Enabled_Ring_Pulse ( P ) BECOMES True
TRANSITION Disable_Ring_Pulse ( P: Area_Phone )
  ENTRY           [ TIME : Tim6 ]
            Enabled_Ring_Pulse ( P )
          &  ( Now - End ( Enable_Ring ( P )  )  >= Uptime_Ring
            |  P.Offhook )
```

```
  EXIT
            Enabled_Ring_Pulse ( P ) BECOMES False
TRANSITION Enable_Ringback ( P: Area_Phone )
  ENTRY           [ TIME : Tim7 ]
            P.Offhook
          & Phone_State ( P )  = Waiting
          & ~Enabled_Ringback_Pulse ( P )
          & FORALL t: Time
                     ( End ( Disable_Ringback_Pulse ( P ) , t )
                     & FORALL t1: Time
                                ( t <= t1
                                & t1 <= Now
                               ->   past ( Phone_State ( P ) , t1 )  = Waiting )
                  ->   Now - t >= Downtime_Ringback )
  EXIT
            Enabled_Ringback_Pulse ( P ) BECOMES True
TRANSITION Disable_Ringback_Pulse ( P: Area_Phone )
  ENTRY           [ TIME : Tim8 ]
            Enabled_Ringback_Pulse ( P )
          &  ( Now - End ( Enable_Ringback ( P )  )  >= Uptime_Ringback
            |  ~P.Offhook )
  EXIT
            Enabled_Ringback_Pulse ( P ) BECOMES False
TRANSITION Receive_LD ( LDIn_Line: Connection )
  ENTRY           [ TIME : Tim9 ]
            LDIn_Line [ To_Area ]  = Get_Area ( Self )
          & Phone_State ( Get_ID ( LDIn_Line [ To_Number ]  )  )  = Idle
          & ~Get_ID ( LDIn_Line [ To_Number ]  ) .Offhook
  EXIT
            Phone_State ( Get_ID ( LDIn_Line [ To_Number ]  )  )  BECOMES Ringing
          & Long_Distance ( Get_ID ( LDIn_Line [ To_Number ]  )  )
          & LDOut_Status ( Get_ID ( LDIn_Line [ To_Number ]  )  )  BECOMES Connect
          & Plug ( LDOut_Line ( Get_ID ( LDIn_Line [ To_Number ]  )  ) , LDIn_Line )
          & FORALL P: Area_Phone
                     ( P ~= Get_ID ( LDIn_Line [ To_Number ]  )
                  ->   NOCHANGE  ( LDOut_Line ( P )  )  )
TRANSITION Start_Talk_1 ( P: Area_Phone )
  ENTRY           [ TIME : Tim10 ]
            P.Offhook
          & Phone_State ( P )  = Ringing
  EXIT
            Phone_State ( P )  = Talk
          & IF
                    ~Long_Distance' ( P )
            THEN
                    Phone_State ( Connected_To' ( P )  )  = Talk
                  & FORALL P1: Area_Phone
                             ( P1 ~= P
                             & P1 ~= Connected_To' ( P )  )
               ->   NOCHANGE  ( Phone_State ( P )  )
            ELSE
                    LDOut_Status ( P )  BECOMES Talking
            FI
TRANSITION Start_Talk_2 ( LDIn_Line: Connection )
  ENTRY           [ TIME : Tim11 ]
            EXISTS P: Area_Phone
                     ( Calling_Out ( P, LDIn_Line )
                     & Phone_State ( P )  = Waiting
                     & LDOut_Status ( P )  = Connect )
  EXIT
            EXISTS P: Area_Phone
                     ( Calling_Out' ( P, LDIn_Line )
                     & Phone_State' ( P )  = Waiting
                     & LDOut_Status' ( P )  = Connect
                     & LDOut_Status ( P )  BECOMES Talking
                     & Phone_State ( P )  BECOMES Talk )
TRANSITION Start_LD ( LDIn_Line: Connection, LDIn_Status: Connection_Status )
  ENTRY           [ TIME : Tim12 ]
            LDIn_Status = Connect
          & EXISTS P: Area_Phone
                     ( Calling_Out ( P, LDIn_Line )
                     & Phone_State ( P )  = Calling
                     & LDOut_Status ( P )  = In_Progress )
  EXIT
            EXISTS P: Area_Phone
                     ( Calling_Out' ( P, LDIn_Line )
                     & Phone_State' ( P )  = Calling
                     & LDOut_Status' ( P )  = In_Progress
                     & LDOut_Status ( P )  BECOMES Connect
                     & Phone_State ( P )  BECOMES Waiting )
  EXCEPT          [ TIME : Tim12 ]
            LDIn_Status = Disconnect
          & EXISTS P: Area_Phone
                     ( Calling_Out ( P, LDIn_Line )
                     & Phone_State ( P )  = Calling
                     & LDOut_Status ( P )  = In_Progress )
```

```
  EXIT
           EXISTS P: Area_Phone
                    ( Calling_Out' ( P, LDIn_Line )
                    & Phone_State' ( P )  = Calling
                    & LDOut_Status' ( P )  = In_Progress
                    & LDOut_Status ( P )  BECOMES Available
                    & Phone_State ( P )  BECOMES Busy )
TRANSITION Terminate_LD_1 ( P: Area_Phone )
  ENTRY           [ TIME : Tim13 ]
           ~P.Offhook
         & Long_Distance ( P )
         & Phone_State ( P )  ~= Idle
         & Phone_State ( P )  ~= Ringing
         & LDOut_Line ( P )  [ From_Area ]  = Get_Area ( Self )
         & LDOut_Line ( P )  [ From_Number ]  = Get_Number ( P )
         & LDOut_Status ( P )  ~= Available
  EXIT
           Phone_State ( P )  BECOMES Idle
         & ~Enabled_Ringback_Pulse ( P )
         & LDOut_Status ( P )  BECOMES Available
TRANSITION Generate_Alarm ( P: Area_Phone )
  ENTRY           [ TIME : Tim14 ]
           P.Offhook
         &  ( Phone_State ( P )  = Ready_To_Dial
           |  Phone_State ( P )  = Dialing
            & P.Call ( Enter_Digit )  < Start ( Process_Digit ( P )  )  )
         &  ( Count ( P )  = 0
            & Now - End ( Give_Dial_Tone ( P )  )  > 30
           |  Count ( P )  > 0
            & Count ( P )  < 7
            & Now - End ( Process_Digit ( P )  )  > 20
           |  ~Long_Distance ( P )
            & Count ( P )  < 7
            & Now - End ( Give_Dial_Tone ( P )  )  > 100
           |  Long_Distance ( P )
            & Count ( P )  < 11
            & Now - End ( Give_Dial_Tone ( P )  )  > 100 )
  EXIT
           Phone_State ( P ) BECOMES Alarm
TRANSITION Terminate_Local_Call ( P: Area_Phone )
  ENTRY           [ TIME : Tim15 ]
           ~P.Offhook
         & ~Long_Distance ( P )
         & Phone_State ( P )  ~= Idle
         & Phone_State ( P )  ~= Ringing
  EXIT
           Phone_State ( P )  = Idle
         & ~Enabled_Ringback_Pulse ( P )
         & IF
                  Phone_State' ( P )  = Talk
                |  Phone_State' ( P )  = Waiting
           THEN
                  IF
                          Phone_State' ( P )  = Talk
                  THEN
                          Phone_State ( Connected_To' ( P )  )  = Disconnecting
                  ELSE
                          Phone_State ( Connected_To' ( P )  )  = Idle
                        & ~Enabled_Ring_Pulse ( Connected_To' ( P )  )
                  FI
                & FORALL P1: Area_Phone
                           ( P1 ~= P
                           & P1 ~= Connected_To' ( P )
                         ->   NOCHANGE  ( Phone_State ( P1 )  )  )
           ELSE
                  FORALL P1: Area_Phone
                           ( P1 ~= P
                         ->   NOCHANGE  ( Phone_State ( P1 )  )  )
           FI
TRANSITION Terminate_LD_2 ( LDIn_Line: Connection )
  ENTRY           [ TIME : Tim16 ]
           EXISTS P: Area_Phone
                    ( Calling_Out ( P, LDIn_Line )
                    & Phone_State ( P )  = Talk
                    & LDOut_Status ( P )  = Talking )
  EXIT
           EXISTS P: Area_Phone
                    ( Calling_Out' ( P, LDIn_Line )
                    & Phone_State' ( P )  = Talk
                    & LDOut_Status' ( P )  = Talking
                    & LDOut_Status ( P )  BECOMES Disconnect
                    & Phone_State ( P )  BECOMES Disconnecting )
```

```
        EXCEPT          [ TIME : Tim16 ]
                  EXISTS P: Area_Phone
                           ( Calling_Out ( P, LDIn_Line )
                           & Phone_State ( P )  = Ringing
                           & LDOut_Status ( P )  = Connect )
        EXIT
                  EXISTS P: Area_Phone
                           ( Calling_Out' ( P, LDIn_Line )
                           & Phone_State' ( P )  = Ringing
                           & LDOut_Status' ( P )  = Connect
                           & LDOut_Status ( P )  BECOMES Available
                           & Phone_State ( P )  BECOMES Idle
                           & ~Enabled_Ring_Pulse ( P )  )
    END Top_Level
  END Central_Control
END Phone_System
```

## A.6. Production Cell

```
SPECIFICATION Production_Cell
  GLOBAL SPECIFICATION Production_Cell
    PROCESSES
              robot: P_Robot,
              press: P_Press,
              feed: P_Feed,
              deposit: P_Deposit,
              the_table: P_Table,
              crane: P_Crane,
              feed_sensor: P_Feed_Sensor,
              deposit_sensor: P_Deposit_Sensor
    TYPE
              pos_real: TYPEDEF r: real ( r > 0 ) ,
              table_statuses:  ( at_belt, at_robot, moving_to_belt, moving_to_robot ) ,
              level_statuses:  ( at_lower, at_middle, at_upper, moving_to_lower, moving_to_middle,
                                 moving_to_upper ) ,
              arm_statuses:  ( extended, retracted, extending, retracting ) ,
              crane_statuses:  ( at_deposit, at_stockpile, moving_to_deposit, moving_to_stockpile )
    CONSTANT
              feed_speed, deposit_speed: pos_real,
              feed_length, deposit_length: pos_real,
              feed_response, deposit_response, table_response, deposit_sensor_response: pos_real,
              blank_length: pos_real
    AXIOM
      /* feed belt length must be big enough to accommodate 2 blanks plus the distance it takes the feed
         sensor to detect a blank */
              feed_length > blank_length + blank_length + feed_response * feed_speed
      /* deposit belt length must be big enough to accommodate 2 blanks plus the distance it takes the
         deposit sensor to detect a blank */
            & deposit_length > blank_length + blank_length + deposit_response * deposit_speed
  END Production_Cell
  PROCESS SPECIFICATION P_Robot
    LEVEL Top_Level
      IMPORT
                pos_real, the_table, the_table.h_status, the_table.v_status, table_statuses, press,
                press.press_status, arm_statuses, level_statuses, deposit_sensor, deposit_sensor.is_object,
                table_response
      EXPORT
                arm1_status, arm2_status, arm1_has_object, arm2_has_object
      TYPE
                robot_statuses:  ( arm1_at_table, arm1_at_press, arm2_at_press, arm2_at_deposit,
                                   moving_arm1_to_table, moving_arm1_to_press, moving_arm2_to_press,
                                   moving_arm2_to_deposit )
      CONSTANT
                t_move_arm1_to_table, t_move_arm2_to_press, t_move_arm2_to_deposit,
                  t_move_arm1_to_press: pos_real,
                t_move_arm: pos_real,
                rotate_arm_dur, arm_arrive_dur, move_arm_dur, arm_moved_dur, arm_object_dur: pos_real
      VARIABLE
                robot_status: robot_statuses,
                arm1_status, arm2_status: arm_statuses,
                arm1_has_object, arm2_has_object: boolean
      AXIOM
                table_response <= rotate_arm_dur
      INITIAL
                robot_status = arm2_at_deposit
              & arm1_status = retracted
              & arm2_status = retracted
              & arm1_has_object
              & ~arm2_has_object
```

```
INVARIANT
  /* arms are retracted when robot is moving */
           ( robot_status = moving_arm1_to_press
          |  robot_status = moving_arm2_to_deposit
          |  robot_status = moving_arm1_to_table
          |  robot_status = moving_arm2_to_press
        ->   arm1_status = retracted
           & arm2_status = retracted )
  /* arms have and don't have objects at right times */
        &  ( robot_status = moving_arm1_to_press
        ->   arm1_has_object )
        &  ( robot_status = moving_arm2_to_deposit
        ->   arm2_has_object )
        &  ( robot_status = moving_arm1_to_table
        ->   ~arm1_has_object )
        &  ( robot_status = moving_arm2_to_press
        ->   ~arm2_has_object )
  /* only drop at right place */
        &  ( ~arm2_has_object
           & Change ( arm2_has_object, now )
        ->   robot_status = arm2_at_deposit
           & arm2_status = extended )
  /* only pickup when something there to pickup */
        &  ( Start ( Arm2_Pickup, now )
        ->   EXISTS t: time
                    ( t < now
                    & End ( Arm1_Drop, t )
                    & FORALL t1: time
                              ( End ( Arm2_Pickup, t1 )
                           ->   t1 < t )  )  )
  /* blanks don't collide */
        &  ( Start ( Arm1_Drop, now )
        ->   FORALL t: time
                    ( End ( Arm1_Drop, t )
                 ->   EXISTS t1: time
                              ( t < t1
                              & t1 < now
                              & End ( Arm2_Pickup, t1 )  )  )  )
        &  ( Start ( Arm2_Drop, now )
        ->   FORALL t: time
                    ( End ( Arm2_Drop, t )
                 ->   EXISTS t1: time
                              ( t < t1
                              & t1 <= now
                              & past ( Change ( deposit_sensor.is_object, t1 ) , t1 )
                              & past ( deposit_sensor.is_object, t1 )  )  )  )
SCHEDULE
  /* don't collide with press */
           ( robot_status = arm1_at_press
        ->    ( press.press_status ~= at_upper
              & press.press_status ~= moving_to_upper
             |  arm1_status = retracted )  )
        &  ( robot_status = arm2_at_press
        ->    ( press.press_status ~= at_middle
              & press.press_status ~= moving_to_middle
             |  arm2_status = retracted )  )
  /* only pickup at right place */
        &  ( arm1_has_object
           & Change ( arm1_has_object, now )
        ->   robot_status = arm1_at_table
           & arm1_status = extended
           & the_table.h_status = at_robot
           & the_table.v_status = at_robot )
        &  ( arm2_has_object
           & Change ( arm2_has_object, now )
        ->   robot_status = arm2_at_press
           & arm2_status = extended
           & press.press_status = at_lower )
  /* only drop at right place */
        &  ( ~arm1_has_object
           & Change ( arm1_has_object, now )
        ->   robot_status = arm1_at_press
           & arm1_status = extended
           & press.press_status = at_middle )
  /* only pickup when something there to pickup */
        &  ( Start ( Arm1_Pickup, now )
        ->   EXISTS t: time
                    ( t <= now
                    & past ( Change ( the_table.v_status = at_robot
                                    & the_table.h_status = at_robot, t ) , t )
                    & past ( the_table.v_status = at_robot
                           & the_table.h_status = at_robot, t )
                    &  ( FORALL t1: time
                                 ( Start [ 2 ]  ( Arm1_Pickup, t1 )
                              ->   t1 < t )  )  )  )
```

```
IMPORTED VARIABLE
  /* press only moves from middle after arm1 drops blank */
          ( Change ( press.press_status, now )
          & press.press_status ~= at_middle
       ->   EXISTS t1, t2: time
                    ( FORALL t: time
                              ( Change [ 2 ]  ( press.press_status, t )
                              & past ( press.press_status, t )  = at_middle
                         ->   t < t1 )
                    & t1 < t2
                    & t2 <= now
                    & past ( Change ( arm1_has_object, t1 ) , t1 )
                    & ~past ( arm1_has_object, t1 )
                    & past ( arm1_status, t2 )  = retracted )  )
  /* press only moves from lower after arm2 picks up blank */
       &  ( Change ( press.press_status, now )
          & press.press_status ~= at_lower
       ->   FORALL t: time
                    ( Change [ 2 ]  ( press.press_status, t )
                    & past ( press.press_status, t )  = at_lower
                 ->   EXISTS t1, t2: time
                              ( t < t1
                              & t1 < t2
                              & t2 <= now
                              & past ( Change ( arm2_has_object, t1 ) , t1 )
                              & past ( arm2_has_object, t1 )
                              & past ( arm2_status, t2 )  = retracted )  )  )
  /* table only moves from robot after arm1 picks up blank */
       &  ( Change ( the_table.v_status = at_robot
                  & the_table.h_status = at_robot, now )
          & ~ ( the_table.v_status = at_robot
              & the_table.h_status = at_robot )
       ->   FORALL t: time
                    ( Change [ 2 ]  ( the_table.v_status = at_robot
                                    & the_table.h_status = at_robot, t )
                 ->   EXISTS t1: time
                              ( t <= t1
                              & t1 < now
                              & past ( Change ( arm1_has_object, t1 ) , t1 )
                              & past ( arm1_has_object, t1 )  )  )  )
  /* table moves within time table_response of arm1 picking up a blank */
       &  ( Change ( arm1_has_object, now - table_response )
          & past ( arm1_has_object, now - table_response )
          & past ( the_table.v_status = at_robot
                 & the_table.h_status = at_robot, now - table_response )
       ->   EXISTS t: time
                    ( now - table_response < t
                    & t <= now
                    & past ( Change ( the_table.v_status = at_robot
                                    & the_table.h_status = at_robot, t ) , t )
                    & ~past ( the_table.v_status = at_robot
                            & the_table.h_status = at_robot, t )  )  )
TRANSITION Rot_Arm1_CCW_To_Press
  ENTRY           [ TIME : rotate_arm_dur ]
            robot_status = arm2_at_deposit
          & ~arm2_has_object
          & arm2_status = retracted
  EXIT
            robot_status = moving_arm1_to_press
TRANSITION Arm1_Arrived_At_Press
  ENTRY           [ TIME : arm_arrive_dur ]
            robot_status = moving_arm1_to_press
          & now - Change ( robot_status )  >= t_move_arm1_to_press
  EXIT
            robot_status = arm1_at_press
TRANSITION Rot_Arm1_CW_To_Table
  ENTRY           [ TIME : rotate_arm_dur ]
            robot_status = arm1_at_press
          & ~arm1_has_object
          & arm1_status = retracted
  EXIT
            robot_status = moving_arm1_to_table
TRANSITION Arm1_Arrived_At_Table
  ENTRY           [ TIME : arm_arrive_dur ]
            robot_status = moving_arm1_to_table
          & now - Change ( robot_status )  >= t_move_arm1_to_table
  EXIT
            robot_status = arm1_at_table
TRANSITION Rot_Arm2_CCW_To_Press
  ENTRY           [ TIME : rotate_arm_dur ]
            robot_status = arm1_at_table
          & arm1_has_object
          & arm1_status = retracted
  EXIT
            robot_status = moving_arm2_to_press
```

```
TRANSITION Arm2_Arrived_At_Press
  ENTRY           [ TIME : arm_arrive_dur ]
           robot_status = moving_arm2_to_press
         & now - Change ( robot_status )  >= t_move_arm2_to_press
  EXIT
           robot_status = arm2_at_press
TRANSITION Rot_Arm2_CCW_To_Deposit
  ENTRY           [ TIME : rotate_arm_dur ]
           robot_status = arm2_at_press
         & arm2_has_object
         & arm2_status = retracted
  EXIT
           robot_status = moving_arm2_to_deposit
TRANSITION Arm2_Arrived_At_Deposit
  ENTRY           [ TIME : arm_arrive_dur ]
           robot_status = moving_arm2_to_deposit
         & now - Change ( robot_status )  >= t_move_arm2_to_deposit
  EXIT
           robot_status = arm2_at_deposit
TRANSITION Extend_Arm1
  ENTRY           [ TIME : move_arm_dur ]
            ( robot_status = arm1_at_table
            & ~arm1_has_object
            & the_table.h_status = at_robot
            & the_table.v_status = at_robot
           |  robot_status = arm1_at_press
            & press.press_status = at_middle
            & arm1_has_object )
         & arm1_status = retracted
  EXIT
           arm1_status = extending
TRANSITION Arm1_Extended
  ENTRY           [ TIME : arm_moved_dur ]
           arm1_status = extending
         & now - Change ( arm1_status )  >= t_move_arm
  EXIT
           arm1_status = extended
TRANSITION Extend_Arm2
  ENTRY           [ TIME : move_arm_dur ]
            ( robot_status = arm2_at_press
            & ~arm2_has_object
            & press.press_status = at_lower
           |  robot_status = arm2_at_deposit
            & arm2_has_object )
         & arm2_status = retracted
  EXIT
           arm2_status = extending
TRANSITION Arm2_Extended
  ENTRY           [ TIME : arm_moved_dur ]
           arm2_status = extending
         & now - Change ( arm2_status )  >= t_move_arm
  EXIT
           arm2_status = extended
TRANSITION Retract_Arm1
  ENTRY           [ TIME : move_arm_dur ]
            ( robot_status = arm1_at_table
            & arm1_has_object
           |  robot_status = arm1_at_press
            & ~arm1_has_object )
         & arm1_status = extended
  EXIT
           arm1_status = retracting
TRANSITION Arm1_Retracted
  ENTRY           [ TIME : arm_moved_dur ]
           arm1_status = retracting
         & now - Change ( arm1_status )  >= t_move_arm
  EXIT
           arm1_status = retracted
TRANSITION Retract_Arm2
  ENTRY           [ TIME : move_arm_dur ]
            ( robot_status = arm2_at_press
            & arm2_has_object
           |  robot_status = arm2_at_deposit
            & ~arm2_has_object )
         & arm2_status = extended
  EXIT
           arm2_status = retracting
TRANSITION Arm2_Retracted
  ENTRY           [ TIME : arm_moved_dur ]
           arm2_status = retracting
         & now - Change ( arm2_status )  >= t_move_arm
  EXIT
           arm2_status = retracted
```

```
    TRANSITION Arm1_Pickup
      ENTRY           [ TIME : arm_object_dur ]
                ~arm1_has_object
              & robot_status = arm1_at_table
              & arm1_status = extended
              & the_table.h_status = at_robot
              & the_table.v_status = at_robot
      EXIT
                arm1_has_object
    TRANSITION Arm1_Drop
      ENTRY           [ TIME : arm_object_dur ]
                arm1_has_object
              & robot_status = arm1_at_press
              & arm1_status = extended
              & press.press_status = at_middle
      EXIT
                ~arm1_has_object
    TRANSITION Arm2_Pickup
      ENTRY           [ TIME : arm_object_dur ]
                ~arm2_has_object
              & robot_status = arm2_at_press
              & arm2_status = extended
              & press.press_status = at_lower
      EXIT
                arm2_has_object
    TRANSITION Arm2_Drop
      ENTRY           [ TIME : arm_object_dur ]
                arm2_has_object
              & robot_status = arm2_at_deposit
              & arm2_status = extended
              & FORALL t: time
                         ( End ( Arm2_Drop, t )
                       ->   EXISTS t1: time
                                    ( t < t1
                                    & t1 <= now
                                    & past ( Change ( deposit_sensor.is_object, t1 ) , t1 )
                                    & past ( deposit_sensor.is_object, t1 )  )  )
      EXIT
                ~arm2_has_object
  END Top_Level
END P_Robot
PROCESS SPECIFICATION P_Press
  LEVEL Top_Level
    IMPORT
              pos_real, level_statuses, robot, robot.arm1_has_object, robot.arm2_has_object, robot.arm1_status,
              robot.arm2_status, arm_statuses
    EXPORT
              press_status
    CONSTANT
              t_move_press_level: pos_real,
              move_press_dur, press_arrive_dur: pos_real
    VARIABLE
              press_status: level_statuses
    INITIAL
              press_status = at_middle
    INVARIANT
      /* press only moves from middle after arm1 drops blank */
               ( Change ( press_status, now )
               & press_status ~= at_middle
            ->   EXISTS t1, t2: time
                         ( FORALL t: time
                                    ( Change [ 2 ]  ( press_status, t )
                                    & past ( press_status, t )  = at_middle
                                  ->   t < t1 )
                         & t1 < t2
                         & t2 <= now
                         & past ( Change ( robot.arm1_has_object, t1 ) , t1 )
                         & ~past ( robot.arm1_has_object, t1 )
                         & past ( robot.arm1_status, t2 )  = retracted )  )
      /* press only moves from lower after arm2 picks up blank */
            &  ( Change ( press_status, now )
               & press_status ~= at_lower
            ->   FORALL t: time
                         ( Change [ 2 ]  ( press_status, t )
                         & past ( press_status, t )  = at_lower
                       ->   EXISTS t1, t2: time
                                    ( t < t1
                                    & t1 < t2
                                    & t2 <= now
                                    & past ( Change ( robot.arm2_has_object, t1 ) , t1 )
                                    & past ( robot.arm2_has_object, t1 )
                                    & past ( robot.arm2_status, t2 )  = retracted )  )  )
```

```
    TRANSITION Move_To_Upper
      ENTRY           [ TIME : move_press_dur ]
                press_status = at_middle
              & EXISTS t1, t2: time
                         ( t1 < t2
                         & t2 <= now
                         & past ( Change ( robot.arm1_has_object, t1 ) , t1 )
                         & ~past ( robot.arm1_has_object, t1 )
                         & past ( robot.arm1_status, t2 )  = retracted
                         & FORALL t3: time
                                    ( Change ( press_status, t3 )
                                ->   t3 < t1 )  )
      EXIT
                press_status = moving_to_upper
    TRANSITION Arrived_At_Upper
      ENTRY           [ TIME : press_arrive_dur ]
                press_status = moving_to_upper
              & now - Change ( press_status )  >= t_move_press_level
      EXIT
                press_status = at_upper
    TRANSITION Move_To_Lower
      ENTRY           [ TIME : move_press_dur ]
                press_status = at_upper
      EXIT
                press_status = moving_to_lower
    TRANSITION Arrived_At_Lower
      ENTRY           [ TIME : press_arrive_dur ]
                press_status = moving_to_lower
              & now - Change ( press_status )  >= t_move_press_level + t_move_press_level
      EXIT
                press_status = at_lower
    TRANSITION Move_To_Middle
      ENTRY           [ TIME : move_press_dur ]
                press_status = at_lower
              & EXISTS t1, t2: time
                         ( Change ( press_status )  < t1
                         & t1 < t2
                         & t2 <= now
                         & past ( Change ( robot.arm2_has_object, t1 ) , t1 )
                         & past ( robot.arm2_has_object, t1 )
                         & past ( robot.arm2_status, t2 )  = retracted )
      EXIT
                press_status = moving_to_middle
    TRANSITION Arrived_At_Middle
      ENTRY           [ TIME : press_arrive_dur ]
                press_status = moving_to_middle
              & now - Change ( press_status )  >= t_move_press_level
      EXIT
                press_status = at_middle
  END Top_Level
END P_Press
PROCESS SPECIFICATION P_Feed
  LEVEL Top_Level
    IMPORT
              pos_real, feed_sensor, feed_sensor.is_object, the_table, the_table.h_status, the_table.v_status,
              table_statuses, feed_response
    EXPORT
              Add_Blank, moving
    CONSTANT
              move_feed_dur, add_blank_dur: pos_real
    VARIABLE
              moving: boolean
    AXIOM
      /* the time it takes to start/stop the feed belt must be less than the required feed response time */
              move_feed_dur <= feed_response
    INITIAL
              ~moving
    SCHEDULE
      /* blank won't be moved off belt unless table is in right place */
               ( moving
               & feed_sensor.is_object
               & Change ( moving )  > Change ( feed_sensor.is_object )
            ->   the_table.h_status = at_belt
               & the_table.v_status = at_belt )
      /* table stops between when a blank arrives and when it departs */
            &  ( Change ( feed_sensor.is_object, now )
               & ~feed_sensor.is_object
            ->   EXISTS t: time
                         ( Change [ 2 ]  ( feed_sensor.is_object )  < t
                         & t < now
                         & ~past ( moving, t )  )  )
    IMPORTED VARIABLE
      /* table only moves from feed belt after blank loaded */
               ( Change ( the_table.v_status = at_belt
                        & the_table.h_status = at_belt, now )
```

```
              & ~ ( the_table.v_status = at_belt
                  & the_table.h_status = at_belt )
            ->   EXISTS t1: time
                          ( FORALL t: time
                                    ( Change [ 2 ]  ( the_table.v_status = at_belt
                                                    & the_table.h_status = at_belt, t )
                                  ->   t < t1 )
                          & t1 <= now
                          & past ( Change ( feed_sensor.is_object, t1 ) , t1 )
                          & ~past ( feed_sensor.is_object, t1 )  )  )
      /* feed sensor asserts is_object for at least feed_response time */
            &  ( Change ( feed_sensor.is_object, now )
               & ~feed_sensor.is_object
            ->   now - Change [ 2 ]  ( feed_sensor.is_object )  > feed_response )
    TRANSITION Start_Move
      ENTRY           [ TIME : move_feed_dur ]
                ~moving
              & ~feed_sensor.is_object
      EXIT
                moving
    TRANSITION Stop_Move
      ENTRY           [ TIME : move_feed_dur ]
                moving
              & feed_sensor.is_object
              & EXISTS t: time
                         (  ( Start ( Start_Move, t )
                           |  Start ( Move_Object_Onto_Table, t )  )
                         & t < Change ( feed_sensor.is_object )  )
      EXIT
                ~moving
    TRANSITION Move_Object_Onto_Table
      ENTRY           [ TIME : move_feed_dur ]
                ~moving
              & feed_sensor.is_object
              & the_table.h_status = at_belt
              & the_table.v_status = at_belt
      EXIT
                moving
    TRANSITION Add_Blank
      ENTRY           [ TIME : add_blank_dur ]
                FORALL t: time
                          ( End ( Add_Blank, t )
                        ->   EXISTS t1: time
                                    ( t < t1
                                    & t1 < now
                                    & past ( Change ( feed_sensor.is_object, t1 ) , t1 )
                                    & past ( feed_sensor.is_object, t1 )  )  )
      EXIT
                TRUE
  END Top_Level
END P_Feed
PROCESS SPECIFICATION P_Deposit
  LEVEL Top_Level
    IMPORT
              pos_real, deposit_sensor, deposit_sensor.is_object, deposit_response
    EXPORT
              moving
    CONSTANT
              move_deposit_dur: pos_real
    VARIABLE
              moving: boolean
    AXIOM
      /* the time it takes to start/stop the deposit belt must be less than the required deposit response
         time */
              move_deposit_dur <= deposit_response
    INITIAL
              ~moving
    TRANSITION Start_Move
      ENTRY           [ TIME : move_deposit_dur ]
                ~moving
              & ~deposit_sensor.is_object
      EXIT
                moving
    TRANSITION Stop_Move
      ENTRY           [ TIME : move_deposit_dur ]
                moving
              & deposit_sensor.is_object
      EXIT
                ~moving
  END Top_Level
END P_Deposit
PROCESS SPECIFICATION P_Table
  LEVEL Top_Level
    IMPORT
              pos_real, table_statuses, feed_sensor, feed_sensor.is_object, robot, robot.arm1_has_object,
              table_response
```

```
EXPORT
         h_status, v_status
CONSTANT
         t_move_table_level, t_rotate_table: pos_real,
         move_table_dur, rotate_table_dur, table_arrive_dur: pos_real
VARIABLE
         h_status: table_statuses,
         v_status: table_statuses
AXIOM
         table_response >= rotate_table_dur
INITIAL
         h_status = at_belt
       & v_status = at_belt
INVARIANT
  /* table only moves from robot after arm1 picks up blank */
          ( Change ( v_status = at_robot
                   & h_status = at_robot, now )
          & ~ ( v_status = at_robot
              & h_status = at_robot )
       ->   FORALL t: time
                    ( Change [ 2 ]  ( v_status = at_robot
                                    & h_status = at_robot, t )
                  ->   EXISTS t1: time
                               ( t <= t1
                               & t1 < now
                               & past ( Change ( robot.arm1_has_object, t1 ) , t1 )
                               & past ( robot.arm1_has_object, t1 )  )  )  )
  /* table only moves from feed belt after blank loaded */
       &  ( Change ( v_status = at_belt
                   & h_status = at_belt, now )
          & ~ ( v_status = at_belt
              & h_status = at_belt )
       ->   EXISTS t1: time
                    ( FORALL t: time
                               ( Change [ 2 ]  ( v_status = at_belt
                                               & h_status = at_belt, t )
                            ->   t <= t1 )
                    & t1 < now
                    & past ( Change ( feed_sensor.is_object, t1 ) , t1 )
                    & ~past ( feed_sensor.is_object, t1 )  )  )
  /* table moves within time table_response of arm1 picking up a blank */
       &  ( Change ( robot.arm1_has_object, now - table_response )
          & past ( robot.arm1_has_object, now - table_response )
          & past ( v_status = at_robot
                 & h_status = at_robot, now - table_response )
       ->   EXISTS t: time
                    ( now - table_response < t
                    & t <= now
                    & past ( Change ( v_status = at_robot
                                    & h_status = at_robot, t ) , t )
                    & ~past ( v_status = at_robot
                            & h_status = at_robot, t )  )  )
TRANSITION Move_To_Upper
  ENTRY           [ TIME : move_table_dur ]
           h_status = at_belt
         & v_status = at_belt
         & EXISTS t: time
                   ( t <= now
                   & past ( Change ( feed_sensor.is_object, t ) , t )
                   & ~past ( feed_sensor.is_object, t )
                   & FORALL t1: time
                              ( Change ( v_status, t1 )
                            ->   t1 <= t )  )
  EXIT
           v_status = moving_to_robot
TRANSITION Arrived_At_Upper
  ENTRY           [ TIME : table_arrive_dur ]
           v_status = moving_to_robot
         & now - Change ( v_status )  >= t_move_table_level
  EXIT
           v_status = at_robot
TRANSITION Rot_CW_To_Robot
  ENTRY           [ TIME : rotate_table_dur ]
           h_status = at_belt
         & v_status = at_robot
  EXIT
           h_status = moving_to_robot
TRANSITION Arrived_At_Robot
  ENTRY           [ TIME : table_arrive_dur ]
           h_status = moving_to_robot
         & now - Change ( h_status )  >= t_rotate_table
  EXIT
           h_status = at_robot
```

```
    TRANSITION Rot_CCW_To_Feed
      ENTRY           [ TIME : rotate_table_dur ]
                h_status = at_robot
              & v_status = at_robot
              & EXISTS t: time
                         ( Change ( v_status )  <= t
                         & t <= now
                         & past ( Change ( robot.arm1_has_object, t ) , t )
                         & past ( robot.arm1_has_object, t )  )
      EXIT
                h_status = moving_to_belt
    TRANSITION Arrived_At_Feed
      ENTRY           [ TIME : table_arrive_dur ]
                h_status = moving_to_belt
              & now - Change ( h_status )  >= t_rotate_table
      EXIT
                h_status = at_belt
    TRANSITION Move_To_Lower
      ENTRY           [ TIME : move_table_dur ]
                h_status = at_belt
              & v_status = at_robot
      EXIT
                v_status = moving_to_belt
    TRANSITION Arrived_At_Lower
      ENTRY           [ TIME : table_arrive_dur ]
                v_status = moving_to_belt
              & now - Change ( v_status )  >= t_move_table_level
      EXIT
                v_status = at_belt
  END Top_Level
END P_Table
PROCESS SPECIFICATION P_Crane
  LEVEL Top_Level
    IMPORT
              pos_real, deposit_sensor, deposit_sensor.is_object, level_statuses, crane_statuses,
              deposit_sensor_response
    EXPORT
              gripper_has_object
    CONSTANT
              t_move_crane, t_move_gripper: pos_real,
              move_crane_dur, crane_arrive_dur: pos_real,
              move_gripper_dur, gripper_arrive_dur, gripper_object_dur: pos_real
    VARIABLE
              crane_status: crane_statuses,
              gripper_status: level_statuses,
              gripper_has_object: boolean
    AXIOM
              deposit_sensor_response < t_move_gripper
    INITIAL
              crane_status = at_deposit
            & gripper_status = at_upper
            & ~gripper_has_object
    INVARIANT
      /* gripper has and doesn't have object at right times */
               ( crane_status = moving_to_stockpile
              ->   gripper_has_object )
              &  ( crane_status = moving_to_deposit
              ->   ~gripper_has_object )
      /* only drop at right place */
             &  ( ~gripper_has_object
                & Change ( gripper_has_object, now )
              ->   crane_status = at_stockpile
                 & gripper_status = at_lower )
    SCHEDULE
      /* only pickup when something there to pickup */
               ( Start ( Gripper_Pickup, now )
              ->   EXISTS t: time
                           ( t <= now
                           & past ( Change ( deposit_sensor.is_object, t ) , t )
                           & past ( deposit_sensor.is_object, t )
                           & FORALL t1: time
                                      ( End ( Gripper_Pickup, t1 )
                                    ->   t1 <= t )  )  )
    IMPORTED VARIABLE
      /* deposit sensor will detect object departure within time deposit_sensor_response */
                ( Change ( gripper_has_object, now - deposit_sensor_response )
                & past ( gripper_has_object, now - deposit_sensor_response )
              ->   EXISTS t: time
                           ( now - deposit_sensor_response <= t
                           & t <= now
                           & ~past ( deposit_sensor.is_object, t )  )  )
    TRANSITION Move_To_Stockpile
      ENTRY           [ TIME : move_crane_dur ]
                gripper_has_object
              & crane_status = at_deposit
              & gripper_status = at_upper
```

```
    EXIT
              crane_status = moving_to_stockpile
  TRANSITION Arrived_At_Stockpile
    ENTRY           [ TIME : crane_arrive_dur ]
              crane_status = moving_to_stockpile
            & now - Change ( crane_status )  >= t_move_crane
    EXIT
              crane_status = at_stockpile
  TRANSITION Move_To_Deposit
    ENTRY           [ TIME : move_crane_dur ]
              ~gripper_has_object
            & crane_status = at_stockpile
            & gripper_status = at_upper
    EXIT
              crane_status = moving_to_deposit
  TRANSITION Arrived_At_Deposit
    ENTRY           [ TIME : crane_arrive_dur ]
              crane_status = moving_to_deposit
            & now - Change ( crane_status )  >= t_move_crane
    EXIT
              crane_status = at_deposit
  TRANSITION Move_To_Lower
    ENTRY           [ TIME : move_gripper_dur ]
              crane_status = at_deposit
            & gripper_status = at_upper
    EXIT
              gripper_status = moving_to_lower
  TRANSITION Arrived_At_Lower
    ENTRY           [ TIME : gripper_arrive_dur ]
              gripper_status = moving_to_lower
            & now - Change ( gripper_status )  >= t_move_gripper
    EXIT
              gripper_status = at_lower
  TRANSITION Move_To_Upper
    ENTRY           [ TIME : move_gripper_dur ]
              crane_status = at_stockpile
            & ~gripper_has_object
            & gripper_status = at_middle
           |  crane_status = at_deposit
            & gripper_has_object
            & gripper_status = at_lower
    EXIT
              gripper_status = moving_to_upper
  TRANSITION Arrived_At_Upper
    ENTRY           [ TIME : gripper_arrive_dur ]
              gripper_status = moving_to_upper
            & now - Change ( gripper_status )  >= t_move_gripper
    EXIT
              gripper_status = at_upper
  TRANSITION Move_To_Middle
    ENTRY           [ TIME : move_gripper_dur ]
              crane_status = at_stockpile
            & gripper_status = at_upper
    EXIT
              gripper_status = moving_to_middle
  TRANSITION Arrived_At_Middle
    ENTRY           [ TIME : gripper_arrive_dur ]
              gripper_status = moving_to_middle
            & now - Change ( gripper_status )  >= t_move_gripper
    EXIT
              gripper_status = at_middle
  TRANSITION Gripper_Pickup
    ENTRY           [ TIME : gripper_object_dur ]
              ~gripper_has_object
            & crane_status = at_deposit
            & gripper_status = at_lower
            & deposit_sensor.is_object
    EXIT
              gripper_has_object
  TRANSITION Gripper_Drop
    ENTRY           [ TIME : gripper_object_dur ]
              gripper_has_object
            & crane_status = at_stockpile
            & gripper_status = at_middle
    EXIT
              ~gripper_has_object
  END Top_Level
END P_Crane
PROCESS SPECIFICATION P_Feed_Sensor
  LEVEL Top_Level
    IMPORT
              pos_real, feed_length, feed_speed, blank_length, feed, feed.moving, feed.Add_Blank, crane,
              crane.gripper_has_object, feed_response
    EXPORT
              is_object
```

```
    CONSTANT
              sensor_dur: pos_real
    VARIABLE
              is_object: boolean
    INITIAL
              ~is_object
    INVARIANT
      /* feed sensor asserts is_object for at least feed_response time */
               ( Change ( is_object, now )
               & ~is_object
           ->   now - Change [ 2 ]  ( is_object )  > feed_response )
    TRANSITION Object_Arrive
      ENTRY           [ TIME : sensor_dur ]
                ~is_object
              & feed.moving
              & EXISTS t: time
                         (  ( Change ( crane.gripper_has_object, t )
                            & ~past ( crane.gripper_has_object, t )
                           |  feed.End ( Add_Blank, t )  )
                         & FORALL t1: time
                                    ( Change ( is_object, t1 )
                                    & past ( is_object, t1 )
                                ->   t1 < t )
                         &  ( t < Change ( feed.moving )
                         ->   now - Change ( feed.moving )  >=  ( feed_length - blank_length )  / feed_speed -
                                sensor_dur - feed_response )
                         &  ( t >= Change ( feed.moving )
                         ->   now - t >=  ( feed_length - blank_length ) / feed_speed - sensor_dur -
                                              feed_response ) )
      EXIT
                is_object
    TRANSITION Object_Depart
      ENTRY           [ TIME : sensor_dur ]
                is_object
              & feed.moving
              & Change ( feed.moving )  > Change ( is_object )
              & now - Change ( feed.moving )  >= blank_length / feed_speed + feed_response
      EXIT
                ~is_object
  END Top_Level
END P_Feed_Sensor
PROCESS SPECIFICATION P_Deposit_Sensor
  LEVEL Top_Level
    IMPORT
              pos_real, deposit_length, deposit_speed, deposit, deposit.moving, robot, robot.arm2_has_object,
              crane, crane.gripper_has_object, deposit_sensor_response, deposit_response, blank_length
    EXPORT
              is_object
    CONSTANT
              sensor_dur: pos_real
    VARIABLE
              is_object: boolean
    AXIOM
              sensor_dur + sensor_dur <= deposit_sensor_response
    INITIAL
              ~is_object
    INVARIANT
      /* deposit sensor will detect object departure within time deposit_sensor_response */
               ( Change ( crane.gripper_has_object, now - deposit_sensor_response )
               & past ( crane.gripper_has_object, now - deposit_sensor_response )
           ->   EXISTS t: time
                           ( now - deposit_sensor_response <= t
                           & t <= now
                           & ~past ( is_object, t )  )  )
    TRANSITION Object_Arrive
      ENTRY           [ TIME : sensor_dur ]
                ~is_object
              & deposit.moving
              & EXISTS t: time
                         ( Change ( robot.arm2_has_object, t )
                         & ~past ( robot.arm2_has_object, t )
                         & FORALL t1: time
                                    ( Change ( is_object, t1 )
                                    & past ( is_object, t1 )
                                ->   t1 < t )
                         &  ( t < Change ( deposit.moving )
                         ->   now - Change ( deposit.moving )  >=  ( deposit_length - blank_length )  /
                                deposit_speed - sensor_dur - deposit_response )
                         &  ( t >= Change ( deposit.moving )
                         ->   now - t >= deposit_length / deposit_speed )  )
      EXIT
                is_object
```

```
      TRANSITION Object_Depart
        ENTRY           [ TIME : sensor_dur ]
                  is_object
                & EXISTS t: time
                           ( Change ( is_object )  <= t
                           & t <= now
                           & past ( Change ( crane.gripper_has_object, t ) , t )
                           & past ( crane.gripper_has_object, t )  )
        EXIT
                  ~is_object
    END Top_Level
  END P_Deposit_Sensor
END Production_Cell
```

## A.7. Railroad Crossing

```
SPECIFICATION Railroad_Crossing
  GLOBAL SPECIFICATION Railroad_Crossing
    PROCESSES
              the_gate: Gate,
              the_sensors: array [ 1..n_tracks ]  of Sensor
    TYPE
              pos_integer: TYPEDEF i: integer ( i > 0 ) ,
              pos_real: TYPEDEF i: real ( i > 0 ) ,
              gate_position:  ( raised, raising, lowered, lowering ) ,
              sensor_id: TYPEDEF i: id ( IDTYPE  ( i )  = Sensor )
    CONSTANT
              n_tracks: pos_integer,
              min_speed, max_speed: pos_real,
              dist_R_to_I, dist_I_to_out: pos_real,
              response_time, wait_time: pos_real
    AXIOM
              max_speed >= min_speed
            & response_time < dist_R_to_I / max_speed
    SCHEDULE
      /* gate will be down before fastest train reaches crossing */
               ( EXISTS s: sensor_id
                           ( s.train_in_R
                           & now - s.Call ( enter_R )  >= dist_R_to_I / max_speed )
            ->   the_gate.position = lowered )
      /* gate will be up after slowest train exits crossing and a reasonable wait time has elapsed */
            &  ( FORALL s: sensor_id
                           ( ~s.train_in_R
                           &  ( EXISTS t: time
                                       ( s.Call ( enter_R, t )  )
                           ->   now - s.Call ( enter_R )  >=  ( dist_R_to_I + dist_I_to_out )  / min_speed +
                                  wait_time )  )
            ->   the_gate.position = raised )
  END Railroad_Crossing
  PROCESS SPECIFICATION Sensor
    LEVEL Top_Level
      IMPORT
                pos_real, max_speed, min_speed, dist_R_to_I, dist_I_to_out, response_time
      EXPORT
                train_in_R, enter_R
      CONSTANT
                enter_dur, exit_dur: pos_real
      VARIABLE
                train_in_R: boolean
      AXIOM
                response_time >= enter_dur
              &  ( dist_R_to_I + dist_I_to_out )  / min_speed >= response_time + exit_dur
      ENVIRONMENT
        /* only one train will be in the region at the same time on the same track */
                Call ( enter_R, now )
              & EXISTS t: time
                         ( t >= 0
                         & t <= now
                         & Call [ 2 ]  ( enter_R, t )  )
           ->   Call ( enter_R )  - Call [ 2 ]  ( enter_R )  >  ( dist_R_to_I + dist_I_to_out )  / min_speed
      INITIAL
                ~train_in_R
      INVARIANT
        /* once a sensor reports a train, it will keep reporting a train at least as long as it takes the
           fastest train to exit the region */
                Change ( train_in_R, now )
              & ~train_in_R
           ->   0 <= now -  (  ( dist_R_to_I + dist_I_to_out )  / max_speed - response_time )
              & FORALL t: time
                         ( now -  (  ( dist_R_to_I + dist_I_to_out )  / max_speed - response_time )  <=
                            t
                         & t < now
                      ->   past ( train_in_R, t )  )
```

```
    SCHEDULE
      /* train will be sensed within enter_dur of call */
               ( now >= response_time
               & Call ( enter_R, now - response_time )
            ->   train_in_R )
      /* sensor will be reset when the slowest train is beyond the crossing */
            &  ( now >=  ( dist_R_to_I + dist_I_to_out )  / min_speed
               & Call ( enter_R, now -  ( dist_R_to_I + dist_I_to_out )  / min_speed )
            ->   ~train_in_R )
    TRANSITION enter_R
      ENTRY           [ TIME : enter_dur ]
                 ~train_in_R
      EXIT
                 train_in_R
    TRANSITION exit_I
      ENTRY           [ TIME : exit_dur ]
                 train_in_R
               & now - Start ( enter_R )  >=  ( dist_R_to_I + dist_I_to_out )  / min_speed - exit_dur
      EXIT
                 ~train_in_R
  END Top_Level
END Sensor
PROCESS SPECIFICATION Gate
  LEVEL Top_Level
    IMPORT
               pos_real, gate_position, max_speed, dist_R_to_I, dist_I_to_out, wait_time, response_time,
               sensor_id, the_sensors.train_in_R
    EXPORT
               position
    CONSTANT
               lower_dur, raise_dur, up_dur, down_dur: pos_real,
               raise_time, lower_time: pos_real
    VARIABLE
               position: gate_position
    AXIOM
               wait_time >= raise_dur + raise_time + up_dur
             & dist_R_to_I / max_speed >= response_time + lower_dur + lower_time + down_dur + raise_dur
             & dist_R_to_I / max_speed >= response_time + lower_dur + lower_time + down_dur + up_dur
    INITIAL
               position = raised
    SCHEDULE
      /* gate will be down before fastest train reaches crossing */
               ( EXISTS s: sensor_id
                           ( s.train_in_R
                           & now - Change ( s.train_in_R )  >= dist_R_to_I / max_speed - response_time )
            ->   position = lowered )
      /* gate will be up after slowest train exits crossing and enough time has elapsed for gate to be raised */
            &  ( FORALL s: sensor_id
                           ( FORALL t: time
                                       ( now - wait_time <= t
                                       & t <= now
                                    ->   ~past ( s.train_in_R, t )  )  )
            ->   position = raised )
    IMPORTED VARIABLE
      /* once a sensor reports a train, it will keep reporting a train at least as long as it takes the
         fastest train to exit the region */
               FORALL s: sensor_id
                        ( Change ( s.train_in_R, now )
                        & ~s.train_in_R
                     ->   0 <= now -  (  ( dist_R_to_I + dist_I_to_out )  / max_speed - response_time )
                        & FORALL t: time
                                    ( now -  (  ( dist_R_to_I + dist_I_to_out )  / max_speed - response_time )  <=
                                      t
                                    & t < now
                                 ->   past ( s.train_in_R, t )  )  )
    TRANSITION lower
      ENTRY           [ TIME : lower_dur ]
                 ~ ( position = lowering
                   |  position = lowered )
               & EXISTS s: sensor_id
                           ( s.train_in_R )
      EXIT
                 position = lowering
    TRANSITION down
      ENTRY           [ TIME : down_dur ]
                 position = lowering
               & now - End ( lower )  >= lower_time
      EXIT
                 position = lowered
    TRANSITION raise
      ENTRY           [ TIME : raise_dur ]
                 ~ ( position = raising
                   |  position = raised )
               & FORALL s: sensor_id
                           ( ~s.train_in_R )
```

```
        EXIT
                position = raising
      TRANSITION up
        ENTRY           [ TIME : up_dur ]
                position = raising
              & now - End ( raise )  >= raise_time
        EXIT
                position = raised
    END Top_Level
  END Gate
END Railroad_Crossing
```

## A.8. Stoplight Control System

```
SPECIFICATION Stoplight
  GLOBAL SPECIFICATION Stoplight
    PROCESSES
              the_controller: Controller,
              the_sensors: array [ 4 ]  of Sensor,
              the_LT_sensors: array [ 4 ]  of Sensor
    TYPE
              pos_real: TYPEDEF r: real ( r > 0 )
    CONSTANT
              min_green, min_yellow, max_wait: pos_real
  END Stoplight
  PROCESS SPECIFICATION Controller
    LEVEL Top_Level
      IMPORT
                pos_real, min_yellow, min_green, max_wait, the_sensors, the_LT_sensors, the_sensors.is_object,
                the_LT_sensors.is_object
      TYPE
                direction: TYPEDEF i: integer ( 1 <= i
                                              & i <= 4 ) ,
                signal:  ( green, yellow, red )
      CONSTANT
                main_dir: direction,
                change_dur: pos_real
      VARIABLE
                circle ( direction ) : signal,
                arrow ( direction ) : signal
      AXIOM
                max_wait >= 3 * min_green + 4 * min_yellow
              & min_green >= change_dur
              & min_yellow >= change_dur
      DEFINE
                adj1 ( d: direction ) : direction ==
                         ( d + 1 )  mod 4,
                opp ( d: direction ) : direction ==
                         ( d + 2 )  mod 4,
                adj2 ( d: direction ) : direction ==
                         ( d + 3 )  mod 4,
                car ( d: direction ) : boolean ==
                        the_sensors [ d ] .is_object,
                LT_car ( d: direction ) : boolean ==
                        the_LT_sensors [ d ] .is_object
      INITIAL
                circle ( main_dir )  = green
              & arrow ( main_dir )  = green
              & FORALL d: direction
                         ( d ~= main_dir
                       ->   circle ( d )  = red )
              & FORALL d: direction
                         ( d ~= main_dir
                       ->   arrow ( d )  = red )
      INVARIANT
        /* there is always some direction that is yellow or green */
                  ( EXISTS d: direction
                             ( circle ( d )  ~= red
                             |  arrow ( d )  ~= red )  )
        /* if a light is green, then all opposing lights are red */
              &  ( FORALL d: direction
                           ( circle ( d )  = green
                         ->   arrow ( opp ( d )  )  = red
                            & circle ( adj1 ( d )  )  = red
                            & circle ( adj2 ( d )  )  = red
                            & arrow ( adj1 ( d )  )  = red
                            & arrow ( adj2 ( d )  )  = red )  )
              &  ( FORALL d: direction
                           ( arrow ( d )  = green
                         ->   circle ( opp ( d )  )  = red
                            & circle ( adj1 ( d )  )  = red
                            & circle ( adj2 ( d )  )  = red
```

```
                    & arrow ( adj1 ( d )  )  = red
                    & arrow ( adj2 ( d )  )  = red )  )
  /* cars must only wait a fixed amount of time before the next green */
        &  ( FORALL d: direction
                    ( now >= max_wait
                    & FORALL t: time
                              ( now - max_wait <= t
                              & t <= now
                           ->   past ( car ( d ) , t )  )
                 ->   EXISTS t: time
                              ( now - max_wait <= t
                              & t <= now
                              & past ( circle ( d ) , t )  = green )  )  )
        &  ( FORALL d: direction
                    ( now >= max_wait
                    & FORALL t: time
                              ( now - max_wait <= t
                              & t <= now
                           ->   past ( LT_car ( d ) , t )  )
                 ->   EXISTS t: time
                              ( now - max_wait <= t
                              & t <= now
                              & past ( arrow ( d ) , t )  = green )  )  )
  /* light will stay green for at least min_green */
        &  ( FORALL d: direction
                    ( Change ( circle ( d ) , now )
                    & circle ( d )  = yellow
                 ->   FORALL t: time
                              ( Change [ 2 ]  ( circle ( d ) , t )
                           ->   t <= now - min_green )  )  )
        &  ( FORALL d: direction
                    ( Change ( arrow ( d ) , now )
                    & arrow ( d )  = yellow
                 ->   FORALL t: time
                              ( Change [ 2 ]  ( arrow ( d ) , t )
                           ->   t <= now - min_green )  )  )
  /* light will stay yellow for at least min_yellow */
        &  ( FORALL d: direction
                    ( Change ( circle ( d ) , now )
                    & circle ( d )  = red
                 ->   FORALL t: time
                              ( Change [ 2 ]  ( circle ( d ) , t )
                           ->   t <= now - min_yellow )  )  )
        &  ( FORALL d: direction
                    ( Change ( arrow ( d ) , now )
                    & arrow ( d )  = red
                 ->   FORALL t: time
                              ( Change [ 2 ]  ( arrow ( d ) , t )
                           ->   t <= now - min_yellow )  )  )
CONSTRAINT
  /* lights change from green to yellow to red to green */
           ( FORALL d: direction
                    (  ( circle ( d )  = yellow
                       & circle' ( d )  ~= circle ( d )
                    ->   circle' ( d )  = green )
                    &  ( circle ( d )  = red
                       & circle' ( d )  ~= circle ( d )
                    ->   circle' ( d )  = yellow )
                    &  ( circle ( d )  = green
                       & circle' ( d )  ~= circle ( d )
                    ->   circle' ( d )  = red )  )  )
        &  ( FORALL d: direction
                    (  ( arrow ( d )  = yellow
                       & arrow' ( d )  ~= arrow ( d )
                    ->   arrow' ( d )  = green )
                    &  ( arrow ( d )  = red
                       & arrow' ( d )  ~= arrow ( d )
                    ->   arrow' ( d )  = yellow )
                    &  ( arrow ( d )  = green
                       & arrow' ( d )  ~= arrow ( d )
                    ->   arrow' ( d )  = red )  )  )
TRANSITION Give_Green_Circle ( d: direction )
  ENTRY           [ TIME : change_dur ]
           car ( d )
         & circle ( d )  = red
         & circle ( opp ( d )  )  ~= yellow
         &  ( arrow ( d )  = yellow
            & now - Change ( arrow ( d )  )  >= min_yellow - change_dur
           |  arrow ( opp ( d )  )  = yellow
            & now - Change ( arrow ( opp ( d )  )  )  >= min_yellow - change_dur )
  EXIT
           circle ( d )  = green
         & Nochange  ( circle ( adj1 ( d )  )  )
         & Nochange  ( circle ( adj2 ( d )  )  )
```

```
        & IF
                  car' ( opp ( d )  )
          THEN
                  circle ( opp ( d )  )  = green
          ELSE
                  Nochange  ( circle ( opp ( d )  )  )
          FI
        & IF
                   ( arrow' ( d )  = yellow )
          THEN
                  arrow ( d )  = red
          ELSE
                  Nochange  ( arrow ( d )  )
          FI
        & arrow ( opp ( d )  )  = red
        & Nochange  ( arrow ( adj1 ( d )  )  )
        & Nochange  ( arrow ( adj2 ( d )  )  )
EXCEPT          [ TIME : change_dur ]
          car ( d )
        & circle ( d )  = red
        &  ( circle ( adj1 ( d )  )  = yellow
           & circle ( adj2 ( d )  )  ~= green
           & arrow ( adj1 ( d )  )  ~= green
           & now - Change ( circle ( adj1 ( d )  )  )  >= min_yellow - change_dur
         |  circle ( adj2 ( d )  )  = yellow
           & circle ( adj1 ( d )  )  ~= green
           & arrow ( adj2 ( d )  )  ~= green
           & now - Change ( circle ( adj2 ( d )  )  )  >= min_yellow - change_dur )
        & ~LT_car ( d )
        & ~LT_car ( opp ( d )  )
EXIT
          circle ( d )  = green
        & circle ( adj1 ( d )  )  = red
        & circle ( adj2 ( d )  )  = red
        & IF
                  car' ( opp ( d )  )
          THEN
                  circle ( opp ( d )  )  = green
          ELSE
                  Nochange  ( circle ( opp ( d )  )  )
          FI
        & arrow ( adj1 ( d )  )  = red
        & arrow ( adj2 ( d )  )  = red
        & Nochange  ( arrow ( d )  )
        & Nochange  ( arrow ( opp ( d )  )  )
EXCEPT          [ TIME : change_dur ]
          car ( d )
        & circle ( d )  = red
        &  ( arrow ( adj1 ( d )  )  = yellow
           & arrow ( adj2 ( d )  )  ~= green
           & circle ( adj1 ( d )  )  = red
           & now - Change ( arrow ( adj1 ( d )  )  )  >= min_yellow - change_dur
         |  arrow ( adj2 ( d )  )  = yellow
           & arrow ( adj1 ( d )  )  ~= green
           & circle ( adj2 ( d )  )  = red
           & now - Change ( arrow ( adj2 ( d )  )  )  >= min_yellow - change_dur )
        & ~car ( adj1 ( d )  )
        & ~car ( adj2 ( d )  )
        & ~LT_car ( d )
        & ~LT_car ( opp ( d )  )
EXIT
          circle ( d )  = green
        & circle ( adj1 ( d )  )  = red
        & circle ( adj2 ( d )  )  = red
        & IF
                  car' ( opp ( d )  )
          THEN
                  circle ( opp ( d )  )  = green
          ELSE
                  Nochange  ( circle ( opp ( d )  )  )
          FI
        & arrow ( adj1 ( d )  )  = red
        & arrow ( adj2 ( d )  )  = red
        & Nochange  ( arrow ( d )  )
        & Nochange  ( arrow ( opp ( d )  )  )
EXCEPT          [ TIME : change_dur ]
          car ( d )
        & circle ( d )  = red
        & circle ( opp ( d )  )  = green
        & arrow ( opp ( d )  )  = red
        & FORALL d2: direction
                    ( ~LT_car ( d2 )  )
        & ~car ( adj1 ( d )  )
        & ~car ( adj2 ( d )  )
EXIT
          circle ( d )  BECOMES green
```

```
  EXCEPT          [ TIME : change_dur ]
            car ( d )
          & circle ( d )  = red
          & arrow ( d )  = green
          & arrow ( opp ( d )  )  = red
          & ~car ( opp ( d )  )
  EXIT
            circle ( d )  BECOMES green
  EXCEPT          [ TIME : change_dur ]
            d = main_dir
          & circle ( d )  ~= red
          & arrow ( d )  ~= red
          &  ( circle ( d )  = yellow
             & now - Change ( circle ( d )  )  >= min_yellow - change_dur
            |  arrow ( d )  = yellow
             & now - Change ( arrow ( d )  )  >= min_yellow - change_dur )
          & FORALL d2: direction
                     (  ( circle ( d2 )  ~= yellow
                     ->   ~car ( d2 )  )
                     &  ( arrow ( d2 )  ~= yellow
                     ->   ~LT_car ( d2 )  )  )
  EXIT
            IF
                     circle' ( d )  = yellow
            THEN
                     circle ( d )  = red
                   & circle ( opp ( d )  )  = red
            ELSE
                     Nochange  ( circle ( d )  )
                   & Nochange  ( circle ( opp ( d )  )  )
            FI
          & Nochange  ( circle ( adj1 ( d )  )  )
          & Nochange  ( circle ( adj2 ( d )  )  )
          & IF
                     arrow' ( d )  = yellow
            THEN
                     IF
                              circle' ( d )  = yellow
                     THEN
                              arrow ( adj1 ( d )  )  = green
                     ELSE
                              Nochange  ( arrow ( adj1 ( d )  )  )
                     FI
                   & arrow ( d )  = red
                   & arrow ( opp ( d )  )  = red
            ELSE
                     Nochange  ( arrow ( d )  )
                   & Nochange  ( arrow ( opp ( d )  )  )
                   & Nochange  ( arrow ( adj1 ( d )  )  )
            FI
          & Nochange  ( arrow ( adj2 ( d )  )  )
TRANSITION Give_Green_Arrow ( d: direction )
  ENTRY           [ TIME : change_dur ]
            LT_car ( d )
          & arrow ( d )  = red
          &  ( circle ( adj1 ( d )  )  = yellow
             & arrow ( adj1 ( d )  )  ~= green
             & circle ( adj2 ( d )  )  ~= green
             & now - Change ( circle ( adj1 ( d )  )  )  >= min_yellow - change_dur
            |  circle ( adj2 ( d )  )  = yellow
             & arrow ( adj2 ( d )  )  ~= green
             & circle ( adj1 ( d )  )  ~= green
             & now - Change ( circle ( adj2 ( d )  )  )  >= min_yellow - change_dur )
  EXIT
            arrow ( d )  = green
          & arrow ( adj1 ( d )  )  = red
          & arrow ( adj2 ( d )  )  = red
          & IF
                     LT_car' ( opp ( d )  )
            THEN
                     arrow ( opp ( d )  )  = green
            ELSE
                     Nochange  ( arrow ( opp ( d )  )  )
            FI
          & circle ( adj1 ( d )  )  = red
          & circle ( adj2 ( d )  )  = red
          & Nochange  ( circle ( opp ( d )  )  )
          & IF
                     car' ( d )
                   & ~LT_car' ( opp ( d )  )
            THEN
                     circle ( d )  = green
            ELSE
                     Nochange  ( circle ( d )  )
            FI
```

```
EXCEPT          [ TIME : change_dur ]
          LT_car ( d )
        & arrow ( d )  = red
        &  ( arrow ( adj1 ( d )  )  = yellow
           & arrow ( adj2 ( d )  )  ~= green
           & circle ( adj1 ( d )  )  = red
           & now - Change ( arrow ( adj1 ( d )  )  )  >= min_yellow - change_dur
          |  arrow ( adj2 ( d )  )  = yellow
           & arrow ( adj1 ( d )  )  ~= green
           & circle ( adj2 ( d )  )  = red
           & now - Change ( arrow ( adj2 ( d )  )  )  >= min_yellow - change_dur )
        & ~car ( adj1 ( d )  )
        & ~car ( adj2 ( d )  )
EXIT
          arrow ( d )  = green
        & arrow ( adj1 ( d )  )  = red
        & arrow ( adj2 ( d )  )  = red
        & IF
                  LT_car' ( opp ( d )  )
          THEN
                  arrow ( opp ( d )  )  = green
          ELSE
                  Nochange  ( arrow ( opp ( d )  )  )
          FI
        & Nochange  ( circle ( adj1 ( d )  )  )
        & Nochange  ( circle ( adj2 ( d )  )  )
        & Nochange  ( circle ( opp ( d )  )  )
        & IF
                  car' ( d )
                & ~LT_car' ( opp ( d )  )
          THEN
                  circle ( d )  = green
          ELSE
                  Nochange  ( circle ( d )  )
          FI
EXCEPT          [ TIME : change_dur ]
          LT_car ( d )
        & arrow ( d )  = red
        & arrow ( opp ( d )  )  ~= yellow
        &  ( circle ( d )  = yellow
           & now - Change ( circle ( d )  )  >= min_yellow - change_dur
          |  circle ( opp ( d )  )  = yellow
           & now - Change ( circle ( opp ( d )  )  )  >= min_yellow - change_dur )
        & ~car ( adj1 ( d )  )
        & ~car ( adj2 ( d )  )
        & ~LT_car ( adj1 ( d )  )
        & ~LT_car ( adj2 ( d )  )
EXIT
          arrow ( d )  = green
        & Nochange  ( circle ( adj1 ( d )  )  )
        & Nochange  ( circle ( adj2 ( d )  )  )
        & IF
                  LT_car' ( opp ( d )  )
          THEN
                  arrow ( opp ( d )  )  = green
          ELSE
                  Nochange  ( arrow ( opp ( d )  )  )
          FI
        & IF
                  circle' ( d )  = yellow
          THEN
                  circle ( d )  = red
          ELSE
                  Nochange  ( circle ( d )  )
          FI
        & circle ( opp ( d )  )  = red
        & Nochange  ( arrow ( adj1 ( d )  )  )
        & Nochange  ( arrow ( adj2 ( d )  )  )
EXCEPT          [ TIME : change_dur ]
          LT_car ( d )
        & arrow ( d )  = red
        & arrow ( opp ( d )  )  = green
        & circle ( opp ( d )  )  = red
        & FORALL d2: direction
                    ( ~car ( d2 )  )
        & ~LT_car ( adj1 ( d )  )
        & ~LT_car ( adj2 ( d )  )
EXIT
          arrow ( d )  BECOMES green
EXCEPT          [ TIME : change_dur ]
          LT_car ( d )
        & arrow ( d )  = red
        & circle ( d )  = green
        & circle ( opp ( d )  )  = red
        & ~car ( opp ( d )  )
        & ~car ( adj1 ( d )  )
```

```
          & ~car ( adj2 ( d )  )
          & ~LT_car ( opp ( d )  )
          & ~LT_car ( adj1 ( d )  )
          & ~LT_car ( adj2 ( d )  )
  EXIT
            arrow ( d )  BECOMES green
  EXCEPT          [ TIME : change_dur ]
            d = main_dir
          &  ( circle ( d )  = red
            |  arrow ( d )  = red )
          & FORALL d2: direction
                    (  ( circle ( d2 )  ~= yellow
                    ->   ~car ( d2 )  )
                    &  ( arrow ( d2 )  ~= yellow
                    ->   ~LT_car ( d2 )  )  )
          & FORALL d2: direction
                    (  ( circle ( d2 )  = yellow
                    ->   now - Change ( circle ( d )  )  >= min_yellow - change_dur )
                    &  ( arrow ( d2 )  = yellow
                    ->   now - Change ( arrow ( d )  )  >= min_yellow - change_dur )  )
  EXIT
            IF
                     ( circle' ( d )  = yellow )
            THEN
                    circle ( d )  = red
            ELSE
                    circle ( d )  = green
            FI
          & FORALL d2: direction
                     ( d2 ~= d
                   ->   circle ( d2 )  = red )
          & IF
                     ( arrow' ( d )  = yellow )
            THEN
                    arrow ( d )  = red
            ELSE
                    arrow ( d )  = green
            FI
          & FORALL d2: direction
                     ( d2 ~= d
                   ->   arrow ( d2 )  = red )
TRANSITION Give_Yellow_Circle ( d: direction )
  ENTRY           [ TIME : change_dur ]
            circle ( d )  = green
          & arrow ( d )  = red
          &  ( car ( adj1 ( d )  )
            |  car ( adj2 ( d )  )
            |  LT_car ( adj1 ( d )  )
            |  LT_car ( adj2 ( d )  )  )
          & now - Change ( circle ( d )  )  >= min_green - change_dur
          &  ( circle ( opp ( d )  )  = green
          ->   now - Change ( circle ( opp ( d )  )  )  >= min_green - change_dur )
  EXIT
            circle ( d )  = yellow
          & Nochange  ( circle ( adj1 ( d )  )  )
          & Nochange  ( circle ( adj2 ( d )  )  )
          & IF
                    circle' ( opp ( d )  )  = green
            THEN
                    circle ( opp ( d )  )  = yellow
            ELSE
                    Nochange  ( circle ( opp ( d )  )  )
            FI
  EXCEPT          [ TIME : change_dur ]
            circle ( d )  = green
          & arrow ( d )  = red
          & LT_car ( opp ( d )  )
          & ~car ( adj1 ( d )  )
          & ~car ( adj2 ( d )  )
          & ~LT_car ( adj1 ( d )  )
          & ~LT_car ( adj2 ( d )  )
          & now - Change ( circle ( d )  )  >= min_green - change_dur
          &  ( circle ( opp ( d )  )  = green
             & LT_car ( d )
          ->   now - Change ( circle ( opp ( d )  )  )  >= min_green - change_dur )
  EXIT
            circle ( d )  = yellow
          & Nochange  ( circle ( adj1 ( d )  )  )
          & Nochange  ( circle ( adj2 ( d )  )  )
          & IF
                    circle' ( opp ( d )  )  = green
                  & LT_car' ( d )
            THEN
                    circle ( opp ( d )  )  = yellow
```

```
          ELSE
                  Nochange  ( circle ( opp ( d )  )  )
          FI
  EXCEPT          [ TIME : change_dur ]
          d ~= main_dir
        & circle ( d )  = green
        & arrow ( d )  = red
        & FORALL d2: direction
                   ( ~car ( d2 )
                   & ~LT_car ( d2 )  )
        & now - Change ( circle ( d )  )  >= min_green - change_dur
        &  ( opp ( d )  ~= main_dir
           & circle ( opp ( d )  )  = green
        ->   now - Change ( circle ( opp ( d )  )  )  >= min_green - change_dur )
  EXIT
          circle ( d )  = yellow
        & Nochange  ( circle ( adj1 ( d )  )  )
        & Nochange  ( circle ( adj2 ( d )  )  )
        & IF
                  opp ( d )  ~= main_dir
                & circle' ( opp ( d )  )  = green
          THEN
                  circle ( opp ( d )  )  = yellow
          ELSE
                  Nochange  ( circle ( opp ( d )  )  )
          FI
  EXCEPT          [ TIME : change_dur ]
          circle ( d )  = green
        & arrow ( d )  = green
        & LT_car ( opp ( d )  )
        & ~car ( opp ( d )  )
        & ~car ( adj1 ( d )  )
        & ~car ( adj2 ( d )  )
        & ~LT_car ( adj1 ( d )  )
        & ~LT_car ( adj2 ( d )  )
        & now - Change ( circle ( d )  )  >= min_green - change_dur
  EXIT
          circle ( d )  BECOMES yellow
TRANSITION Give_Yellow_Arrow ( d: direction )
  ENTRY           [ TIME : change_dur ]
          arrow ( d )  = green
        & circle ( d )  = red
        & ~car ( d )
        & ~car ( opp ( d )  )
        &  ( car ( adj1 ( d )  )
          |  car ( adj2 ( d )  )
          |  LT_car ( adj1 ( d )  )
          |  LT_car ( adj2 ( d )  )  )
        & now - Change ( arrow ( d )  )  >= min_green - change_dur
        &  ( arrow ( opp ( d )  )  = green
        ->   now - Change ( arrow ( opp ( d )  )  )  >= min_green - change_dur )
  EXIT
          arrow ( d )  = yellow
        & Nochange  ( arrow ( adj1 ( d )  )  )
        & Nochange  ( arrow ( adj2 ( d )  )  )
        & IF
                  arrow' ( opp ( d )  )  = green
          THEN
                  arrow ( opp ( d )  )  = yellow
          ELSE
                  Nochange  ( arrow ( opp ( d )  )  )
          FI
  EXCEPT          [ TIME : change_dur ]
          arrow ( d )  = green
        & circle ( d )  = red
        & car ( opp ( d )  )
        & now - Change ( arrow ( d )  )  >= min_green - change_dur
        &  ( arrow ( opp ( d )  )  = green
           & car ( d )
        ->   now - Change ( arrow ( opp ( d )  )  )  >= min_green - change_dur )
  EXIT
          arrow ( d )  = yellow
        & Nochange  ( arrow ( adj1 ( d )  )  )
        & Nochange  ( arrow ( adj2 ( d )  )  )
        & IF
                  arrow' ( opp ( d )  )  = green
                & car' ( d )
          THEN
                  arrow ( opp ( d )  )  = yellow
          ELSE
                  Nochange  ( arrow ( opp ( d )  )  )
          FI
```

```
        EXCEPT          [ TIME : change_dur ]
                  arrow ( d )  = green
                & circle ( d )  = green
                & ~car ( opp ( d )  )
                &  ( car ( adj1 ( d )  )
                  |  car ( adj2 ( d )  )
                  |  LT_car ( adj1 ( d )  )
                  |  LT_car ( adj2 ( d )  )  )
                & now - Change ( arrow ( d )  )  >= min_green - change_dur
                & now - Change ( circle ( d )  )  >= min_green - change_dur
        EXIT
                  arrow ( d )  BECOMES yellow
                & circle ( d )  BECOMES yellow
        EXCEPT          [ TIME : change_dur ]
                  d ~= main_dir
                & arrow ( d )  = green
                & circle ( d )  = red
                & FORALL d2: direction
                           ( ~car ( d2 )
                           & ~LT_car ( d2 )  )
                & now - Change ( arrow ( d )  )  >= min_green - change_dur
                &  ( opp ( d )  ~= main_dir
                   & arrow ( opp ( d )  )  = green
                ->   now - Change ( arrow ( opp ( d )  )  )  >= min_green - change_dur )
        EXIT
                  arrow ( d )  = yellow
                & Nochange  ( arrow ( adj1 ( d )  )  )
                & Nochange  ( arrow ( adj2 ( d )  )  )
                & IF
                           opp ( d )  ~= main_dir
                         & arrow' ( opp ( d )  )  = green
                  THEN
                           arrow ( opp ( d )  )  = yellow
                  ELSE
                           Nochange  ( arrow ( opp ( d )  )  )
                  FI
        EXCEPT          [ TIME : change_dur ]
                  d ~= main_dir
                & arrow ( d )  = green
                & circle ( d )  = green
                & now - Change ( arrow ( d )  )  >= min_green - change_dur
                & now - Change ( circle ( d )  )  >= min_green - change_dur
                & FORALL d2: direction
                           ( ~car ( d2 )
                           & ~LT_car ( d2 )  )
        EXIT
                  arrow ( d )  BECOMES yellow
                & circle ( d )  BECOMES yellow
        EXCEPT          [ TIME : change_dur ]
                  arrow ( d )  = green
                & circle ( d )  = green
                & car ( opp ( d )  )
                & ~car ( adj1 ( d )  )
                & ~car ( adj2 ( d )  )
                & ~LT_car ( adj1 ( d )  )
                & ~LT_car ( adj2 ( d )  )
                & now - Change ( arrow ( d )  )  >= min_green - change_dur
        EXIT
                  arrow ( d )  BECOMES yellow
    END Top_Level
  END Controller
  PROCESS SPECIFICATION Sensor
    LEVEL Top_Level
      IMPORT
                pos_real
      EXPORT
                Arrive, Depart, is_object
      CONSTANT
                sense_dur: pos_real
      VARIABLE
                is_object: boolean
      INITIAL
                ~is_object
      TRANSITION Arrive
        ENTRY           [ TIME : sense_dur ]
                  ~is_object
        EXIT
                  is_object
      TRANSITION Depart
        ENTRY           [ TIME : sense_dur ]
                  is_object
        EXIT
                  ~is_object
    END Top_Level
  END Sensor
END Stoplight
```

# Appendix B

# ASTRAL Undecidability Proof

Define the *untimed quantifier-free Presburger invariance problem* as "given a time independent ASTRAL property P and an ASTRAL specification S that does not use any timed constructs or quantification and is restricted to Presburger arithmetic operations, does P always hold in S?".

**Theorem 1:** The untimed quantifier-free Presburger invariance problem is undecidable.

Proof: Define the *acceptance problem* as "given a Turing machine M and a string w as input, does M accept w?". From [HU 79], the acceptance problem is undecidable. Suppose an algorithm UQPI exists that solves the untimed quantifier-free Presburger invariance problem. From UQPI, it is possible to construct an algorithm ACC that solves the acceptance problem. Without loss of generality, assume that ACC is given a two-counter deterministic Turing machine (2DTM) M with a single accepting state and an input string w. Furthermore, assume that all symbols and states of M are represented as integers. Let n_states be the number of states of M, w_size be the size of the input string, $Q_0$ be the start state of M, $Q_a$ be the accepting state, and D be a set of control tuples of the form (q′, a, c1, c2, q, Δh, Δc1, Δc2) meaning if M is in state q′ with a at the read head, and c1 and c2 at the head of each counter, then M changes to state q, moves its input head Δh, and moves its counter heads Δc1 and Δc2, respectively. Assume that Δh, Δc1, and Δc2 are either -1, 0, or 1 and that the symbol LEFT marks the left end of all three tapes. Algorithm ACC first constructs an ASTRAL specification S as follows:

```
SPECIFICATION   S
   PROCESS SPECIFICATION     P
      TYPE
            state_type: TYPEDEF i: integer (0 ≤ i & i ≤ n_states - 1),
            position_type: TYPEDEF i: integer (0 ≤ i & i ≤ w_size - 1),
            delta_type: TYPEDEF i: integer (-1 ≤ i & i ≤ 1)
      CONSTANT
            n_states, w_size: pos_integer
            Q0, Qa: state_type,
            LEFT, w(position_type): integer
```

VARIABLE
    state: state_type,
    position: position_type,
    counter1, counter2: integer
INITIAL
    state = $Q_0$
  & position = 0
  & counter1 = 0
  & counter2 = 0
TRANSITION T_d
  For each tuple d = (q′, a, c1, c2, q, Δp, Δc1, Δc2) in D, add an exception:

  EXCEPT
      state = q′
    & w(position) = a
    & (c1 = LEFT & counter1 = 0 | c1 ≠ LEFT & counter1 > 0)
    & (c2 = LEFT & counter2 = 0 | c2 ≠ LEFT & counter2 > 0)
  EXIT
      state = q
    & position = position′ + Δp
    & counter1 = counter1′ + Δc1
    & counter2 = counter2′ + Δc2

Note that no timed constructs or quantifiers are used in S and that each expression is specified within the restrictions of Presburger arithmetic. Let P = "state ≠ $Q_a$". ACC then applies the assumed algorithm UQPI to (P, S) and if the answer from UQPI is yes, returns no, otherwise returns yes.

**Lemma 1:** For any finite sequence of length m of TM states initiated from the initial state ($q_0$, $pos_0$, $c1_0$, $c2_0$), ($q_1$, $pos_1$, $c1_1$, $c2_1$), …, ($q_m$, $pos_m$, $c1_m$, $c2_m$), produced by the occurrence of a sequence of control tuples of M, an equivalent sequence of ASTRAL states ($state_0$, $position_0$, $counter1_0$, $counter2_0$), ($state_1$, $position_1$, $counter1_1$, $counter2_1$), …, ($state_m$, $position_m$, $counter1_m$, $counter2_m$) is generated by a sequence of m - 1 T_d transitions in S, where $q_i = state_i$, $pos_i = position_i$, $c1_i = counter1_i$, and $c2_i = counter2_i$.

Proof: Proof by induction on the length of the sequence.

base step: For a sequence of length 1, $q_0 = Q_0$, $pos_0 = 0$, $c1_0 = 0$, $c2_0 = 0$ by definition of M. Similarly, $state_0 = Q_0$, $position_0 = 0$, $counter1_0 = 0$, $counter2_0 = 0$ by the initial state of S. Thus, ($q_0$, $pos_0$, $c1_0$, $c2_0$) = ($state_0$, $position_0$, $counter1_0$, $counter2_0$) so the hypothesis holds for sequences of length 1.

induction step: Assume the hypothesis holds for sequences of length k, thus for the kth TM state in the sequence ($q_{k-1}$, $pos_{k-1}$, $c1_{k-1}$, $c2_{k-1}$), there is an equivalent ASTRAL state ($state_{k-1}$, $position_{k-1}$, $counter1_{k-1}$, $counter2_{k-1}$) produced by a sequence of k - 1 T_d transitions with $q_{k-1} = state_{k-1}$, $pos_{k-1} =$

$position_{k-1}$, $c1_{k-1} = counter1_{k-1}$, and $c2_{k-1} = counter2_{k-1}$. Now suppose the ith control tuple of M (q′, a, c1, c2, q, Δp, Δc1, Δc2) is used to move M into its (k+1)th state ($q_k$, $pos_k$, $c1_k$, $c2_k$). The enabling condition of the ith exception of T_d is defined:

```
EXCEPT
        state = q′
    &   w(position) = a
    &   (c1 = LEFT & counter1 = 0 | c1 ≠ LEFT & counter1 > 0)
    &   (c2 = LEFT & counter2 = 0 | c2 ≠ LEFT & counter2 > 0)
```

The first two conjuncts hold at the kth state in the sequence because $state_{k-1} = q_{k-1} = q'$ and $position_{k-1} = pos_{k-1}$ so $w(position_{k-1}) = w(pos_{k-1}) = a$. Suppose the symbol at position $c1_{k-1}$ of the first counter is LEFT (i.e. c1 = LEFT). By definition of a 2DTM, only the leftmost position of a tape may contain LEFT, thus $c1_{k-1} = counter1_{k-1} = 0$. Suppose the symbol at position $c1_{k-1}$ of the first counter is not LEFT (i.e c1 ≠ LEFT). In this case, $c1_{k-1} > 0$ because by definition of a 2DTM, LEFT may not be overwritten nor passed by the read head. Thus, $c1_{k-1} = counter1_{k-1} > 0$, so the third conjunct of the exception holds. A similar argument can be used to show that the fourth conjunct holds as well, so the ith exception of S is enabled at the kth state of the sequence. Since the exception conditions are mutually exclusive by definition of a 2DTM, no other exception may be enabled after the kth state in the sequence, so the ith exception fires. The exit assertion of the ith exception of T_d is defined:

```
EXIT
        state = q
    &   position = position′ + Δp
    &   counter1 = counter1′ + Δc1
    &   counter2 = counter2′ + Δc2
```

By definition of a control tuple, $q_k = q$, $pos_k = pos_{k-1} + \Delta p$, $c1_k = c1_{k-1} + \Delta c1$, and $c2_k = c2_{k-1} + \Delta c2$. By the semantics of ASTRAL, $state_k = q = q_k$, $position_k = position_{k-1} + \Delta p = pos_{k-1} + \Delta p = pos_k$, $counter1_k = counter1_{k-1} + \Delta c1 = c1_{k-1} + \Delta c1 = c1_k$, $counter2_k = counter2_{k-1} + \Delta c2 = c2_{k-1} + \Delta c2 = c2_k$. Therefore, the induction hypothesis holds. □

**Lemma 2:** ACC answers yes to inputs M and w if and only if M accepts w.

Proof: For the forward direction, suppose ACC returns yes. This means that UQPI returned no and hence P does not always hold in S. Thus, the value of state is $Q_a$ at some time in the execution of S. Suppose M does not accept w. Thus, M uses some sequence of control tuples that does not contain any tuple with $Q_a$ as a new state. By lemma 1, a sequence of transitions of S occurs where each transition directly corresponds to a control tuple used. Therefore, the value of state cannot possibly be $Q_a$, which is a contradiction so M must accept w. For the reverse direction, suppose M accepts w. Then M must reach state $Q_a$ by the occurrence of some finite sequence of control tuples. If that is the

case, then by lemma 1, S also reaches a point at which state = $Q_a$. Thus, P does not always hold so by definition, UQPI returns no and hence ACC returns yes. □

To complete the proof of theorem 1, ACC is an algorithm for the acceptance problem by lemma 2, which is a contradiction because no such algorithm exists by [HU 79]. Thus, UQPI cannot exist, so the untimed quantifier-free Presburger invariance problem is undecidable. □

The proof of theorem 1 relies on the fact that ASTRAL variables can be of arbitrary size, so a TM can be simulated by using variables to hold infinite TM tapes. Unlimited variable size, however, is not the only aspect of ASTRAL that renders invariance problems undecidable. The addition of time to ASTRAL specifications, even with bounded variable domains and a discrete time domain still allows ASTRAL to simulate a TM. The following proof takes advantage of the fact that ASTRAL uses an explicit time variable that must by definition be unbounded and the predicates Start and End, which essentially allow arbitrary time values to be stored. In addition, ASTRAL allows variable time values to be combined with arbitrary arithmetic operations. The essence of the simulation is three transitions that occur cyclically. The first transition, T_d, simulates a control step of the TM. Before the next step can occur, however, the new positions of the counter heads are recorded by preventing the enabling of the other two transitions, T_c1 and T_c2, until the time difference between the end of one transition and the start of the next is the value of the new position.

Define the *discrete-time finite-domain quantifier-free Presburger invariance problem* as "given an ASTRAL property P and an ASTRAL specification S that uses finite variable domains and discrete time, does not use quantification, and is restricted to Presburger arithmetic operations, does P always hold in S?".

**Theorem 2:** The discrete-time finite-domain quantifier-free Presburger invariance problem is undecidable.

Proof: The proof is similar to that of theorem 1, but an algorithm DFQPI is assumed in place of UQPI. In addition, the construction of S is changed to the following:

```
SPECIFICATION   S
   PROCESS SPECIFICATION     P
      TYPE
            next_type: (T_d_next, T_c1_next, T_c2_next),
            state_type: TYPEDEF i: integer (0 ≤ i & i ≤ n_states - 1),
            position_type: TYPEDEF i: integer (0 ≤ i & i ≤ w_size - 1),
            delta_type: TYPEDEF i: integer (-1 ≤ i & i ≤ 1)
```

```
CONSTANT
        n_states, w_size: pos_integer
        Q0, Qa: state_type,
        LEFT, w(position_type): integer
VARIABLE
        next: next_type,
        state: state_type,
        position: position_type,
        delta_c1, delta_c2: delta_type,
        T_d_started, T_c1_started, T_c2_started: boolean
INITIAL
        next = T_d_next
    &   state = Q0
    &   position = 0
    &   delta_c1 = 0
    &   delta_c2 = 0
    &   ~T_d_started
    &   ~T_c1_started
    &   ~T_c2_started
TRANSITION    T_c1
    ENTRY       [TIME:    1]
            next = T_c1_next
        &   now - End(T_d) =
                        IF T_c1_started
                        THEN Start(T_c1) - End2(T_d) + delta_c1
                        ELSE delta_c1
                        FI
    EXIT
            next = T_c2_next
        &   T_c1_started
TRANSITION    T_c2
    ENTRY       [TIME:    1]
            next = T_c2_next
        &   now - End(T_c1) =
                        IF T_c2_started
                        THEN Start(T_c2) - End2(T_c1) + delta_c2
                        ELSE delta_c2
                        FI
    EXIT
            next = T_d_next
        &   T_c2_started
TRANSITION    T_d
    For each tuple d = (q′, a, c1, c2, q, Δp, Δc1, Δc2) in D, add an exception:

    EXCEPT      [TIME:    1]
            next = T_d_next
        &   state = q′
        &   w(position) = a
        &   IF T_d_started
            THEN
                    (    c1 = LEFT & Start(T_c1) - End(T_d) = 0
```

```
                    |   c1 ≠ LEFT & Start(T_c1) - End(T_d) > 0)
                &   (   c2 = LEFT & Start(T_c2) - End(T_c1) = 0
                    |   c2 ≠ LEFT & Start(T_c2) - End(T_c1) > 0)
            ELSE
                    c1 = LEFT
                &   c2 = LEFT
            FI
        EXIT
            next = T_c1_next
        &   state = q
        &   position = position′ + Δp
        &   delta_c1 = Δc1
        &   delta_c2 = Δc2
        &   T_d_started
```

Note that state, position, delta_c1, and delta_c2 are bounded by the number of states in M, the size of the input string w, and two identical domains {-1, 0, 1}, respectively. Also note that no timed constructs or quantifiers are used in S and that each expression is specified within the restrictions of Presburger arithmetic.

**Lemma 3:** For any finite sequence of length m ≥ 2 of TM states initiated from the initial state ($q_0$, $pos_0$, $c1_0$, $c2_0$), ($q_1$, $pos_1$, $c1_1$, $c2_1$), ... ($q_m$, $pos_m$, $c1_m$, $c2_m$), produced by the occurrence of a sequence of control tuples of M, a sequence of m - 1 transition sequences {T_d, T_c1, T_c2} occurs in S and at the ith End(T_c2), $q_i$ = $state_i$, $pos_i$ = $position_i$, $c1_i$ = Start(T_c1) - End(T_d), and $c2_i$ = Start(T_c2) - End(T_c1).

Proof: Proof by induction on the length of the sequence.

base step: For a sequence of length 2, let ($q_1$, $pos_1$, $c1_1$, $c2_1$) be the state of M resulting from the application of some control tuple (q′, a, c1, c2, q, Δp, Δc1, Δc2) to state ($q_0$, $pos_0$, $c1_0$, $c2_0$). By definition of M, $q_0$ = $Q_0$, $pos_0$ = 0, $c1_0$ = 0, and $c2_0$ = 0, so q′ = $Q_0$, a = LEFT, c1 = LEFT, c2 = LEFT, Δc1 ≥ 0, and Δc2 ≥ 0. Furthermore, $q_1$ = q, $pos_1$ = Δp, $c1_1$ = Δc1, and $c2_1$ = Δc2. The initial condition of S states that next = T_d_next so T_d must fire first. Consider the enabling condition of the exception of T_d defined for the control tuple of M that was applied:

```
EXCEPT      [TIME:    1]
        next = T_d_next
    &   state = q′
    &   w(position) = a
    &   IF T_d_started
        THEN
                (   c1 = LEFT & Start(T_c1) - End(T_d) = 0
                |   c1 ≠ LEFT & Start(T_c1) - End(T_d) > 0)
            &   (   c2 = LEFT & Start(T_c2) - End(T_c1) = 0
                |   c2 ≠ LEFT & Start(T_c2) - End(T_c1) > 0)
```

```
ELSE
        c1 = LEFT
    &   c2 = LEFT
FI
```

The first three conjuncts hold by the initial condition of S. No transition has fired at system initialization, so the ELSE case must hold for the exception to be enabled. From the corresponding control tuple, c1 = LEFT and c2 = LEFT, so the exception is enabled. The transition fires immediately because no other transition can be enabled by definition of a 2DTM and mutually exclusive next conditions. Thus, Start(T_d, 0) and End(T_d, 1) hold by the semantics of ASTRAL. From the exit assertion, $state_1 = q = q_1$ and $position_1 = position_0 + \Delta p = pos_0 + \Delta p = pos_1$ will hold later at End(T_c2) if T_c1 and T_c2 fire because only T_d changes state and position. Consider the entry assertion of T_c1:

```
ENTRY       [TIME:    1]
        next = T_c1_next
    &   now - End(T_d) =
                IF T_c1_started
                THEN Start(T_c1) - End2(T_d) + delta_c1
                ELSE delta_c1
                FI
```

Since T_c1 has not occurred previously, the ELSE case is considered. T_c1 will be enabled at now = delta_c1 + 1 = Δc1 + 1. All the transitions in the system are mutually exclusive as previously argued, so Start(T_c1, Δc1 + 1) and End(T_c1, Δc1 + 2) hold by the semantics of ASTRAL. Note that by definition of a 2DTM, the counter positions can never be negative so the earliest T_c1 can start is at End(T_d). By the exit assertion of T_c1, T_c2 is the only transition that can be enabled next. Consider the entry assertion of T_c2:

```
ENTRY       [TIME:    1]
        next = T_c2_next
    &   now - End(T_c1) =
                IF T_c2_started
                THEN Start(T_c2) - End2(T_c1) + delta_c2
                ELSE delta_c2
                FI
```

Since T_c2 has not occurred previously, the ELSE case is considered. T_c2 will be enabled at now = delta_c2 + Δc1 + 2 = Δc2 + Δc1 + 1, so Start(T_c2, Δc2 + Δc1 + 2) and End(T_c2, Δc2 + Δc1 + 3) hold by the semantics of ASTRAL. Note that similar to the T_c1 case, the counter position can never be negative so the earliest T_c2 can start is at End(T_c1). One transition sequence {T_d, T_c1, T_c2} has occurred and at the first End(T_c2), $q_1 = state_1$ and $pos_1 = position_1$ from above.

Start(T_c1) - End(T_d) = (Δc1 + 1) - (1) = Δc1 = $c1_1$ and Start(T_c2) - End(T_c1) = (Δc2 + Δc1 + 2) - (Δc1 + 2) = Δc2 = $c2_1$ so the hypothesis holds for m = 2.

induction step: Assume the hypothesis holds for sequences of length k > 2, thus for the kth TM state in the sequence ($q_{k-1}$, $pos_{k-1}$, $c1_{k-1}$, $c2_{k-1}$), a sequence of k-1 transition sequences {T_d, T_c1, T_c2} have occurred and at the (k-1)th End(T_c2), $q_{k-1}$ = $state_{k-1}$, $pos_{k-1}$ = $position_{k-1}$, $c1_{k-1}$ = Start(T_c1) - End(T_d), and $c2_{k-1}$ = Start(T_c2) - End(T_c1). Let $t_0$ = End(T_c2) at the (k-1)th end of T_c2. Now suppose the ith control tuple of M (q′, a, c1, c2, q, Δp, Δc1, Δc2) is used to move M into its (k+1)th state ($q_k$, $pos_k$, $c1_k$, $c2_k$). The enabling condition of the ith exception of T_d is defined:

```
EXCEPT     [TIME:    1]
        next = T_d_next
    &   state = q′
    &   w(position) = a
    &   IF T_d_started
        THEN
                (    c1 = LEFT & Start(T_c1) - End(T_d) = 0
                |    c1 ≠ LEFT & Start(T_c1) - End(T_d) > 0)
            &   (    c2 = LEFT & Start(T_c2) - End(T_c1) = 0
                |    c2 ≠ LEFT & Start(T_c2) - End(T_c1) > 0)
        ELSE
                c1 = LEFT
            &   c2 = LEFT
        FI
```

The first three conjuncts holds at $t_0$ because $state_{k-1}$ = $q_{k-1}$ = q′ and $position_{k-1}$ = $pos_{k-1}$ by the induction hypothesis so w($position_{k-1}$) = w($pos_{k-1}$) = a. Since m > 2, T_d must have fired so the THEN case is considered. Suppose the symbol at position $c1_{k-1}$ of the first counter is LEFT (i.e. c1 = LEFT). By definition of a 2DTM, only the leftmost position of a tape may contain LEFT, thus $c1_{k-1}$ = Start(T_c1) - End(T_d) = 0. Suppose the symbol at position $c1_{k-1}$ of the first counter is not LEFT (i.e c1 ≠ LEFT). In this case, $c1_{k-1}$ > 0 because by definition of a 2DTM, LEFT may not be overwritten nor passed by the read head. Thus, $c1_{k-1}$ = Start(T_c1) - End(T_d) > 0, so the c1 portion of the THEN case holds. A similar argument can be used to show that the c2 portion holds as well, so the ith exception of S is enabled at $t_0$. Thus, Start(T_d, $t_0$) and End(T_d, $t_0$ + 1) by the semantics of ASTRAL. By reasoning similar to that of the base case, it can be shown that the following hold:

$state_k = q_k$
$position_k = pos_k$
Start(T_c1, $t_0 + 1 + c1_{k-1} + \Delta c1$)
End(T_c1, $t_0 + 2 + c1_{k-1} + \Delta c1$)
Start(T_c2, $t_0 + 2 + c1_{k-1} + \Delta c1 + c2_{k-1} + \Delta c2$)
End(T_c2, $t_0 + 3 + c1_{k-1} + \Delta c1 + c2_{k-1} + \Delta c2$)

Thus, at the (k+1)th End(T_c2), we have:

$$\begin{aligned}
\text{Start}(T_c1) - \text{End}(T_d) &= (t_0 + 1 + c1_{k-1} + \Delta c1) - (t_0 + 1) \\
&= c1_{k-1} + \Delta c1 \\
&= c1_k \\
\text{Start}(T_c2) - \text{End}(T_c1) &= (t_0 + 2 + c1_{k-1} + \Delta c1 + c2_{k-1} + \Delta c2) - (t_0 + 2 + c1_{k-1} + \Delta c1) \\
&= c2_{k-1} + \Delta c2 \\
&= c2_k
\end{aligned}$$

so the induction hypothesis holds. □

To complete the proof of theorem 2, ACC is an algorithm for the acceptance problem by lemma 2 (with lemma 1 replaced by lemma 3 and UQPI replaced by DFQPI in the proof), which is a contradiction because no such algorithm exists by [HU 79]. Thus, DFQPI cannot exist, so the discrete-time finite-domain quantifier-free Presburger invariance problem is undecidable. □

Define the *unrestricted invariance problem* as "given an unrestricted ASTRAL property P and an unrestricted ASTRAL specification S, does P always hold in S?".

**Corollary 1:** The unrestricted invariance problem is undecidable.

Proof: This corollary follows directly from both theorems 1 and 2 since the unrestricted ASTRAL model is a superset of both of the restricted models discussed. □

**Corollary 2:** There is no algorithm to prove an invariant or schedule of an ASTRAL specification.

Proof: Any proof of an invariant or schedule formula must show that the formula holds at all times in the system. From corollary 1, no such algorithm exists. □

# Appendix C

# PVS-Strategies File

```
;; is s1 a substring of s2, initialized with at = strlen(s2) - strlen(s1)
(defun sub-str-aux (s1 s2 at)
   (cond
      ((< at 0) nil)
      ((string= s2 s1 :start1 at :end1 (+ at (length s1))) t)
      (t (sub-str-aux s1 s2 (- at 1)))
   )
)

;; is s1 a substring of s2
(defun sub-str (s1 s2)
   (sub-str-aux s1 s2 (- (length s2) (length s1)))
)

;; is any string of l used in s1
(defun use-str (l s1)
   (cond
      ((null l) nil)
      ((sub-str (car l) s1) t)
      (t (use-str (cdr l) s1))
   )
)

;; are there any expensive expressions in s1
(defun use-bad (s1)
   (or (and (sub-str "ENDIF" s1) (sub-str "Fire_Parms" s1))
      (use-str (list "Change1" "Changen" "Start1" "Startn" "End1" "Endn"
            "Call1" "Calln" "Issued_Call" "UQ" "Mod" "Div") s1))
)

(defstep delete-bad ()
   (let ((dl (gather-fnums (s-forms *goal*) '* nil
               #'(lambda (sf) (use-bad (format nil "~A" (formula sf)))))))
   (delete dl))
   "Deletes antecedents and consequents using Change1, Changen, Start1, Startn,
    End1, Endn, Call1, Calln, Issued_Call, UQ, Mod, Div, and if-then-else
    expressions using Fire_Parms."
   "Deleting expensive antecedents and consequents."
)

(defstep astral-expand (&optional (fnums *))
   (then@
      (expand "extend" fnums)
      (expand "now" fnums)
      (expand "const" fnums)
      (expand "Past" fnums)
      (expand "If_Then_Else" fnums)
      (expand "=" fnums)
      (expand "/=" fnums)
      (expand "Base_Trans" fnums)
      (expand "Exported" fnums)
      (expand "Has_Parms" fnums)
      (expand "Duration" fnums)
      (expand "astral_bool.AND" fnums)
      (expand "astral_bool.OR" fnums)
      (expand "astral_bool.IMPLIES" fnums)
      (expand "astral_bool.IFF" fnums)
      (expand "astral_bool.NOT" fnums)
      (expand "astral_num.-" fnums)
      (expand "astral_num.+" fnums)
      (expand "astral_num.*" fnums)
      (expand "astral_num./" fnums)
      (expand "astral_num.<" fnums)
      (expand "astral_num.>" fnums)
      (auto-rewrite "astral_num.<=" "astral_num.>=")
      (do-rewrite fnums)
      (stop-rewrite "astral_num.<=" "astral_num.>=")
   )
   "Expands basic ASTRAL definitions.

    FNUMS specifies the formulas to be expanded."
   "Expanding basic ASTRAL definitions."
)

(defstep astral-expand-clause (&optional (fnums *))
   (then@
      (expand "Vars_No_Change" fnums)
      (expand "Initial" fnums)
      (expand "i_Initial" fnums)
      (expand "Environment" fnums)
      (expand "i_Environment" fnums)
      (expand "Invariant" fnums)
      (expand "s_Invariant" fnums)
      (expand "i_Invariant" fnums)
      (expand "Schedule" fnums)
      (expand "s_Schedule" fnums)
      (expand "i_Schedule" fnums)
      (expand "Constraint" fnums)
      (expand "s_Constraint" fnums)
      (expand "Imported_Variable" fnums)
      (expand "i_Imported_Variable" fnums)
      (expand "Further_Environment" fnums)
      (expand "Constant_Refinement" fnums)
      (expand "Transition_Selection" fnums)
      (expand "Elgible_Set" fnums)
      (expand "Entry" fnums)
      (expand "Exit" fnums)
      (expand "Enabled_Set" fnums)
      (expand "Enabled" fnums)
      (expand "Entry_No_Parms" fnums)
      (expand "Entry_Parms" fnums)
      (expand "Exit_No_Parms" fnums)
      (expand "Exit_Parms" fnums)
      (expand "EX" fnums)
      (expand "FA" fnums)
      (expand "UQ" fnums)
      (astral-expand fnums)
   )
   "Expands ASTRAL clauses and basic definitions.

    FNUMS specifies the formulas to be expanded."
   "Expanding ASTRAL clauses and basic definitions."
)

(defstep astral-expand-all (&optional (fnums *))
   (then@
      (expand "Id_Type" fnums)
      (expand "Start1" fnums)
      (expand "Startn" fnums)
      (expand "Startn_0" fnums)
      (expand "Call1" fnums)
      (expand "Calln" fnums)
      (expand "Calln_0" fnums)
      (expand "Change1" fnums)
      (expand "Changen" fnums)
      (expand "Changen_0" fnums)
      (expand "Issued_Call" fnums)
      (expand "Var_Changes" fnums)
```

```
        (expand "i_Var_Changes" fnums)
        (expand "Mod" fnums)
        (expand "Div" fnums)
        (astral-expand-clause fnums)
        (expand "End1" fnums)
        (expand "Endn" fnums)
        (expand "Endn_0" fnums)
      )
      "Expands all ASTRAL definitions.

       FNUMS specifies the formulas to be expanded."
      "Expanding all ASTRAL definitions."
)

(defstep my-grind (&optional (if-match NIL))
    (then@
        (astral-expand-clause)
        (repeat (try (skosimp*) (assert) (skip)))
        (delete-bad)
        (grind
            :exclude("Start1" "Startn" "End1" "Endn" "Call1" "Calln"
                "Change1" "Changen" "Issued_Call" "UQ" "Mod" "Div")
            :if-match if-match
        )
    )
    "Expands astral clauses, repeatedly skosimp*/asserts, deletes expensive
     antecedents and consequents, then grinds with given IF-MATCH argument."
    "Performing modified grind."
)

(defstep try-seq-gen ()
    (then@
        (expand "Not_Sequence")
        (skosimp*)
        (expand "Exported")
        (lemma "trans_entry")
        (inst-cp -1 "tr1!1" "t1!1")
        (inst -1 "tr2!1" "t2!1")
        (lemma "trans_exit")
        (inst -1 "tr1!1" "t1!1 + Duration(tr1!1)")
        (lemma "vars_no_change")
        (inst -1 "t1!1 + Duration(tr1!1)" "t2!1")
        (replace -5)
        (replace -6)
        (delete (-5 -6))
        (spread (split -1) (
            (then@
                (inst -1 "t2!1")
                (delete -9)
                (my-grind T)
            )
            (assert)
            (assert)
        ))
    )
    "Attempts sequence generator obligation for transitions that are not
     parameterized."
    "Attempting sequence generator obligation for unparameterized transitions."
)

(defstep try-seq-gen-p (parm1 parm2)
    (then@
        (expand "Not_Sequence")
        (skosimp*)
        (expand "Exported")
        (lemma "trans_entry")
        (inst-cp -1 "tr1!1" "t1!1")
        (inst -1 "tr2!1" "t2!1")
        (lemma "trans_exit")
        (inst -1 "tr1!1" "t1!1 + Duration(tr1!1)")
        (lemma "vars_no_change")
        (inst -1 "t1!1 + Duration(tr1!1)" "t2!1")
        (spread (split -1) (
            (then@
                (inst -1 "t2!1")
                (delete -11)
                (replace -5)
                (replace -6)
                (astral-expand-clause)
                (assert)
                (repeat (try (skosimp*) (assert) (skip)))
                (if parm1
                    (spread (name "FP1" "Fire_Parms(Base_Trans(tr1!1), t1!1)") (
                        (then@
                            (let ((fnum (- 0 (- (length (gather-seq
                                (s-forms *goal*) '- nil #'(lambda (sf) T))) 5)))
                            ) (replace fnum))
                            (expand "Base_Trans" -1)
                            (replace -1)
                            (let ((fp1 (format nil "~A(FP1)" parm1))
                            ) (repeat (inst? * ("V1" fp1))))
                        )
                        (then@
                            (inst 1 "tr1!1")
                            (expand* "Base_Trans" "Exported" "Has_Parms")
                        )
                        (expand* "Base_Trans" "Exported" "Has_Parms")
                    ))
                    (skip)
                )
                (if parm2
                    (spread (name "FP2" "Fire_Parms(Base_Trans(tr2!1), t2!1)") (
                        (then@
                            (let ((fnum (- 0 (- (length (gather-seq
                                (s-forms *goal*) '- nil #'(lambda (sf) T))) 4)))
                            ) (replace fnum))
                            (expand "Base_Trans" -1)
                            (replace -1)
                            (let ((fnum (- 0 (length (gather-seq
                                (s-forms *goal*) '- nil #'(lambda (sf) T)))))
                            ) (replace fnum))
                            (let ((fp2 (format nil "~A(FP2)" parm2))
                            ) (repeat (inst? * ("V1" fp2))))
                        )
                        (then@
                            (inst 1 "tr2!1")
                            (expand* "Base_Trans" "Exported" "Has_Parms")
                        )
                        (expand* "Base_Trans" "Exported" "Has_Parms")
                    ))
                    (skip)
                )
                (my-grind T)
            )
            (assert)
            (assert)
        ))
    )
    "Attempts sequence generator obligation for transitions that are
     parameterized.

     PARM1 is the parameter type of the first transition.
     PARM2 is the parameter type of the second transition.

     If the two parameter types are the same, only PARM1 should be given."
    "Attempting sequence generator obligation for parameterized transitions."
)

(defstep try-seq-gen-0 ()
    (then@
        (expand "Not_Initial")
        (skosimp*)
        (lemma "vars_no_change")
        (inst -1 "0" "t2!1")
        (spread (split -1) (
            (then@
                (inst -1 "t2!1")
                (lemma "trans_entry")
                (inst -1 "tr2!1" "t2!1")
                (replace -4)
                (delete -4)
                (lemma "initial_state")
                (my-grind T)
            )
            (assert)
            (then@
                (skosimp*)
                (inst -6 "tr2!2" "t2!2")
                (assert)
            )
        ))
    )
    "Attempts initial case sequence generator obligation for transitions that
     are not parameterized."
    "Attempting initial case sequence generator obligation for unparameterized
     transitions."
```

```
)

(defstep try-seq-gen-0-p (parm)
   (then@
      (expand "Not_Initial")
      (skosimp*)
      (lemma "vars_no_change")
      (inst -1 "0" "t2!1")
      (spread (split -1) (
         (then@
            (inst -1 "t2!1")
            (delete -4)
            (lemma "trans_entry")
            (inst -1 "tr2!1" "t2!1")
            (replace -3)
            (lemma "initial_state")
            (astral-expand-clause)
            (assert)
            (repeat (try (skosimp*) (assert) (skip)))
            (spread (name "FP" "Fire_Parms(Base_Trans(tr2!1), t2!1)") (
               (then@
                  (let ((fnum (- 0 (- (length (gather-seq
                     (s-forms *goal*) '- nil #'(lambda (sf) T))) 1)))
                  ) (replace fnum))
                  (expand "Base_Trans" -1)
                  (replace -1)
                  (let ((fp (format nil "~A(FP)" parm))
                  ) (repeat (inst? * ("V1" fp))))
                  (my-grind T)
               )
               (then@
                  (inst 1 "tr2!1")
                  (expand* "Base_Trans" "Exported" "Has_Parms")
               )
               (expand* "Base_Trans" "Exported" "Has_Parms")
            ))
         )
         (assert)
         (then@
            (skosimp*)
            (inst -6 "tr2!2" "t2!2")
            (assert)
         )
      ))
   )
   "Attempts initial case sequence generator obligation for transitions that
    are parameterized.

    PARM is the parameter type of the transition."
   "Attempting initial case sequence generator obligation for parameterized
    transitions."
)

(defstep try-base-case ()
   (then@
      (lemma "initial_state")
      (try (try (grind) (fail) (skip)) (skip) (skip))
   )
   "Attempts proof of invariant/schedule base case."
   "Attempting proof of invariant/schedule base case."
)

(defstep try-untimed (fnum_i fnum_d &optional (do_grind T))
   (spread (case "Vars_No_Change(T0, T1!1)") (
      (then@
         (let ((fnum1 (- fnum_i 1)))
            (inst fnum1 "T0"))
         (let ((fnum1 (- fnum_d 1)))
            (delete fnum1))
         (try (try (my-grind) (fail) (skip)) (skip) (skip))
      )
      (then@
         (lemma "not_vnc_vc")
         (inst -1 "T0" "T1!1")
         (assert)
         (skolem! -1)
         (flatten -1)
         (lemma "var_changes")
         (inst -1 "t1!1")
         (assert)
         (delete (1 -4))
         (inst -4 "t1!1")
         (skolem! -1)
         (flatten -1)
         (let ((fnum5 (- fnum_i 5)))
            (inst fnum5 "t1!1 - Duration(tr1!1)"))
         (let ((fnum5 (- fnum_d 5)))
            (inst fnum5 "tr1!1"))
         (lemma "trans_entry")
         (inst -1 "tr1!1" "t1!1 - Duration(tr1!1)")
         (lemma "trans_exit")
         (inst -1 "tr1!1" "t1!1")
         (spread (case-trans "tr1!1") (
            (branch (split -1) (
               (then@
                  (replace -1)
                  (delete -1)
                  (if do_grind
                     (try (try (my-grind) (fail) (skip))
                        (skip) (skip))
                     (skip)
                  )
               )
            ))
            (then@ (flatten) (assert))
         ))
      )
   ))
   "Attempts proof of untimed invariant/schedule induction case.

    DO_GRIND specifies whether to grind each transition case."
   "Attempting proof of untimed invariant/schedule induction case."
)

(defstep try-untimed-con (&optional (do_grind T))
   (then@
      (skosimp*)
      (expand "s_Constraint")
      (skosimp*)
      (inst-cp -1 "T1!1 - Duration(TR1!1)")
      (inst -1 "T1!1")
      (lemma "trans_entry")
      (inst -1 "TR1!1" "T1!1 - Duration(TR1!1)")
      (lemma "trans_exit")
      (inst -1 "TR1!1" "T1!1")
      (spread (case-trans "TR1!1") (
         (branch (split -1) (
            (then@
               (replace -1)
               (delete -1)
               (if do_grind
                  (try (try (my-grind) (fail) (skip))
                     (skip) (skip))
                  (skip)
               )
            )
         ))
         (then@ (flatten) (assert))
      ))
   )
   "Attempts proof of untimed constraint.

    DO_GRIND specifies whether to grind each transition case."
   "Attempting proof of untimed constraint."
)

(defstep step-bw-indeterminate (t_from
      &optional (fnum_i NIL) (fnum_d NIL) (do_grind T))
   (let (
      (sc (format nil "~A" (collect-skolem-constants)))
      (i (+ (loop for i upfrom 1
            while (or (search (format nil "tr1!~A" i) sc)
                  (search (format nil "t1!~A" i) sc))
            count i) 1))
      (tr1 (format nil "tr1!~A" i))
      (t1 (format nil "t1!~A" i))
      (t1_d (format nil "~A + Duration(~A)" t1 tr1))
      (cstr (format nil "EXISTS (TR1: transition, T1: time):
         T1 + Duration(TR1) <= ~A AND Fired(TR1, T1)" t_from))
   ) (spread (case cstr) (
      (then@
         (lemma "ended_last_ended")
         (inst -1 t_from)
         (split -1)
         (delete -2)
         (skolem -1 (tr1 t1))
```

```
        (if fnum_d
            (let ((fnum1 (- fnum_d 1)))
                (inst fnum1 tr1))
            (skip)
        )
        (if fnum_i
            (let ((fnum1 (- fnum_i 1)))
                (then@
                    (inst-cp fnum1 t1)
                    (inst fnum1 t1_d)
                )
            )
            (skip)
        )
        (lemma "vars_no_change")
        (inst -1 t1_d t_from)
        (flatten)
        (assert)
        (split -1)
        (inst -1 t_from)
        (lemma "trans_entry")
        (inst -1 tr1 t1)
        (lemma "trans_exit")
        (inst -1 tr1 t1_d)
        (spread (case-trans tr1) (
            (branch (split -1) (
                (then@
                    (replace -1)
                    (delete -1)
                    (if do_grind
                        (try (try (my-grind) (fail) (skip))
                            (skip) (skip))
                        (skip)
                    )
                )
            ))
            (then@ (flatten) (assert))
        ))
    )
    (then@
        (if fnum_i
            (inst fnum_i 0)
            (skip)
        )
        (lemma "initial_state")
        (lemma "vars_no_change")
        (inst -1 0 t_from)
        (assert)
        (spread (split -1) (
            (then@
                (inst -1 t_from)
                (try (try (my-grind) (fail) (skip)) (skip) (skip))
            )
            (then@
                (skolem 1 (tr1 t1))
                (inst 2 tr1 t1)
                (flatten)
                (assert)
            )
        ))
    )
)))
    "Steps backward indeterminately from T_FROM to the last transition to fire.

    FNUM_I is the formula number of a quantified inductive assumption.
    FNUM_D is the formula number of a quantified DELTA expression.
    DO_GRIND specifies whether to grind each transition case."
    "Stepping backward indeterminately to last transition."
)

(defstep step-fw-delay (tr_from t_from tr_to t_to
        &optional (add_entry NIL) (fnum_i NIL) (do_grind T))
    (let (
        (sc (format nil "~A" (collect-skolem-constants)))
        (i (+ (loop for i upfrom 1
                while (or (search (format nil "tr1!~A" i) sc)
                        (search (format nil "t1!~A" i) sc))
                count i) 1))
        (tr1 (format nil "tr1!~A" i))
        (t1 (format nil "t1!~A" i))
        (t_from_d (format nil "~A + Duration(~A)" t_from tr_from))
        (t_delay (format nil "~A - (~A) - Duration(~A)" t_to t_from tr_from))
        (cstr_delay (format nil "~A > 0" t_delay))
        (cstr_tr_to (format nil "~A = ~A" tr1 tr_to))
        (cstr_t1 (format nil "~A >= ~A" t1 t_from_d))
    ) (spread (case cstr_delay) (
        (then@
            (assert)
            (if fnum_i
                (let ((fnum1 (- fnum_i 1)))
                    (then@
                        (inst-cp fnum1 t_from)
                        (inst fnum1 t_from_d)
                    )
                )
                (skip)
            )
            (if add_entry
                (then@
                    (lemma "trans_entry")
                    (inst -1 tr_from t_from)
                    (lemma "trans_exit")
                    (inst -1 tr_from t_from_d)
                    (astral-expand -1)
                    (assert)
                )
                (skip)
            )
            (lemma "no_trans_fire_vnc_lt")
            (inst -1 t_from_d t_to)
            (assert)
            (spread (split -1) (
                (then@
                    (flatten)
                    (inst -2 t_to)
                    (lemma "trans_fire")
                    (inst -1 t_to)
                    (spread (split -1) (
                        (then@
                            (skolem -1 (tr1))
                            (lemma "trans_entry")
                            (inst -1 tr1 t_to)
                            (spread (case cstr_tr_to) (
                                (then@
                                    (replace -1)
                                    (delete -1)
;; main line of proof continues
                                )
                                (spread (case-trans tr1) (
                                    (branch (split -1) (
                                        (then@
                                            (replace -1)
                                            (delete -1)
                                            (if do_grind
                                                (try (try (my-grind)
                                                    (fail) (skip))
                                                    (skip) (skip))
                                                (skip)
;; user must show that transitions besides tr_to are not enabled at t_to
                                            )
                                        )
                                    ))
                                    (then@ (flatten) (assert))
                                ))
                            ))
                        )
                        (then@
                            (inst 1 tr_to)
                            (flatten)
                            (assert)
;; user must show that tr_to is enabled at t_to
                        )
                        (then@
                            (skolem 1 (tr1 t1))
                            (inst -1 tr1 t1)
                            (flatten)
                            (spread (case cstr_t1) (
                                (assert)
                                (then@
                                    (lemma "trans_mutex_end")
                                    (inst -1 tr1 t1)
                                    (assert)
                                    (inst -1 tr_from t_from)
                                    (assert)
                                )
                            ))
```

```
                    )
                ))
            )
            (then@
                (inst 1 tr_from)
                (assert)
            )
            (then@
                (skolem 1 (tr1 t1))
                (flatten 1)
                (inst -3 t1)
                (lemma "trans_entry")
                (inst -1 tr1 t1)
                (spread (case-trans tr1) (
                    (branch (split -1) (
                        (then@
                            (replace -1)
                            (delete -1)
                            (if do_grind
                                (try (try (my-grind) (fail) (skip))
                                    (skip) (skip))
                                (skip)
                            )
;; user must show that transitions are not enabled before t_to
                        )
                    ))
                    (then@ (flatten) (assert))
                ))
            )
        ))
    )
;; could finish this by introducing all typepreds
    (assert)
))
"Steps forward from TR_FROM, which fired at T_FROM, to TR_TO, which is
 to fire at T_TO after a delay.

 ADD_ENTRY specifies whether to assert entry assertion of TR_FROM.
 FNUM_I is the formula number of a quantified inductive assumption.
 DO_GRIND specifies whether to grind each transition case."
"Stepping forward with delay to next transition."
)

(defstep step-fw-immediate (tr_from t_from tr_to
        &optional (add_entry NIL) (fnum_i NIL) (do_grind T))
    (let (
        (sc (format nil "~A" (collect-skolem-constants)))
        (i (+ (loop for i upfrom 1
                while (or (search (format nil "tr1!~A" i) sc)
                          (search (format nil "t1!~A" i) sc))
                count i) 1))
        (tr1 (format nil "tr1!~A" i))
        (t1 (format nil "t1!~A" i))
        (t_from_d (format nil "~A + Duration(~A)" t_from tr_from))
        (cstr_tr_to (format nil "~A = ~A" tr1 tr_to))
    ) (then@
        (if fnum_i
            (inst fnum_i t_from)
            (skip)
        )
        (if add_entry
            (then@
                (lemma "trans_entry")
                (inst -1 tr_from t_from)
                (lemma "trans_exit")
                (inst -1 tr_from t_from_d)
                (astral-expand -1)
                (assert)
            )
            (skip)
        )
        (lemma "trans_fire")
        (inst -1 t_from_d)
        (spread (split -1) (
            (then@
                (skolem -1 (tr1))
                (lemma "trans_entry")
                (inst -1 tr1 t_from_d)
                (spread (case cstr_tr_to) (
                    (then@
                        (replace -1)
                        (delete -1)
;; main line of proof continues
                    )
                    (spread (case-trans tr1) (
                        (branch (split -1) (
                            (then@
                                (replace -1)
                                (delete -1)
                                (if do_grind
                                    (try (try (my-grind) (fail) (skip))
                                        (skip) (skip))
                                    (skip)
                                )
                            )
                        ))
                        (then@ (flatten) (assert))
                    ))
                ))
            )
            (then@
                (inst 1 tr_to)
                (try (try (my-grind) (fail) (skip)) (skip) (skip))
            )
            (then@
                (skolem 1 (tr1 t1))
                (flatten)
                (lemma "trans_mutex_end")
                (inst -1 tr1 t1)
                (assert)
                (inst -1 tr_from t_from)
                (assert)
            )
        ))
    ))
    "Steps forward from TR_FROM, which fired at T_FROM, to TR_TO, which is
     to fire immediately at T_FROM + Duration(TR_FROM).

     ADD_ENTRY specifies whether to assert the entry assertion of TR_FROM.
     FNUM_I is the formula number of a quantified inductive assumption.
     DO_GRIND specifies whether to grind each transition case."
    "Stepping forward immediately to next transition."
)

(defstep step-bw-delay (tr_from t_from tr_to t_to
        &optional (add_entry NIL) (fnum_i NIL) (do_grind T))
    (then@ (if add_entry
        (then@
            (lemma "trans_entry")
            (inst -1 tr_from t_from)
            (astral-expand -1)
            (assert)
        )
        (skip)
    )
    (let (
        (sc (format nil "~A" (collect-skolem-constants)))
        (i (+ (loop for i upfrom 1
                while (or (search (format nil "tr1!~A" i) sc)
                          (search (format nil "t1!~A" i) sc))
                count i) 1))
        (tr1 (format nil "tr1!~A" i))
        (t1 (format nil "t1!~A" i))
        (t1_d (format nil "~A + Duration(~A)" t1 tr1))
        (cstr (format nil "EXISTS (TR1: transition, T1: time):
            T1 + Duration(TR1) <= ~A AND Fired(TR1, T1)" t_from))
    ) (spread (case cstr) (
        (then@
            (lemma "ended_last_ended")
            (inst -1 t_from)
            (split -1)
            (delete -2)
            (skolem -1 (tr1 t1))
            (lemma "vars_no_change")
            (inst -1 t1_d t_from)
            (flatten)
            (assert)
            (split -1)
            (inst -1 t_from)
            (lemma "trans_entry")
            (inst -1 tr1 t1)
            (lemma "trans_exit")
            (inst -1 tr1 t1_d)
            (let (
                (sc2 (format nil "~A" (collect-skolem-constants)))
                (i2 (+ (loop for i2 upfrom 1
```

```
                    while (or (search (format nil "tr1!~A" i2) sc2)
                              (search (format nil "t1!~A" i2) sc2))
                            count i2) 1))
                (tr2 (format nil "tr1!~A" i2))
                (t2 (format nil "t1!~A" i2))
                (cstr_tr_to (format nil "~A = ~A" tr1 tr_to))
                (cstr_t_to (format nil "~A = ~A" t1 t_to))
                (cstr_t1_gte (format nil "~A >= ~A" t1 t_to))
                (cstr_t1_lte (format nil "~A <= ~A" t1 t_to))
                (t_delay (format nil "~A - (~A) - Duration(~A)"
                   t_from t_to tr_to))
                (t_fire (format nil "~A - (~A) + ~A" t_from t_to t1))
                (cstr_delay (format nil "~A > 0" t_delay))
                (cstr_t_f (format nil "~A > 0" t_fire))
            ) (spread (case cstr_tr_to) (
                (then@
                    (replace -1)
                    (delete -1)
                    (spread (case cstr_t_to) (
                        (then@
                            (typepred t1)
                            (replace -2)
                            (delete -2)
;; main line of proof continues
                        )
                        (spread (case cstr_t1_lte) (
                            (spread (case cstr_t1_gte) (
                                (assert)
                                (spread (case cstr_delay) (
                                    (spread (case cstr_t_f) (
                                        (then@
                                            (typepred t1)
                                            (assert)
                                            (lemma "trans_fire")
                                            (inst -1 t_fire)
                                            (spread (split -1) (
                                                (then@
                                                    (skolem -1 (tr2))
                                                    (inst -12 tr2 t_fire)
                                                    (spread (split -12) (
                                                        (assert)
                                                        (then@
                                                            (lemma "trans_mutex")
                                                            (inst -1 tr_to t1)
                                                            (assert)
                                                            (flatten)
                                                            (inst -2 tr2 t_fire)
                                                            (assert)
                                                        )
                                                        (then@
                                                            (lemma "trans_mutex")
                                                            (inst -1 tr2 t_fire)
                                                            (assert)
                                                            (flatten)
                                                            (inst -2 tr_from t_from)
                                                            (assert)
                                                        )
                                                    ))
                                                )
                                                (then@
                                                    (inst 1 tr_from)
                                                    (flatten)
                                                    (assert)
;; user must show that tr_from is enabled earlier if tr_to fires early
                                                )
                                                (then@
                                                    (skolem 1 (tr2 t2))
                                                    (inst -10 tr2 t2)
                                                    (spread (split -11) (
                                                        (assert)
                                                        (then@
                                                            (flatten)
                                                            (assert)
                                                        )
                                                        (then@
                                                            (flatten)
                                                            (lemma "trans_mutex")
                                                            (inst -1 tr2 t2)
                                                            (assert)
                                                            (flatten)
                                                            (inst -2 tr_from t_from)
                                                            (assert)
                                                        )
                                                        ))
                                                    )
                                                    (then@
                                                        (skolem 1 (tr1 t1))
                                                        (inst 2 tr1 t1)
                                                        (flatten)
                                                        (assert)
                                                    )
                                                ))
                                            )
                                            (assert)
                                        ))
                                    ))
                                ))
                            ))
                        ))
                    ))
                )
                (spread (case-trans tr1) (
                    (then@
                        (if fnum_i
                            (if add_entry
                                (let ((fnum8 (- fnum_i 8)))
                                    (then@
                                        (inst-cp fnum8 t1)
                                        (inst fnum8 t1_d)
                                    )
                                )
                                (let ((fnum7 (- fnum_i 7)))
                                    (then@
                                        (inst-cp fnum7 t1)
                                        (inst fnum7 t1_d)
                                    )
                                )
                            )
                            (skip)
                        )
                        (branch (split -1) (
                            (then@
                                (replace -1)
                                (delete -1)
                                (if do_grind
                                    (try (try (my-grind) (fail) (skip))
                                        (skip) (skip))
                                    (skip)
;; user must show that transitions besides tr_to do not end at t_to
                                )
                            )
                        ))
                    )
                    (then@ (flatten) (assert))
                ))
            )))
        )
        (then@
            (lemma "initial_state")
            (lemma "vars_no_change")
            (inst -1 0 t_from)
            (assert)
            (spread (split -1) (
                (then@
                    (inst -1 t_from)
                    (delete 1)
                    (try (try (my-grind) (fail) (skip)) (skip) (skip))
                )
                (then@
                    (skolem 1 (tr1 t1))
                    (inst 2 tr1 t1)
                    (flatten)
                    (assert)
                )
            ))
        )
    ))))
    "Steps backward from TR_FROM, which fired at T_FROM, to TR_TO, which is
     to end at T_TO after a delay.

     ADD_ENTRY specifies whether to assert the entry assertion of TR_FROM.
     FNUM_I is the formula number of a quantified inductive assumption.
     DO_GRIND specifies whether to grind each transition case."
    "Stepping backward with delay to last transition."
)
```

```
(defstep step-bw-immediate (tr_from t_from tr_to
        &optional (add_entry NIL) (fnum_i NIL) (do_grind T))
    (then@ (if add_entry
        (then@
            (lemma "trans_entry")
            (inst -1 tr_from t_from)
            (astral-expand -1)
            (assert)
        )
        (skip)
    )
    (let (
        (sc (format nil "~A" (collect-skolem-constants)))
        (i (+ (loop for i upfrom 1
                while (or (search (format nil "tr1!~A" i) sc)
                          (search (format nil "t1!~A" i) sc))
                count i) 1))
        (tr1 (format nil "tr1!~A" i))
        (t1 (format nil "t1!~A" i))
        (t1_d (format nil "~A + Duration(~A)" t1 tr1))
        (t_to (format nil "~A - Duration(~A)" t_from tr_to))
        (cstr (format nil "EXISTS (TR1: transition, T1: time):
            T1 + Duration(TR1) <= ~A AND Fired(TR1, T1)" t_from))
    ) (spread (case cstr) (
        (then@
            (lemma "ended_last_ended")
            (inst -1 t_from)
            (split -1)
            (delete -2)
            (skolem -1 (tr1 t1))
            (flatten -1)
            (lemma "vars_no_change")
            (inst -1 t1_d t_from)
            (assert)
            (split -1)
            (inst -1 t_from)
            (lemma "trans_entry")
            (inst -1 tr1 t1)
            (lemma "trans_exit")
            (inst -1 tr1 t1_d)
            (let (
                (sc2 (format nil "~A" (collect-skolem-constants)))
                (i2 (+ (loop for i2 upfrom 1
                        while (or (search (format nil "tr1!~A" i2) sc2)
                                  (search (format nil "t1!~A" i2) sc2))
                        count i2) 1))
                (tr2 (format nil "tr1!~A" i2))
                (t2 (format nil "t1!~A" i2))
                (cstr_tr_to (format nil "~A = ~A" tr1 tr_to))
                (cstr_t_to (format nil "~A = ~A" t1 t_to))
                (cstr_t1_gte (format nil "~A >= ~A" t1 t_to))
                (cstr_t1_lte (format nil "~A <= ~A" t1 t_to))
                (t_fire (format nil "~A + Duration(~A)" t1 tr_to))
                (cstr_tf (format nil "~A + Duration(~A) <= ~A"
                    t_fire tr2 t_from))
                (cstr_tr2_1 (format nil "~A + Duration(~A) <= ~A"
                    t2 tr2 t_from))
                (cstr_tr2_2 (format nil "~A + Duration(~A) < ~A + Duration(~A)"
                    t1 tr_to t2 tr2))
            ) (spread (case cstr_tr_to) (
                (then@
                    (replace -1)
                    (delete -1)
                    (spread (case cstr_t_to) (
                        (then@
                            (typepred t1)
                            (replace -2)
                            (delete -2)
;; main line of proof continues
                        )
                        (spread (case cstr_t1_lte) (
                            (spread (case cstr_t1_gte) (
                                (assert)
                                (then@
                                    (lemma "trans_fire")
                                    (inst -1 t_fire)
                                    (spread (split -1) (
                                        (then@
                                            (skolem -1 (tr2))
                                            (spread (case cstr_tf) (
                                                (then@
                                                    (inst -9 tr2 t_fire)
                                                    (assert)
                                                )
                                                (then@
                                                    (lemma "trans_mutex")
                                                    (inst -1 tr2 t_fire)
                                                    (assert)
                                                    (flatten)
                                                    (inst -2 tr_from t_from)
                                                    (assert)
                                                )
                                            ))
                                        )
                                        (then@
                                            (inst 1 tr_from)
                                            (try (try (my-grind) (fail) (skip))
                                                (skip) (skip))
                                        )
                                        (then@
                                            (skolem 1 (tr2 t2))
                                            (spread (case cstr_tr2_1) (
                                                (spread (case cstr_tr2_2) (
                                                    (then@
                                                        (inst -10 tr2 t2)
                                                        (flatten)
                                                        (assert)
                                                    )
                                                    (then@
                                                        (flatten)
                                                        (lemma "trans_mutex_end")
                                                        (inst -1 tr2 t2)
                                                        (assert)
                                                        (inst -1 tr_to t1)
                                                        (assert)
                                                    )
                                                ))
                                                (then@
                                                    (flatten)
                                                    (lemma "trans_mutex")
                                                    (inst -1 tr2 t2)
                                                    (assert)
                                                    (flatten)
                                                    (inst -2 tr_from t_from)
                                                    (assert)
                                                )
                                            ))
                                        )
                                    ))
                                )
                            ))
                        )
                    ))
                )
                (assert)
            ))
        ))
    )
    (spread (case-trans tr1) (
        (then@
            (if fnum_i
                (if add_entry
                    (let ((fnum8 (- fnum_i 8)))
                        (then@
                            (inst-cp fnum8 t1)
                            (inst fnum8 t1_d)
                        )
                    )
                    (let ((fnum7 (- fnum_i 7)))
                        (then@
                            (inst-cp fnum7 t1)
                            (inst fnum7 t1_d)
                        )
                    )
                )
                (skip)
            )
            (branch (split -1) (
                (then@
                    (replace -1)
                    (delete -1)
                    (if do_grind
                        (try (try (my-grind) (fail) (skip))
                            (skip) (skip))
                        (skip)
                    )
                )
            ))
        )
```

```
                (then@ (flatten) (assert))
              ))
            )))
        )
        (then@
          (lemma "initial_state")
          (lemma "vars_no_change")
          (inst -1 0 t_from)
          (assert)
          (spread (split -1) (
            (then@
              (inst -1 t_from)
              (delete 1)
              (try (try (my-grind) (fail) (skip)) (skip) (skip))
            )
            (then@
              (skolem 1 (tr1 t1))
              (inst 2 tr1 t1)
              (flatten)
              (assert)
            )
          ))
        )
      ))))
      "Steps backward from TR_FROM, which fired at T_FROM, to TR_TO, which is
       to end immediately at T_FROM.

       ADD_ENTRY specifies whether to assert the entry assertion of TR_FROM.
       FNUM_I is the formula number of a quantified inductive assumption.
       DO_GRIND specifies whether to grind each transition case."
      "Stepping backward immediately to last transition."
)

(defstep change-fire (fnum_c vname_c time_c &optional (do_grind T))
    (let (
        (sc (format nil "~A" (collect-skolem-constants)))
        (i (+ (loop for i upfrom 1
                while (or (search (format nil "tr1!~A" i) sc)
                          (search (format nil "t1!~A" i) sc))
                count i) 1))
        (tr1 (format nil "tr1!~A" i))
        (t1 (format nil "t1!~A" i))
        (i2 (+ (loop for i2 upfrom 1
                while (or (search (format nil "tr2!~A" i2) sc)
                          (search (format nil "t2!~A" i2) sc))
                count i2) 1))
        (tr2 (format nil "tr2!~A" i2))
        (t2 (format nil "t2!~A" i2))
        (time_c_d (format nil "~A - Duration(~A)" time_c tr1))
        (cstr (format nil "~A(~A) /= ~A(~A)" vname_c time_c_d vname_c time_c))
        (cstr2 (format nil "~A <= ~A" t1 time_c_d))
    ) (then@
        (expand "const" fnum_c)
        (lemma "var_changes")
        (inst -1 time_c)
        (spread (split -1) (
          (then@
            (skolem -1 tr1)
            (flatten -1)
            (let ((fnum2 (- fnum_c 2)))
              (then@
                (expand "Change1" fnum2)
                (flatten fnum2)
                (skolem fnum2 t1)
                (flatten fnum2)
              )
            )
            (spread (case cstr) (
              (then@
                (lemma "trans_entry")
                (inst -1 tr1 time_c_d)
                (lemma "trans_exit")
                (inst -1 tr1 time_c)
                (spread (case-trans tr1) (
                  (branch (split -1) (
                    (then@
                      (replace -1)
                      (delete -1)
                      (if do_grind
                          (try (try (my-grind) (fail) (skip))
                               (skip) (skip))
                          (skip)
                      )
                      )
                    )
                    (then@ (flatten) (assert))
                  ))
                ))
              )
              (spread (case (cstr2)) (
                (then@
                  (let ((fnum4 (- fnum_c 4)))
                    (inst fnum4 time_c_d))
                  (assert)
                )
                (then@
                  (lemma "vars_no_change")
                  (inst -1 time_c_d t1)
                  (assert)
                  (spread (split -1) (
                    (then@
                      (inst -1 t1)
                      (assert)
                      (let ((fnum4 (- fnum_c 4)))
                        (inst fnum4 t1))
                      (assert)
                      (expand "Vars_No_Change")
                    )
                    (then@
                      (skolem 1 (tr2 t2))
                      (flatten 1)
                      (lemma "trans_mutex_end")
                      (inst -1 tr1 time_c_d)
                      (assert)
                      (inst -1 tr2 t2)
                      (assert)
                    )
                  ))
                )
              ))
            ))
          ))
        ))
        (try (try (grind) (fail) (skip)) (skip) (skip))
      ))
    ))
    "Determines transitions that ended at TIME_C given the change expression
     of FNUM_C to variable VNAME_C at time TIME_C.

     DO_GRIND specifies whether to grind each transition case."
    "Determining transitions that ended at change"
)

(defstep is-pred-indeterminate (tr_from t_from)
    (let (
        (sc (format nil "~A" (collect-skolem-constants)))
        (i (+ (loop for i upfrom 1
                while (or (search (format nil "tr1!~A" i) sc)
                          (search (format nil "t1!~A" i) sc))
                count i) 1))
        (tr1 (format nil "tr1!~A" i))
        (t1 (format nil "t1!~A" i))
        (t1_d (format nil "~A + Duration(~A)" t1 tr1))
        (i2 (+ (loop for i2 upfrom 1
                while (or (search (format nil "tr2!~A" i2) sc)
                          (search (format nil "t2!~A" i2) sc))
                count i2) 1))
        (tr2 (format nil "tr2!~A" i2))
        (t2 (format nil "t2!~A" i2))
        (cstr (format nil "~A < ~A + Duration(~A)" t2 t1 tr1))
        (cstr2 (format nil "~A + Duration(~A) > ~A" t2 tr2 t_from))
    ) (then@
        (expand "Is_Predecessor")
        (flatten)
        (lemma "trans_entry")
        (inst -1 tr_from t_from)
        (astral-expand -1)
        (assert)
        (step-bw-indeterminate t_from)
        (inst 1 t1)
        (assert)
        (spread (split 1) (
          (assert)
          (then@
            (skolem 1 (tr2 t2))
            (flatten 1)
            (spread (case cstr) (
              (then@
```

```
                    (lemma "trans_mutex")
                    (inst -1 tr1 t1)
                    (assert)
                    (flatten -1)
                    (inst -2 tr2 t2)
                    (assert)
                )
                (spread (case cstr2) (
                    (then@
                        (lemma "trans_mutex")
                        (inst -1 tr2 t2)
                        (assert)
                        (flatten -1)
                        (inst -2 tr_from t_from)
                        (assert)
                    )
                    (then@
                        (inst -12 tr2 t2)
                        (assert)
                    )
                ))
            ))
        )
    ))
    "Attempts proof of untimed Is_Predecessor obligation from the transition
     TR_FROM, which fired at T_FROM."
    "Attempting proof of untimed Is_Predecessor obligation."
)

(defstep is-pred-immediate (tr_from t_from tr_to)
    (let (
        (sc (format nil "~A" (collect-skolem-constants)))
        (i2 (+ (loop for i2 upfrom 1
                while (or (search (format nil "tr2!~A" i2) sc)
                          (search (format nil "t2!~A" i2) sc))
                count i2) 1))
        (tr2 (format nil "tr2!~A" i2))
        (t2 (format nil "t2!~A" i2))
        (t_to (format nil "~A - Duration(~A)" t_from tr_to))
    ) (then@
        (expand "Is_Predecessor")
        (flatten)
        (step-bw-immediate tr_from t_from tr_to 1)
        (assert)
        (spread (split 1) (
            (then@
                (skosimp* 1)
                (assert)
            )
            (then@
                (skolem 1 (tr2 t2))
                (flatten 1)
                (lemma "trans_mutex")
                (inst -1 tr_to t_to)
                (assert)
                (flatten)
                (inst -2 tr2 t2)
                (assert)
            )
        ))
    ))
    "Attempts proof of immediate Is_Predecessor obligation from the transition
     TR_FROM, which fired at T_FROM, to the transition TR_TO."
    "Attempting proof of immeidate Is_Predecessor obligation."
)

(defstep expand-is-pred-indeterminate (fnum t_from tr_to &optional (f_hide T))
    (let (
        (sc (format nil "~A" (collect-skolem-constants)))
        (i (+ (loop for i upfrom 1
                while (search (format nil "t1!~A" i) sc)
                count i) 1))
        (t1 (format nil "t1!~A" i))
        (t1_d (format nil "~A + Duration(~A)" t1 tr_to))
    ) (then@
        (expand "Is_Predecessor" fnum)
        (assert)
        (skolem fnum t1)
        (flatten fnum)
        (let ((fnum2 (- fnum 2)))
            (inst fnum2 t_from))
        (if f_hide
            (let ((fnum3 (- fnum 3)))
                (hide fnum3))
            (skip)
        )
        (lemma "trans_entry")
        (inst -1 tr_to t1)
        (lemma "trans_exit")
        (inst -1 tr_to t1_d)
        (assert)
    ))
    "Expands the untimed Is_Predecessor declaration in formula number FNUM
     for the transition TR_FROM, which fired at T_FROM.

     F_HIDE specifies whether to hide quantification stating nothing fired."
    "Expanding untimed Is_Predecessor declaration."
)
```

# Appendix D

# Example PVS Proofs

## D.1. PVS Global Schedule Proof (described in 10.6)

```
 |-------
{1}     (FORALL (T1: time):
           Invariant(T1) AND Environment(T1)
           AND i_Invariant(T1) AND i_Schedule(T1))
           AND (FORALL (T1: time): T1 ≤ T0 IMPLIES Schedule(T1))
         IMPLIES
         (FORALL (T1: time):
           T0 < T1 AND T1 < T0 + DELTA IMPLIES s_Schedule(1)(T1))

;; two processes are in their critical sections

("")
   (SKOSIMP*)
   (INST -1 "T1!1")
   (FLATTEN)
   (DELETE -1 -2 -5)
   (ASTRAL-EXPAND-CLAUSE 1)
   (SKOSIMP*)
   (TYPEPRED "V1!1")
   (TYPEPRED "V1!2")
   (ASTRAL-EXPAND (-1 -2))
   (FLATTEN)

{-1}    1 ≤ V1!2
{-2}    V1!2 ≤ n_procs
{-3}    1 ≤ V1!1
{-4}    V1!1 ≤ n_procs
[-5]    i_Invariant(T1!1)
[-6]    i_Schedule(T1!1)
[-7]    T0 < T1!1
[-8]    T1!1 < T0 + DELTA
[-9]    i_proc__in_critical(procs(V1!1))(T1!1)
[-10]   i_proc__in_critical(procs(V1!2))(T1!1)
 |-------
[1]     V1!1 = V1!2

;; the number of neither of the processes is zero (1/2)

   (ASTRAL-EXPAND-CLAUSE -5)
   (INST-CP -5 "procs(V1!1)")
   (("1"
      (INST -5 "procs(V1!2)")
      (("1"
         (FLATTEN)
         (DELETE-BAD)
         (ASSERT)

[-1]    1 ≤ V1!2
[-2]    V1!2 ≤ n_procs
[-3]    1 ≤ V1!1
[-4]    V1!1 ≤ n_procs
[-5]    i_Schedule(T1!1)
[-6]    T0 < T1!1
{-7}    T1!1 < DELTA + T0
[-8]    i_proc__in_critical(procs(V1!1))(T1!1)
[-9]    i_proc__in_critical(procs(V1!2))(T1!1)
 |-------
{1}     i_proc__choosing(procs(V1!2))(T1!1)
{2}     i_proc__number(procs(V1!2))(T1!1) = 0
{3}     i_proc__choosing(procs(V1!1))(T1!1)
{4}     i_proc__number(procs(V1!1))(T1!1) = 0
[5]     V1!1 = V1!2

;; both processes have the same number

         (ASTRAL-EXPAND-CLAUSE -5)
         (INST-CP -5 "procs(V1!1)")
         (INST -5 "procs(V1!2)")
         (FLATTEN)
         (INST -5 "V1!1" "V1!2")
         (INST -6 "V1!1" "V1!2")
         (INST -7 "V1!2" "V1!1")
         (INST -8 "V1!2" "V1!1")
         (ASSERT)

[-1]    1 ≤ V1!2
[-2]    V1!2 ≤ n_procs
[-3]    1 ≤ V1!1
[-4]    V1!1 ≤ n_procs
{-5}    i_proc__number(procs(V1!2))(T1!1) < i_proc__number(procs(V1!1))(T1!1)
        OR   i_proc__number(procs(V1!2))(T1!1)
           = i_proc__number(procs(V1!1))(T1!1)
{-6}    i_proc__number(procs(V1!2))(T1!1) < i_proc__number(procs(V1!1))(T1!1)
        OR V1!2 < V1!1
{-7}    i_proc__number(procs(V1!1))(T1!1) < i_proc__number(procs(V1!2))(T1!1)
        OR   i_proc__number(procs(V1!1))(T1!1)
           = i_proc__number(procs(V1!2))(T1!1)
{-8}    i_proc__number(procs(V1!1))(T1!1) < i_proc__number(procs(V1!2))(T1!1)
        OR V1!1 < V1!2
[-9]    T0 < T1!1
[-10]   T1!1 < DELTA + T0
[-11]   i_proc__in_critical(procs(V1!1))(T1!1)
[-12]   i_proc__in_critical(procs(V1!2))(T1!1)
 |-------
[1]     i_proc__choosing(procs(V1!2))(T1!1)
[2]     i_proc__number(procs(V1!2))(T1!1) = 0
[3]     i_proc__choosing(procs(V1!1))(T1!1)
[4]     i_proc__number(procs(V1!1))(T1!1) = 0
[5]     V1!1 = V1!2

         (SPLIT -5)

;; achieve contradiction for numbers equal and unequal

         (("1" (ASSERT)) ("2" (ASSERT))))

;; the number of neither of the processes is zero (2/2)

        ("2" (ASSERT))))
   ("2" (ASSERT))))
```

# D.2. PVS Transition Sequence Analysis Proof (described in 10.7.2.5)

```
 |-------
{1}     (FORALL (T1: time):
           Invariant(T1) AND Environment(T1) AND Imported_Variable(T1)
           AND Further_Environment(1)(T1) AND Constant_Refinement(1)(T1)
           AND Transition_Selection(1)(T1))
           AND (FORALL (TR1: transition): DELTA < Duration(TR1))
           AND (FORALL (T1: time): T1 ≤ T0 IMPLIES Schedule(T1))
        IMPLIES
        (FORALL (T1: time):
           T0 < T1 AND T1 < T0 + DELTA IMPLIES s_Schedule(9)(T1))

;; start_busytone and start_ringback are only transitions that assert
;;   busytone and ringback (1/2)

("")
   (SKOSIMP*)
   (ASTRAL-EXPAND-CLAUSE 1)
   (FLATTEN)
   (CASE "FORALL (T1: time): T1 ≤ T0 IMPLIES
      Schedule(T1) AND Invariant(T1)")
   (("1"
      (HIDE -2 -4)
      (TRY-UNTIMED -1 -2)

;; start_ringback case

      (("1" (POSTPONE))

;; start_busytone case

      ("2"
         (ASSERT)

{-1}    Exit(start_ringback, t1!1)
{-2}    Entry(start_ringback, t1!1 - Duration(start_ringback))
[-3]    t1!1 - Duration(start_ringback) ≥ 0
[-4]    Fired(start_ringback, t1!1 - Duration(start_ringback))
[-5]    T0 < t1!1
[-6]    t1!1 ≤ T1!1
{-7}    Vars_No_Change(t1!1, T1!1)
{-8}    Schedule(t1!1 - Duration(start_ringback))
           AND Invariant(t1!1 - Duration(start_ringback))
[-9]    DELTA < Duration(start_ringback)
[-10]   T0 < T1!1
[-11]   T1!1 < DELTA + T0
[-12]   busytone(T1!1)
[-13]   ringback(T1!1)
 |-------

;; determine transitions that can precede start_busytone

         (REVEAL -4)
         (STEP-BW-INDETERMINATE "t1!1-Duration(start_busytone)" -1)
         (("1"
            (ASSERT)

{-1}    Exit(enter_digit, Duration(enter_digit) + t1!2)
{-2}    Entry(enter_digit, t1!2)
{-3}    Vars_No_Change(Duration(enter_digit) + t1!2,
           t1!1 - Duration(start_busytone))
[-4]    Duration(enter_digit) + t1!2 ≤ t1!1 - Duration(start_busytone)
[-5]    Fired(enter_digit, t1!2)
[-6]    FORALL (tr2: transition, t2: time):
           Duration(enter_digit) + t1!2 < t2 + Duration(tr2)
           AND t2 + Duration(tr2) ≤ t1!1 - Duration(start_busytone)
        IMPLIES NOT Fired(tr2, t2)
[-7]    Schedule(Duration(enter_digit) + t1!2)
           AND Invariant(Duration(enter_digit) + t1!2)
[-8]    Schedule(t1!2) AND Invariant(t1!2)
[-9]    Exit(start_busytone, t1!1)
[-10]   Entry(start_busytone, t1!1 - Duration(start_busytone))
[-11]   t1!1 - Duration(start_busytone) ≥ 0
[-12]   Fired(start_busytone, t1!1 - Duration(start_busytone))
[-13]   T0 < t1!1
[-14]   t1!1 ≤ T1!1
[-15]   Vars_No_Change(t1!1, T1!1)
[-16]   Schedule(t1!1 - Duration(start_busytone))
[-17]   Invariant(t1!1 - Duration(start_busytone))
[-18]   DELTA < Duration(start_busytone)
[-19]   T0 < T1!1
[-20]   T1!1 < DELTA + T0
[-21]   busytone(T1!1)
[-22]   ringback(T1!1)
 |-------

;; enter_digit case
;; suppose ringback holds

            (ASTRAL-EXPAND-CLAUSE -2)
            (FLATTEN)
            (SPLIT -3)
            (("1" (MY-GRIND))
            ("2"

{-1}    i_central_control__phone_state(in_area(self))(self)(t1!2) = dialing
[-2]    Exit(enter_digit, Duration(enter_digit) + t1!2)
[-3]    offhook(t1!2)
[-4]    Vars_No_Change(Duration(enter_digit) + t1!2,
           t1!1 - Duration(start_busytone))
[-5]    Duration(enter_digit) + t1!2 ≤ t1!1 - Duration(start_busytone)
[-6]    Fired(enter_digit, t1!2)
[-7]    FORALL (tr2: transition, t2: time):
           Duration(enter_digit) + t1!2 < t2 + Duration(tr2)
           AND t2 + Duration(tr2) ≤ t1!1 - Duration(start_busytone)
        IMPLIES NOT Fired(tr2, t2)
[-8]    Schedule(Duration(enter_digit) + t1!2)
[-9]    Invariant(Duration(enter_digit) + t1!2)
[-10]   Schedule(t1!2)
[-11]   Invariant(t1!2)
[-12]   Exit(start_busytone, t1!1)
[-13]   Entry(start_busytone, t1!1 - Duration(start_busytone))
[-14]   t1!1 - Duration(start_busytone) ≥ 0
[-15]   Fired(start_busytone, t1!1 - Duration(start_busytone))
[-16]   T0 < t1!1
[-17]   t1!1 ≤ T1!1
[-18]   Vars_No_Change(t1!1, T1!1)
[-19]   Schedule(t1!1 - Duration(start_busytone))
[-20]   Invariant(t1!1 - Duration(start_busytone))
[-21]   DELTA < Duration(start_busytone)
[-22]   T0 < T1!1
[-23]   T1!1 < DELTA + T0
[-24]   busytone(T1!1)
[-25]   ringback(T1!1)
 |-------

;; suppose ~ringback holds

               (REVEAL -11)
               (INST -1 "t1!2")
               (FLATTEN)
               (DELETE -1 -2 -4 -5 -6)
               (ASTRAL-EXPAND-CLAUSE -1)
               (FLATTEN)
               (DELETE -1 -2 -4 -5)
               (ASSERT)
               (SKOSIMP*)
               (ASSERT)
               (ASTRAL-EXPAND-ALL -7)
               (FLATTEN)
               (SKOSIMP*)
               (REPLACE -7)
               (DELETE -7 -9)
               (LEMMA "trans_entry")
               (INST -1 "enter_digit" "V1!2")
               (ASSERT)
               (INST-CP -6 "V1!2")
               (ASSERT)
               (ASTRAL-EXPAND-CLAUSE -1)
               (FLATTEN)

{-1}    offhook(V1!2)
{-2}    dialtone(V1!2)
[-3]    V1!1 ≤ V1!2
[-4]    V1!2 < V1!3
[-5]    V1!3 < V1!4
[-6]    V1!4 ≤ t1!2
[-7]    FORALL (V1_279: time):
           V1!1 ≤ V1_279 AND V1_279 < V1!4
```

```
        IMPLIES
          i_central_control__phone_state(in_area(self))(self)(V1_279)
          = ready_to_dial
[-8]    i_central_control__phone_state(in_area(self))(self)(V1!2) = ready_to_dial
[-9]    FORALL (V1: time):
          V1!4 ≤ V1 AND V1 ≤ t1!2
        IMPLIES i_central_control__phone_state(in_area(self))(self)(V1) = dialing
[-10]   Fired(enter_digit, V1!2)
[-11]   End1(enter_digit, LAMBDA (t1: time): V1!3)(V1!3)
[-12]   i_central_control__phone_state(in_area(self))(self)(V1!4) = dialing
[-13]   i_central_control__phone_state(in_area(self))(self)(t1!2) = dialing
[-14]   Exit(enter_digit, Duration(enter_digit) + t1!2)
[-15]   offhook(t1!2)
[-16]   Vars_No_Change(Duration(enter_digit) + t1!2,
          t1!1 - Duration(start_busytone))
[-17]   Duration(enter_digit) + t1!2 ≤ t1!1 - Duration(start_busytone)
[-18]   Fired(enter_digit, t1!2)
[-19]   FORALL (tr2: transition, t2: time):
          Duration(enter_digit) + t1!2 < t2 + Duration(tr2)
          AND t2 + Duration(tr2) ≤ t1!1 - Duration(start_busytone)
        IMPLIES NOT Fired(tr2, t2)
[-20]   Schedule(Duration(enter_digit) + t1!2)
[-21]   Invariant(Duration(enter_digit) + t1!2)
[-22]   Schedule(t1!2)
[-23]   Invariant(t1!2)
[-24]   Exit(start_busytone, t1!1)
[-25]   Entry(start_busytone, t1!1 - Duration(start_busytone))
[-26]   t1!1 - Duration(start_busytone) ≥ 0
[-27]   Fired(start_busytone, t1!1 - Duration(start_busytone))
[-28]   T0 < t1!1
[-29]   t1!1 ≤ T1!1
[-30]   Vars_No_Change(t1!1, T1!1)
[-31]   Schedule(t1!1 - Duration(start_busytone))
[-32]   Invariant(t1!1 - Duration(start_busytone))
[-33]   DELTA < Duration(start_busytone)
[-34]   T0 < T1!1
[-35]   T1!1 < DELTA + T0
[-36]   busytone(T1!1)
[-37]   ringback(T1!1)
 |-------

;; ~ringback holds when enter_digit fires

              (REVEAL -7)
              (INST -1 "V1!2")
              (ASSERT)
              (ASTRAL-EXPAND-CLAUSE -1)
              (FLATTEN)
              (DELETE -1 -2 -3 -4 -5 -6 1 3 4 5 6 7)

[-1]    offhook(V1!2)
[-2]    dialtone(V1!2)
[-3]    V1!1 ≤ V1!2
[-4]    V1!2 < V1!3
[-5]    V1!3 < V1!4
[-6]    V1!4 ≤ t1!2
[-7]    FORALL (V1_279: time):
          V1!1 ≤ V1_279 AND V1_279 < V1!4)
        IMPLIES
          i_central_control__phone_state(in_area(self))(self)(V1_279)
          = ready_to_dial
[-8]    i_central_control__phone_state(in_area(self))(self)(V1!2) = ready_to_dial
[-9]    FORALL (V1: time):
          V1!4 ≤ V1 AND V1 ≤ t1!2
        IMPLIES i_central_control__phone_state(in_area(self))(self)(V1) = dialing
[-10]   Fired(enter_digit, V1!2)
[-11]   End1(enter_digit, LAMBDA (t1: time): V1!3)(V1!3)
[-12]   i_central_control__phone_state(in_area(self))(self)(V1!4) = dialing
[-13]   i_central_control__phone_state(in_area(self))(self)(t1!2) = dialing
[-14]   Exit(enter_digit, Duration(enter_digit) + t1!2)
[-15]   offhook(t1!2)
[-16]   Vars_No_Change(Duration(enter_digit) + t1!2,
          t1!1 - Duration(start_busytone))
[-17]   Duration(enter_digit) + t1!2 ≤ t1!1 - Duration(start_busytone)
[-18]   Fired(enter_digit, t1!2)
[-19]   FORALL (tr2: transition, t2: time):
          Duration(enter_digit) + t1!2 < t2 + Duration(tr2)
          AND t2 + Duration(tr2) ≤ t1!1 - Duration(start_busytone)
        IMPLIES NOT Fired(tr2, t2)
[-20]   Schedule(Duration(enter_digit) + t1!2)
[-21]   Invariant(Duration(enter_digit) + t1!2)
[-22]   Schedule(t1!2)
[-23]   Invariant(t1!2)
[-24]   Exit(start_busytone, t1!1)
[-25]   Entry(start_busytone, t1!1 - Duration(start_busytone))
[-26]   t1!1 - Duration(start_busytone) ≥ 0
[-27]   Fired(start_busytone, t1!1 - Duration(start_busytone))
[-28]   T0 < t1!1
[-29]   t1!1 ≤ T1!1
[-30]   Vars_No_Change(t1!1, T1!1)
[-31]   Schedule(t1!1 - Duration(start_busytone))
[-32]   Invariant(t1!1 - Duration(start_busytone))
[-33]   DELTA < Duration(start_busytone)
[-34]   T0 < T1!1
[-35]   T1!1 < DELTA + T0
[-36]   busytone(T1!1)
[-37]   ringback(T1!1)
 |-------
[1]     ringback(V1!2)

;; start_ringback fires

              (CASE "FORALL (t: time):V1!2 ≤ t
                AND t ≤ t1!2 IMPLIES NOT ringback(t)")
              (("1" (INST -1 "t1!2") (MY-GRIND))
              ("2"
                (SKOSIMP*)
                (LEMMA "exists_change1[boolean]")
                (INST -1 "ringback" "V1!2" "t!1")
                (ASSERT)
                (SKOSIMP*)
                (CASE "Change1(ringback, const(t2!1))(t2!1)")
                (("1"
                  (DELETE -5 -22 -24 -27 -28 -29 -30 -31
                    -32 -33 -38 -39 -40)
                  (CHANGE-FIRE -1 "ringback" "t2!1")
                  (ASSERT)

{-1}    Exit(start_ringback, t2!1)
{-2}    Entry(start_ringback, t2!1 - Duration(start_ringback))
[-3]    ringback(t2!1 - Duration(start_ringback)) ≠ ringback(t2!1)
[-4]    t2!1 - Duration(start_ringback) ≥ 0
[-5]    Fired(start_ringback, t2!1 - Duration(start_ringback))
[-6]    t1!4 < t2!1
[-7]    FORALL (t3: time):
          t1!4 ≤ t3 AND t3 < t2!1 IMPLIES NOT ringback(t3) = ringback(t2!1)
[-8]    FORALL (t2: time):
          t2!1 ≤ t2 AND t2 ≤ t2!1 IMPLIES ringback(t2) = ringback(t2!1)
[-9]    V1!2 < t2!1
[-10]   t2!1 ≤ t!1
[-11]   ringback(t2!1)
[-12]   V1!2 ≤ t!1
[-13]   t!1 ≤ t1!2
[-14]   ringback(t!1)
[-15]   offhook(V1!2)
[-16]   dialtone(V1!2)
[-17]   V1!1 ≤ V1!2
[-18]   V1!2 < V1!3
[-19]   V1!3 < V1!4
[-20]   V1!4 ≤ t1!2
[-21]   FORALL (V1_279: time):
          V1!1 ≤ V1_279 AND V1_279 < V1!4)
        IMPLIES
          i_central_control__phone_state(in_area(self))(self)(V1_279)
          = ready_to_dial
[-22]   i_central_control__phone_state(in_area(self))(self)(V1!2) = ready_to_dial
[-23]   FORALL (V1: time):
          V1!4 ≤ V1 AND V1 ≤ t1!2
        IMPLIES i_central_control__phone_state(in_area(self))(self)(V1) = dialing
[-24]   Fired(enter_digit, V1!2)
[-25]   End1(enter_digit, LAMBDA (t1: time): V1!3)(V1!3)
[-26]   i_central_control__phone_state(in_area(self))(self)(V1!4) = dialing
[-27]   i_central_control__phone_state(in_area(self))(self)(t1!2) = dialing
[-28]   offhook(t1!2)
[-29]   Duration(enter_digit) + t1!2 ≤ t1!1 - Duration(start_busytone)
[-30]   Fired(enter_digit, t1!2)
[-31]   t1!1 - Duration(start_busytone) ≥ 0
[-32]   Fired(start_busytone, t1!1 - Duration(start_busytone))
[-33]   T0 < t1!1
[-34]   t1!1 ≤ T1!1
[-35]   DELTA < Duration(start_busytone)
[-36]   T0 < T1!1
[-37]   T1!1 < DELTA + T0
[-38]   busytone(T1!1)
```

```
[-39]    ringback(T1!1)
 |-------
[1]      ringback(V1!2)

;; start_ringback could not have fired after start of the earlier enter_digit

                    (ASTRAL-EXPAND-CLAUSE -2)
                    (FLATTEN)
                    (CASE "t2!1 - t7 ≥ V1!2")
                    (("1"
                       (EXPAND "Duration")
                       (INST -23 "t2!1 - t7")
                       (INST -25 "t2!1 - t7")
                       (ASSERT))
                     ("2"
                       (LEMMA "trans_mutex")
                       (INST -1 "start_ringback" "t2!1 - Duration(start_ringback)")
                       (ASSERT)
                       (FLATTEN)
                       (INST -2 "enter_digit" "V1!2")
                       (DELETE -1)
                       (EXPAND "Duration")
                       (PROPAX))))
                  ("2" (HIDE -4) (DELETE -) (REVEAL -1) (DELETE 2) (GRIND))
                  ("3" (EXPAND "const" 1) (ASSERT)))))))

;; start_ringback case

        ("2"
           (ASSERT)

{-1}     Exit(start_ringback, Duration(start_ringback) + t1!2)
{-2}     Entry(start_ringback, t1!2)
{-3}     Vars_No_Change(Duration(start_ringback) + t1!2,
           t1!1 - Duration(start_busytone))
[-4]     Duration(start_ringback) + t1!2 ≤ t1!1 - Duration(start_busytone)
[-5]     Fired(start_ringback, t1!2)
[-6]     FORALL (tr2: transition, t2: time):
           Duration(start_ringback) + t1!2 < t2 + Duration(tr2)
           AND t2 + Duration(tr2) ≤ t1!1 - Duration(start_busytone)
         IMPLIES NOT Fired(tr2, t2)
[-7]     Schedule(Duration(start_ringback) + t1!2)
           AND Invariant(Duration(start_ringback) + t1!2)
[-8]     Schedule(t1!2) AND Invariant(t1!2)
[-9]     Exit(start_busytone, t1!1)
[-10]    Entry(start_busytone, t1!1 - Duration(start_busytone))
[-11]    t1!1 - Duration(start_busytone) ≥ 0
[-12]    Fired(start_busytone, t1!1 - Duration(start_busytone))
[-13]    T0 < t1!1
[-14]    t1!1 ≤ T1!1
[-15]    Vars_No_Change(t1!1, T1!1)
[-16]    Schedule(t1!1 - Duration(start_busytone))
[-17]    Invariant(t1!1 - Duration(start_busytone))
[-18]    DELTA < Duration(start_busytone)
[-19]    T0 < T1!1
[-20]    T1!1 < DELTA + T0
[-21]    busytone(T1!1)
[-22]    ringback(T1!1)
 |-------

;; previous value of phone_state was dialing

           (ASTRAL-EXPAND-CLAUSE -2)
           (FLATTEN)
           (ASTRAL-EXPAND-CLAUSE -14)
           (FLATTEN)
           (REVEAL -11)
           (HIDE -9)
           (EXPAND "Duration")
           (INST -1 "t1!1 - t9")
           (FLATTEN)
           (DELETE -1 -2 -4 -5 -6)
           (ASTRAL-EXPAND-CLAUSE -1)
           (FLATTEN)
           (DELETE -2 -3 -4 -5)
           (ASSERT)
           (SKOSIMP*)
           (ASSERT)

[-1]     V1!1 ≤ V1!2
[-2]     V1!2 < V1!3
[-3]     V1!3 < V1!4
[-4]     V1!4 ≤ t1!1 - t9
[-5]     FORALL (V1_530: time):
           V1!1 ≤ V1_530 AND V1_530 < V1!4)
         IMPLIES
           i_central_control__phone_state(in_area(self))(self)(V1_530) = dialing
[-6]     FORALL (V1: time):
           V1!4 ≤ V1 AND V1 ≤ t1!1 - t9
         IMPLIES i_central_control__phone_state(in_area(self))(self)(V1) = busy
{-7}     Start1(enter_digit, LAMBDA (t1: time): V1!2)(V1!2)
{-8}     End1(enter_digit, LAMBDA (t1: time): V1!3)(V1!3)
[-9]     i_central_control__phone_state(in_area(self))(self)(V1!4) = busy
[-10]    Exit(start_ringback, t1!2 + t7)
[-11]    offhook(t1!2)
[-12]    i_central_control__phone_state(in_area(self))(self)(t1!2) = waiting
[-13]    i_central_control__enabled_ringback_pulse(in_area(self))(self)(t1!2)
[-14]    Vars_No_Change(t1!2 + t7, t1!1 - t9)
[-15]    t1!2 + t7 ≤ t1!1 - t9
[-16]    Fired(start_ringback, t1!2)
[-17]    Schedule(t1!2 + t7)
[-18]    Invariant(t1!2 + t7)
[-19]    Schedule(t1!2)
[-20]    Invariant(t1!2)
[-21]    Exit(start_busytone, t1!1)
[-22]    offhook(t1!1 - t9)
[-23]    i_central_control__phone_state(in_area(self))(self)(t1!1 - t9) = busy
[-24]    t1!1 - t9 ≥ 0
[-25]    Fired(start_busytone, t1!1 - t9)
[-26]    T0 < t1!1
[-27]    t1!1 ≤ T1!1
[-28]    Vars_No_Change(t1!1, T1!1)
[-29]    Schedule(t1!1 - t9)
[-30]    Invariant(t1!1 - t9)
[-31]    DELTA < t9
[-32]    T0 < T1!1
[-33]    T1!1 < DELTA + T0
[-34]    busytone(T1!1)
[-35]    ringback(T1!1)
 |-------
[1]      ringback(t1!2)
[2]      busytone(t1!1 - t9)

;; start_ringback cannot be a predecessor of start_busytone

              (REVEAL -1)
              (EXPAND "Duration" -1 1)
              (EXPAND "Duration" -1 3)
              (ASTRAL-EXPAND-ALL -9)
              (FLATTEN)
              (SKOSIMP*)
              (EXPAND "Base_Trans" -9)
              (REPLACE -9)
              (DELETE -9 -12)
              (INST -1 "enter_digit" "V1!3 - Duration(enter_digit)")
              (("1"
                 (ASSERT)
                 (INST -5 "t1!2")
                 (INST -6 "t1!2")
                 (ASSERT)
                 (SPLIT 2)
                 (("1"
                    (CASE "t1!2 < V1!2")
                    (("1"
                       (ASTRAL-EXPAND-ALL -6)
                       (FLATTEN)
                       (SKOSIMP*)
                       (REPLACE -6)
                       (DELETE -6)
                       (LEMMA "trans_mutex")
                       (INST -1 "start_ringback" "t1!2")
                       (ASSERT)
                       (FLATTEN)
                       (DELETE -1)
                       (INST -1 "enter_digit" "V1!2")
                       (ASSERT)
                       (EXPAND "Duration")
                       (ASSERT))
                       ("2" (ASSERT))))
                  ("2" (ASSERT))))
               ("2" (ASSERT)))))))

;; start_busytone and start_ringback are only transitions that assert
;;   busytone and ringback (2/2)

     ("2"
```

```
    (SKOSIMP*)
    (INST -2 "T1!2")
    (INST -4 "T1!2")
    (FLATTEN)
    (ASSERT))))
```

## D.3. PVS Timed Operator Analysis Proof (described in 10.7.3.2)

```
 |-------
{1}   (FORALL (TR1: transition): DELTA < Duration(TR1))
          AND (FORALL (T1: time): T1 ≤ T0 IMPLIES Invariant(T1))
      IMPLIES
      (FORALL (T1: time):
          T0 < T1 AND T1 < T0 + DELTA IMPLIES s_Invariant(6)(T1))

;; find transitions that can change number appropriately

("")
   (SKOSIMP*)
   (ASTRAL-EXPAND-CLAUSE 1)
   (SKOSIMP*)
   (DELETE -1 -2)
   (CHANGE-FIRE -3 "number" "T1!1")
   (ASSERT)

{-1}  Exit(set_number, T1!1)
{-2}  Entry(set_number, T1!1 - Duration(set_number))
[-3]  number(T1!1 - Duration(set_number)) ≠ number(T1!1)
[-4]  T1!1 - Duration(set_number) ≥ 0
[-5]  Fired(set_number, T1!1 - Duration(set_number))
[-6]  T0 < T1!1
{-7}  T1!1 < DELTA + T0
[-8]  t1!1 < T1!1
[-9]  FORALL (t3: time):
          t1!1 ≤ t3 AND t3 < T1!1 IMPLIES NOT number(t3) = number(T1!1)
[-10] FORALL (t2: time):
          T1!1 ≤ t2 AND t2 ≤ T1!1 IMPLIES number(t2) = number(T1!1)
 |-------
[1]   number(T1!1) = 0
[2]   number(T1!1) ≥ 1 + i_proc__number(procs(V1!1))(T1!1 - exec_time)

;; exit assertion of set_number satisfies requirement

   (ASTRAL-EXPAND-CLAUSE -1)
   (FLATTEN)
   (INST -1 "V1!1"))
```

## D.4. PVS Liveness Property Proof (described in 10.8.2.5)

```
 |-------
{1}   (FORALL (T1: time):
          Invariant(T1) AND Environment(T1) AND Imported_Variable(T1)
          AND Further_Environment(1)(T1) AND Constant_Refinement(1)(T1)
          AND Transition_Selection(1)(T1))
          AND (FORALL (TR1: transition): DELTA < Duration(TR1))
          AND (FORALL (T1: time): T1 ≤ T0 IMPLIES Schedule(T1))
      IMPLIES
      (FORALL (T1: time):
          T0 < T1 AND T1 < T0 + DELTA IMPLIES s_Schedule(1)(T1))

;; split proof into cases based on operating environment and local state (1/2)

("")
   (SKOSIMP*)
   (ASTRAL-EXPAND-CLAUSE 1)
   (SKOSIMP*)
   (HIDE -1 -2 -3 -4 -5)
   (EXPAND "Change1")
   (TYPEPRED "choose!  (t2: time):
      t2 ≤ T1!1 AND
      Change1(i_sensor__train_in_r(V1!1), const(t2))(T1!1)")
   (("1"
      (NAME "ct" "choose!  (t2: time):
         t2 ≤ T1!1 AND
         Change1(i_sensor__train_in_r(V1!1), const(t2))(T1!1)")
      (REPLACE -1)
      (DELETE -1)
      (LEMMA "idle_or_firing")
      (INST -1 "ct")
      (SPLIT -1)
      (("1" (POSTPONE))
      ("2"
         (SKOSIMP*)
         (CASE-TRANS "tr2!1")
         (("1"
            (SPLIT -1)
            (("1" (POSTPONE)) ("2" (POSTPONE)) ("3" (POSTPONE))

;; up case

               ("4"
                  (REPLACE -1)
                  (DELETE -1)
                  (EXPAND "Duration")

{-1}  ct - up_dur < t2!1
[-2]  t2!1 < ct
[-3]  Fired(up, t2!1)
[-4]  ct ≥ 0
[-5]  ct ≤ T1!1
[-6]  Change1[boolean](i_sensor__train_in_r(V1!1), const(ct))(T1!1)
[-7]  i_sensor__train_in_r(V1!1)(T1!1)
[-8]  T1!1 - ct ≥ dist_r_to_i / max_speed - response_time
 |-------
[1]   position(T1!1) = lowered

;; lower fires immediately at the end of up (1/2)

                     (STEP-FW-IMMEDIATE "up" "t2!1" "lower" T)
                     (("1"
                        (ASSERT)

{-1}  Entry(lower, Duration(up) + t2!1)
{-2}  Fired(lower, Duration(up) + t2!1)
[-3]  Exit(up, (t2!1 + up_dur))
[-4]  Entry(up, t2!1)
[-5]  ct - up_dur < t2!1
[-6]  t2!1 < ct
[-7]  Fired(up, t2!1)
[-8]  ct ≥ 0
[-9]  ct ≤ T1!1
[-10] Change1(i_sensor__train_in_r(V1!1)), const(ct))(T1!1)
[-11] i_sensor__train_in_r(V1!1)(T1!1)
[-12] T1!1 - ct ≥ dist_r_to_i / max_speed - response_time
 |-------
[1]   position(T1!1) = lowered

;; down fires lower_time after the end of lower (1/2)

                     (EXPAND "Duration")
                     (LEMMA "trans_exit")
                     (INST -1 "lower" "t2!1 + up_dur + Duration(lower)")
                     (("1"
                        (ASSERT)
                        (LEMMA "local_axiom")
                        (TYPEPRED "lower_dur" "up_dur" "down_dur" "lower_time")
                        (ASTRAL-EXPAND (-1 -2 -3 -4 -5))
                        (FLATTEN)
                        (DELETE -5 -6)
                        (STEP-FW-DELAY "lower" "t2!1+up_dur" "down"
                           "t2!1+up_dur+Duration(lower)+lower_time")
                        (("1"
                           (ASSERT)

{-1}  Entry(down, Duration(lower) + lower_time + t2!1 + up_dur)
{-2}  Fired(down, Duration(lower) + lower_time + t2!1 + up_dur)
[-3]  FORALL (tr1: transition, t1: time):
```

```
          Duration(lower) + t2!1 + up_dur ≤ t1
          AND t1 < Duration(lower) + lower_time + t2!1 + up_dur
      IMPLIES NOT Fired(tr1, t1)
{-4}  Vars_No_Change(Duration(lower) + t2!1 + up_dur,
          Duration(lower) + lower_time + t2!1 + up_dur)
[-5]  lower_time > 0
[-6]  lower_dur > 0
[-7]  up_dur > 0
[-8]  down_dur > 0
[-9]  lower_time > 0
[-10] dist_r_to_i / max_speed
      ≥ down_dur + lower_dur + lower_time + response_time + up_dur
[-11] Exit(lower, Duration(lower) + t2!1 + up_dur)
[-12] Entry(lower, t2!1 + up_dur)
[-13] Fired(lower, t2!1 + up_dur)
[-14] Exit(up, (t2!1 + up_dur))
[-15] Entry(up, t2!1)
[-16] ct - up_dur < t2!1
[-17] t2!1 < ct
[-18] Fired(up, t2!1)
[-19] ct ≥ 0
[-20] ct ≤ T1!1
[-21] Change1(i_sensor__train_in_r(V1!1), const(ct))(T1!1)
[-22] i_sensor__train_in_r(V1!1)(T1!1)
[-23] T1!1 - ct ≥ dist_r_to_i / max_speed - response_time
 |-------
[1]   position(T1!1) = lowered
```

;; nothing fires from the end of down until now

```
(LEMMA "trans_exit")
(INST -1 "down"
   "Duration(lower) + lower_time + t2!1 + up_dur +
       Duration(down)")
((“1”
   (ASSERT)
   (ASTRAL-EXPAND-CLAUSE -1)
   (LEMMA “no_trans_fire_vnc_lt”)
   (INST -1 “down_dur + lower_dur + lower_time + t2!1 +
       up_dur” “T1!1”)
   (ASSERT)
   (SPLIT -1)
   ((“1”
      (FLATTEN)
      (INST -2 “T1!1”)
      (ASSERT)
      (ASTRAL-EXPAND-CLAUSE -2)
      (ASSERT))
   (“2”
      (INST 1 “down”)
      (EXPAND “Duration”)
      (PROPAX))
   (“3”
      (SKOSIMP*)
      (INST -3 “t1!1”)
      (ASSERT)
      (LEMMA “trans_entry”)
      (INST -1 “tr1!3” “t1!1”)
      (ASSERT)
      (ASTRAL-EXPAND-CLAUSE -4)
      (CASE-TRANS “tr1!3”)
      ((“1”
         (SPLIT -1)
         ((“1”
            (REPLACE -1)
            (ASTRAL-EXPAND-CLAUSE -2)
            (ASSERT))
         (“2”
            (REPLACE -1)
            (ASTRAL-EXPAND-CLAUSE -2)
            (ASSERT))
         (“3”
            (REPLACE -1)
            (ASTRAL-EXPAND-CLAUSE -2)
            (FLATTEN)
            (INST -2 “V1!1”)
            (ASTRAL-EXPAND-ALL -28)
            (FLATTEN)
            (INST -29 “t1!1”)
            (ASSERT))
         (“4”
            (REPLACE -1)
            (ASTRAL-EXPAND-CLAUSE -2)
                  (ASSERT))))
            (“2” (FLATTEN) (ASSERT))))))
         (“2” (ASSERT))))
```

;; down fires lower_time after the end of lower (2/2)

```
(“2”
   (ASSERT)

{-1}  Entry(raise, Duration(lower) + lower_time + t2!1 + up_dur)
{-2}  Fired(raise, Duration(lower) + lower_time + t2!1 + up_dur)
[-3]  FORALL (tr1: transition, t1: time):
          Duration(lower) + t2!1 + up_dur ≤ t1
          AND t1 < Duration(lower) + lower_time + t2!1 + up_dur
      IMPLIES NOT Fired(tr1, t1)
{-4}  Vars_No_Change(Duration(lower) + t2!1 + up_dur,
          Duration(lower) + lower_time + t2!1 + up_dur)
[-5]  lower_time > 0
[-6]  lower_dur > 0
[-7]  up_dur > 0
[-8]  down_dur > 0
[-9]  lower_time > 0
[-10] dist_r_to_i / max_speed
      ≥ down_dur + lower_dur + lower_time + response_time + up_dur
[-11] Exit(lower, Duration(lower) + t2!1 + up_dur)
[-12] Entry(lower, t2!1 + up_dur)
[-13] Fired(lower, t2!1 + up_dur)
[-14] Exit(up, (t2!1 + up_dur))
[-15] Entry(up, t2!1)
[-16] ct - up_dur < t2!1
[-17] t2!1 < ct
[-18] Fired(up, t2!1)
[-19] ct ≥ 0
[-20] ct ≤ T1!1
[-21] Change1(i_sensor__train_in_r(V1!1)), const(ct))(T1!1)
[-22] i_sensor__train_in_r(V1!1)(T1!1)
[-23] T1!1 - ct ≥ dist_r_to_i / max_speed - response_time
 |-------
[1]   raise = down
[2]   position(T1!1) = lowered

                  (ASTRAL-EXPAND-CLAUSE -1)
                  (FLATTEN)
                  (INST -1 “V1!1”)
                  (ASTRAL-EXPAND-ALL -20)
                  (FLATTEN)
                  (INST -21 “lower_dur+lower_time + t2!1 + up_dur”)
                  (ASSERT))
               (“3”
                  (ASTRAL-EXPAND-CLAUSE 1)
                  (ASTRAL-EXPAND-CLAUSE -9)
                  (ASTRAL-EXPAND-CLAUSE -2)
                  (ASSERT)
                  (EXPAND “End1” 1)
                  (TYPEPRED “choose!  (t2: time):
                     t2 ≤ lower_dur + lower_time + t2!1 + up_dur
                     AND End1(lower, const(t2))
                        (lower_dur + lower_time + t2!1+ up_dur)”)
                  ((“1”
                     (NAME “et” “choose!  (t2: time):
                        t2 ≤ lower_dur + lower_time + t2!1 + up_dur
                        AND End1(lower, const(t2))
                           (lower_dur + lower_time + t2!1 + up_dur)”)
                     ((“1”
                        (REPLACE -1)
                        (DELETE -1)
                        (EXPAND “Duration”)
                        (ASTRAL-EXPAND-ALL -3)
                        (FLATTEN)
                        (SKOSIMP* -3)
                        (EXPAND “Base_Trans”)
                        (REPLACE -3)
                        (DELETE -3)
                        (CASE “lower_dur + t2!1 + up_dur ≤ et -
                           Duration(lower)”)
                        ((“1”
                           (INST -7 “lower” “et - Duration(lower)”)
                           ((“1” (ASSERT)) (“2” (ASSERT))))
                        (“2”
                           (CASE “et < lower_dur + t2!1 + up_dur”)
                           ((“1”
                              (INST -6 “lower_dur + t2!1 + up_dur”)
                              (ASSERT))
```

```
            ("2"
               (LEMMA "trans_mutex_end")
               (INST -1 "lower" "et - Duration(lower)")
               (("1"
                  (ASSERT)
                  (INST -1 "lower" "t2!1 + up_dur")
                  (EXPAND "Duration")
                  (ASSERT))
               ("2" (ASSERT))))))))
      ("2" (SKOSIMP*) (ASTRAL-EXPAND 1))))
   ("2" (SKOSIMP*) (ASTRAL-EXPAND 1))
   ("3" (SKOSIMP*) (ASTRAL-EXPAND 1))
   ("4"
      (DELETE 2)
      (EXPAND "nonempty?")
      (EXPAND "empty?")
      (LEMMA "exists_end1")
      (INST -1 "lower" "t2!1 + up_dur + Duration(lower)")
      (ASSERT)
      (EXPAND "Duration")
      (INST -1 "lower_dur + lower_time + t2!1 + up_dur")
      (ASSERT)
      (EXPAND "member")
      (SKOSIMP*)
      (INST -4 "t3!1")
      (EXPAND "Base_Trans")
      (ASSERT))
   ("5" (SKOSIMP*) (ASTRAL-EXPAND 1))
   ("6" (SKOSIMP*) (ASTRAL-EXPAND 1))))
("4"
   (ASSERT)
   (EXPAND "Duration")
   (ASTRAL-EXPAND-CLAUSE -1)
   (FLATTEN)
   (EXPAND "End1" -2)
   (TYPEPRED "choose!  (t2: time):
      t2 ≤ t1!2
      AND End1(lower, const(t2))(t1!2)")
   (("1"
      (NAME "et" "choose!  (t2: time):
         t2 ≤ t1!2
         AND End1(lower, const(t2))(t1!2)")
      (REPLACE -1)
      (DELETE -1)
      (ASTRAL-EXPAND-ALL -3)
      (FLATTEN)
      (SKOSIMP* -3)
      (EXPAND "Base_Trans")
      (REPLACE -3)
      (DELETE -3)
      (EXPAND "Duration" (-3 -4))
      (CASE "et < t2!1 + up_dur + lower_dur")
      (("1"
         (INST -6 "t2!1 + up_dur + lower_dur")
         (ASSERT)
         (INST -6 "lower")
         (EXPAND "Duration" -6)
         (PROPAX))
      ("2"
         (CASE "lower_dur + t2!1 + up_dur ≤ e t - lower_dur")
         (("1"
            (INST -12 "lower" "et - lower_dur")
            (("1" (ASSERT)) ("2" (ASSERT))))
         ("2"
            (LEMMA "trans_mutex_end")
            (INST -1 "lower" "et - lower_dur")
            (("1" (ASSERT)) ("2" (ASSERT)))))))
   ("2" (SKOSIMP*) (ASTRAL-EXPAND 1))
   ("3" (SKOSIMP*) (ASTRAL-EXPAND))
   ("4"
      (EXPAND "nonempty?")
      (EXPAND "empty?")
      (EXPAND "member")
      (LEMMA "exists_end1")
      (INST -1 "lower" "lower_dur + t2!1 + up_dur")
      (DELETE -4)
      (EXPAND "Duration")
      (INST -1 "t1!2")
      (ASSERT)
      (SKOSIMP*)
         (EXPAND "Base_Trans")
         (INST -4 "t3!1")
         (ASSERT))
      ("5" (SKOSIMP*) (ASTRAL-EXPAND 1))
      ("6" (SKOSIMP*) (ASTRAL-EXPAND 1))))
   ("5"
      (ASSERT)
      (ASTRAL-EXPAND-CLAUSE -1)
      (FLATTEN)
      (INST -1 "V1!1")
      (ASTRAL-EXPAND-ALL -22)
      (FLATTEN)
      (INST -23 "t1!2")
      (EXPAND "Duration")
      (ASSERT))))
   ("2"
      (EXPAND "Duration")
      (TYPEPRED "lower_dur" "up_dur")
      (ASTRAL-EXPAND  (-1 -2))
      (ASSERT))))
```

```
{-1}    Exit(up, (t2!1 + up_dur))
{-2}    Entry(up, t2!1)
[-3]    ct - up_dur < t2!1
[-4]    t2!1 < ct
[-5]    Fired(up, t2!1)
[-6]    ct ≥ 0
[-7]    ct ≤ T1!1
[-8]    Change1(i_sensor__train_in_r(V1!1), const(ct))(T1!1)
[-9]    i_sensor__train_in_r(V1!1)(T1!1)
[-10]   T1!1 - ct ≥ dist_r_to_i / max_speed - response_time
 |-------
{1}     Enabled(lower, t2!1 + Duration(up))
[2]     position(T1!1) = lowered

;; lower fires immediately at the end of up (2/2)

            ("2"
               (ASTRAL-EXPAND-CLAUSE 1)
               (ASTRAL-EXPAND-CLAUSE -1)
               (ASSERT)
               (INST 1 "V1!1")
               (ASTRAL-EXPAND-ALL -8)
               (FLATTEN)
               (INST -9 "t2!1 + up_dur")
               (ASSERT)
               (LEMMA "local_axiom")
               (ASTRAL-EXPAND -1)
               (FLATTEN)
               (TYPEPRED "down_dur" "lower_dur" "lower_time"
                  "raise_dur" "response_time")
               (ASTRAL-EXPAND (-1 -2 -3 -4 -5))
               (ASSERT))))))

;; split proof into cases based on operating environment and local state (1/2)

      ("2" (FLATTEN) (ASSERT))))))
   ("2" (SKOSIMP*) (ASTRAL-EXPAND))
   ("3"
      (EXPAND "nonempty?")
      (EXPAND "empty?")
      (EXPAND "member")
      (DELETE -3)
      (LEMMA "exists_change1[boolean]")
      (INST -1 "i_sensor__train_in_r(V1!1)" "0" "T1!1")
      (LEMMA "i_initial_state")
      (ASTRAL-EXPAND-CLAUSE -1)
      (FLATTEN)
      (INST -1 "V1!1")
      (DELETE -2)
      (TYPEPRED "V1!1")
      (ASTRAL-EXPAND -1)
      (ASSERT)
      (DELETE -1)
      (SKOSIMP*)
      (INST -4 "t2!1")
      (ASSERT)
      (ASTRAL-EXPAND-ALL))
   ("4" (SKOSIMP*) (ASTRAL-EXPAND))))
```

# D.5. PVS Safety Property Proof (described in 10.8.2.6)

```
 |-------
{1}   (FORALL (TR1: transition): DELTA < Duration(TR1))
          AND (FORALL (T1: time): T1 ≤ T0 IMPLIES Invariant(T1))
      IMPLIES
      (FORALL (T1: time):
         T0 < T1 AND T1 < T0 + DELTA IMPLIES s_Invariant(2)(T1))

;; exit_i fires at now - exit_dur (1/2)

("")
   (SKOSIMP*)
   (DELETE -1 -2)
   (ASTRAL-EXPAND-CLAUSE 1)
   (SKOSIMP*)
   (ASSERT)
   (CHANGE-FIRE -3 "train_in_r" "T1!1")
   (("1"
      (ASSERT)

{-1}  Exit(exit_i, T1!1)
{-2}  Entry(exit_i, T1!1 - Duration(exit_i))
[-3]  T1!1 - Duration(exit_i) ≥ 0
[-4]  Fired(exit_i, T1!1 - Duration(exit_i))
[-5]  T0 < T1!1
[-6]  T1!1 < DELTA + T0
[-7]  t1!1 < T1!1
{-8}  FORALL (t3: time): t1!1 ≤ t3 AND t3 < T1!1 IMPLIES train_in_r(t3)
[-9]  FORALL (t2: time): T1!1 ≤ t2 AND t2 ≤ T1!1 IMPLIES NOT train_in_r(t2)
[-10] T1!1 - (dist_i_to_out + dist_r_to_i) / max_speed + response_time ≤ V1!1
[-11] V1!1 < T1!1
 |-------
{1}   train_in_r(T1!1 - Duration(exit_i)) = FALSE
[2]   train_in_r(T1!1)
[3]   train_in_r(V1!1)

;; achieve contradiction between fact that enter_r fires and entry of exit_i (1/3)

      (DELETE -7 -8 -9 1)
      (ASTRAL-EXPAND-CLAUSE)
      (FLATTEN)
      (EXPAND "Start1")
      (TYPEPRED "choose!  (t2: time):
          t2 ≤  (T1!1 - exit_dur)
          AND Start1(enter_r, const(t2))(T1!1 - exit_dur)")
      (("1"
         (NAME "ts" " choose!  (t2: time):
             t2 ≤ T1!1 - exit_dur
             AND Start1(enter_r, const(t2))(T1!1 - exit_dur)")
         (REPLACE -1)
         (DELETE -1)
         (CASE "V1!1 ≥ T1!1 - exit_dur")
         (("1"
            (LEMMA "vars_no_change")
            (INST -1 "T1!1 - exit_dur" "V1!1")
            (ASSERT)
            (SPLIT -1)
            (("1" (INST -1 "V1!1") (ASSERT) (ASTRAL-EXPAND-CLAUSE -1))
            ("2"
               (SKOSIMP*)
               (LEMMA "trans_mutex_end")
               (INST -1 "exit_i" "T1!1 - exit_dur")
               (ASSERT)
               (INST -1 "tr2!1" "t2!1")
               (ASSERT)
               (EXPAND "Duration" 1 2)
               (ASSERT))))

[-1]  ts ≥ 0
[-2]  ts ≤ (T1!1 - exit_dur)
[-3]  Start1(enter_r, const(ts))(T1!1 - exit_dur)
[-4]  train_in_r(T1!1 - exit_dur)
[-5]  T1!1 - ts - exit_dur
      ≥ (dist_i_to_out + dist_r_to_i) / min_speed - exit_dur
[-6]  T1!1 - exit_dur ≥ 0
[-7]  Fired(exit_i, T1!1 - exit_dur)
[-8]  T0 < T1!1
[-9]  T1!1 < DELTA + T0
[-10] T1!1 - (dist_i_to_out + dist_r_to_i) / max_speed + response_time) ≤ V1!1
[-11] V1!1 < T1!1
 |-------
{1}   V1!1 ≥ T1!1 - exit_dur
[2]   train_in_r(T1!1)
[3]   train_in_r(V1!1)

;; enter_r fires

      ("2"
         (LEMMA "not_vnc_vc")
         (INST -1 "V1!1" "T1!1 - exit_dur")
         (ASSERT)
         (SPLIT -1)
         (("1"
            (SKOSIMP* -1)
            (EXPAND "Var_Changes")
            (INST -5 "t1!2")
            (ASSERT)
            (ASTRAL-EXPAND-CLAUSE -5)
            (CHANGE-FIRE -4 "train_in_r" "t1!2")
            (ASSERT)

{-1}  Exit(enter_r, t1!2)
{-2}  Entry(enter_r, t1!2 - Duration(enter_r))
[-3]  train_in_r(t1!2 - Duration(enter_r)) ≠ train_in_r(t1!2)
[-4]  t1!2 - Duration(enter_r) ≥ 0
[-5]  Fired(enter_r, t1!2 - Duration(enter_r))
[-6]  t1!2 ≥ 0
[-7]  V1!1 < t1!2
[-8]  t1!2 ≤ T1!1 - exit_dur
[-9]  t1!3 < t1!2
[-10] FORALL (t3: time):
         t1!3 ≤ t3 AND t3 < t1!2 IMPLIES NOT train_in_r(t3) = train_in_r(t1!2)
[-11] FORALL (t2: time):
         t1!2 ≤ t2 AND t2 ≤ t1!2 IMPLIES train_in_r(t2) = train_in_r(t1!2)
[-12] train_in_r(t1!2)
[-13] ts ≥ 0
[-14] ts ≤ T1!1 - exit_dur
[-15] Start1(enter_r, const(ts))(T1!1 - exit_dur)
[-16] train_in_r(T1!1 - exit_dur)
[-17] T1!1 - exit_dur - ts
      ≥ (dist_i_to_out + dist_r_to_i) / min_speed - exit_dur
[-18] T1!1 - exit_dur ≥ 0
[-19] Fired(exit_i, T1!1 - exit_dur)
[-20] T0 < T1!1
[-21] T1!1 < DELTA + T0
[-22] T1!1 - (dist_i_to_out + dist_r_to_i) / max_speed + response_time ≤ V1!1
[-23] V1!1 < T1!1
 |-------
[1]   V1!1 ≥ T1!1 - exit_dur
[2]   train_in_r(T1!1)
[3]   train_in_r(V1!1)

;; achieve contradiction between fact that enter_r fires and entry of exit_i (2/3)

               (ASTRAL-EXPAND-ALL -15)
               (FLATTEN)
               (SKOSIMP*)
               (REPLACE -14)
               (DELETE -14)
               (EXPAND "Duration")
               (INST -15 "t1!2 - enter_dur")
               (SPLIT -15)
               (("1" (INST -1 "enter_r") (ASSERT))
               ("2"
                  (LEMMA "global_axiom")
                  (LEMMA "local_axiom")
                  (TYPEPRED "min_speed" "max_speed" "dist_i_to_out"
                     "dist_r_to_i")
                  (ASTRAL-EXPAND)
                  (FLATTEN)
                  (CASE "(dist_i_to_out + dist_r_to_i) / min_speed
                     ≥ (dist_i_to_out+dist_r_to_i) / max_speed")
                  (("1" (ASSERT))
                  ("2"
                     (LEMMA "both_sides_div_pos_ge2")
                     (INST -1 "min_speed" "max_speed"
                        "dist_i_to_out + dist_r_to_i")
                     (("1" (ASSERT)) ("2" (ASSERT)) ("3" (ASSERT))
                     ("4" (ASSERT))))
```

```
                ("3" (ASSERT)) ("4" (ASSERT))))
              ("3"
                (TYPEPRED "enter_dur" "exit_dur")
                (ASTRAL-EXPAND (-1 -2))
                (ASSERT))))

;; enter_r fires (2/2)

        ("2" (ASTRAL-EXPAND-CLAUSE -1))))))

;; achieve contradiction between fact that enter_r fires and entry of exit_i (3/3)

      ("2" (SKOSIMP*) (ASTRAL-EXPAND 1) (ASSERT))
      ("3" (SKOSIMP*) (ASTRAL-EXPAND 1))
      ("4"
        (EXPAND "nonempty?")
        (EXPAND "empty?")
        (EXPAND "member")
        (LEMMA "not_vnc_vc")
        (INST -1 "0" "T1!1 - exit_dur")
        (LEMMA "initial_state")
        (ASTRAL-EXPAND-CLAUSE -1)
        (CASE "T1!1-exit_dur = 0")
        (("1" (ASSERT))
        ("2"
          (ASSERT)
          (SPLIT -1)
          (("1"
            (SKOSIMP*)
            (INST -4 "t1!2")
            (ASSERT)
            (ASTRAL-EXPAND-CLAUSE -4)
            (EXPAND "Var_Changes" -3)
            (CHANGE-FIRE -3 "train_in_r" "t1!2")
            (ASSERT)
            (EXPAND "Duration")
            (LEMMA "exists_start1")
            (INST -1 "enter_r" "t1!2 - enter_dur")
            (ASSERT)
            (INST -1 "T1!1 - exit_dur")
            (SPLIT -1)
            (("1"
              (SKOSIMP*)
              (INST -14 "t3!1")
              (EXPAND "Base_Trans")
              (ASSERT))
             ("2" (TYPEPRED "enter_dur") (ASTRAL-EXPAND -1) (ASSERT))))
           ("2" (ASTRAL-EXPAND-CLAUSE -1))))))
      ("5" (SKOSIMP*) (ASTRAL-EXPAND 1) (ASSERT))
      ("6" (SKOSIMP*) (ASTRAL-EXPAND 1))))

;; exit_i fires at now - exit_dur (2/2)

    ("2" (ASSERT))))
```

# Appendix E

# ASTRAL Grammar

## E.1. Tokens

| Token | Associated Regular Expression |
|---|---|
| ALL_TRANS | ”ALL_TRANSITIONS” |
| ALT | ”ALT” |
| AND | ”&” |
| ANY_SUBSET | ”ANY_SUBSET” |
| ARRAY | ”ARRAY” |
| AS | “AS” |
| ASSUMPTIONS | “ASSUMPTIONS” |
| AXIOM | “AXIOM” |
| BECOMES | ”BECOMES” |
| BEFORE | ”BEFORE” |
| BOOLEAN | ”BOOLEAN” |
| CALL | ”CALL” |
| CHANGE | ”CHANGE” |
| CLOSECURLY | ”}” |
| CLOSEROUND | ”)” |
| CLOSESQUARE | ”]” |
| COLON | ”:” |
| COMMA | ”,” |
| COMPOSITION | “COMPOSITION” |
| CONCAT | ”CONCAT” |
| CONSTANT | “CONSTANT” |
| CONSTRAINT | “CONSTRAINT” |
| CONTAINED_IN | ”CONTAINED_IN” |
| CONTAINS | ”CONTAINS” |
| DECIMALPART | ”.”{DIGIT}+ |
| DEFINE | “DEFINE” |
| DIGIT | [0-9] |
| DIV | ”DIV” |
| DIVIDE | ”/” |
| DO | ”DO” |
| DOT | ”.” |
| ELIGIBLE_TRANS | ”ELIGIBLE_TRANSITIONS” |
| ELSE | ”ELSE” |

| | |
|---|---|
| EMPTY | ”EMPTY” |
| ENABLED_TRANS | ”ENABLED_TRANSITIONS” |
| END | ”END” |
| ENTRY | “ENTRY” |
| ENVIRONMENT | “ENVIRONMENT” |
| EQ | ”=” |
| EQEQ | ”==” |
| EXCEPT | “EXCEPT” |
| EXISTS | ”EXISTS” |
| EXIT | “EXIT” |
| EXPORT | “EXPORT” |
| FALSE_ | ”FALSE” |
| FI | ”FI” |
| FIRSTCHAR | {LETTER}|”_” |
| FORALL | ”FORALL” |
| FURTHER | “FURTHER” |
| GENERATION | “GENERATION” |
| GLOBAL | “GLOBAL” |
| GT | ”>” |
| GTE | ”>=” |
| ID | ”ID” |
| IDENTIFIER | {FIRSTCHAR}{OTHERCHAR}* |
| IDTYPE | ”IDTYPE” |
| IF | ”IF” |
| IFF | ”<->” |
| IMPL_OR | ”OR” |
| IMPLEMENTATION | “IMPLEMENTATION” |
| IMPLIES | ”->” |
| IMPORT | “IMPORT” |
| IMPORTED | “IMPORTED” |
| INITIAL | “INITIAL” |
| INTEGER | ”INTEGER” |
| INTEGER_CONST | {INTEGERPART} |
| INTEGERPART | {DIGIT}+ |
| INTERSECT | ”INTERSECT” |
| INVARIANT | “INVARIANT” |
| IS | ”IS” |
| ISIN | ”ISIN” |
| LETTER | [A-Za-z] |
| LEVEL | “LEVEL” |
| LIST | ”LIST” |
| LIST_LEN | ”LIST_LEN” |
| LISTDEF | ”LISTDEF” |
| LT | ”<” |
| LTE | ”<=” |
| MINUS | ”-“ |
| MOD | ”MOD” |

| | |
|---|---|
| NAND | ”~&” |
| NCONTAINED_IN | ”~CONTAINED_IN” |
| NCONTAINS | ”~CONTAINS” |
| NEQ | ”~=” |
| NGT | ”~>” |
| NGTE | ”~>=” |
| NIFF | ”~<->” |
| NIL | ”NIL” |
| NIMPLIES | ”~->” |
| NISIN | ”~ISIN” |
| NLT | ”~<” |
| NLTE | ”~<=” |
| NOCHANGE | ”NOCHANGE” |
| NOR | ”~\|” |
| NOT | ”~” |
| NOW | ”NOW” |
| NSUBSET | ”~SUBSET” |
| NSUPERSET | ”~SUPERSET” |
| OD | ”OD” |
| OF | ”OF” |
| OPENCURLY | ”{” |
| OPENROUND | ”(“ |
| OPENSQUARE | ”[” |
| OR | ”\|” |
| OTHERCHAR | {LETTER}\|{DIGIT}\|”_” |
| PAST | ”PAST” |
| PLUS | ”+” |
| POUND | “#” |
| PRIME | ”’” |
| PROCESS | “PROCESS” |
| PROCESSES | “PROCESSES” |
| REAL | ”REAL” |
| REAL_CONST | {INTEGERPART}{DECIMALPART} |
| REFINEMENT | “REFINEMENT” |
| REFINES | “REFINES” |
| SCHEDULE | “SCHEDULE” |
| SELECTION | “SELECTION” |
| SELF | ”SELF” |
| SET | ”SET” |
| SET_DIFF | ”SET_DIFF” |
| SET_SIZE | ”SET_SIZE” |
| SETDEF | ”SETDEF” |
| SPECIFICATION | “SPECIFICATION” |
| START | ”START” |
| STRUCTURE | ”STRUCTURE” |
| SUBSET | ”SUBSET” |
| SUBTYPE | ”SUBTYPE” |

| | |
|---|---|
| SUPERSET | ”SUPERSET” |
| SYM_DIFF | ”SYM_DIFF” |
| THEN | ”THEN” |
| TIME | ”TIME” |
| TIMES | ”*” |
| TRANSITION | “TRANSITION” |
| TRUE_ | ”TRUE” |
| TYPE | “TYPE” |
| TYPEDEF | ”TYPEDEF” |
| UNION | ”UNION” |
| UNIQUE | ”UNIQUE” |
| VARIABLE | “VARIABLE” |
| WHEN | ”WHEN” |

## E.2. Associativity and Precedence

| Associativity | Precedence (high to low) |
|---|---|
| left | DOT |
| nonassociative | LISTDEF, SETDEF |
| nonassociative | LIST_LEN, SET_SIZE |
| left | CONCAT |
| nonassociative | ANY_SUBSET |
| nonassociative | COLL_UNION, COLL_INTERSECT, COLL_SYM_DIFF |
| left | INTERSECT, SET_DIFF, SYM_DIFF |
| left | UNION |
| right | UNARY_MINUS |
| left | TIMES, DIVIDE, MOD, DIV |
| left | PLUS, MINUS |
| nonassociative | CONTAINED_IN, SUBSET, SUPERSET, CONTAINS, NCONTAINED_IN, NSUBSET, NSUPERSET, NCONTAINS |
| right | NOT |
| left | ISIN, NISIN |
| nonassociative | EQ, NEQ, GT, NGT, GTE, NGTE, LT, NLT, LTE, NLTE |
| nonassociative | BECOMES |
| left | AND, NAND |
| left | OR, NOR |
| left | IMPLIES, NIMPLIES |
| left | IFF, NIFF |
| left | ALT |
| left | WHEN |
| left | BEFORE |
| left | IMPL_OR |

## E.3. Compositions

```
comp:
        COMPOSITION OF id_list AS IDENTIFIER
        PROCESSES processes_decl_list
        type_clause
        axiom_clause
        constant_clause
        define_clause
        cg_decl_list
        spec_list
        END IDENTIFIER

id_list:
        IDENTIFIER
    |   id_list COMMA IDENTIFIER
```

## E.4. Specifications

```
spec_list:
        spec
    |   spec_list spec

spec:
        SPECIFICATION IDENTIFIER
        global_spec
        process_spec_list
        END IDENTIFIER
```

### E.4.1. Global Specifications

```
global_spec:
        GLOBAL SPECIFICATION IDENTIFIER
        PROCESSES processes_decl_list
        type_clause
        axiom_clause
        constant_clause
        define_clause
        environment_clause
        invariant_clause
        schedule_clause
        END IDENTIFIER
```

### E.4.2. Process Specifications

```
process_spec_list:
        process_spec
    |   process_spec_list process_spec
```

```
process_spec:
        PROCESS SPECIFICATION IDENTIFIER
        top_level
        opt_level_list
        END IDENTIFIER
```

### E.4.3. Level Specifications

```
top_level:
        LEVEL IDENTIFIER
        import_clause
        export_clause
        environment_clause
        impvar_clause
        level_decl
        END IDENTIFIER

opt_level_list:
        /* empty */
    |   lower_level_list

lower_level_list:
        lower_level
    |   lower_level_list lower_level

lower_level:
        LEVEL IDENTIFIER REFINES IDENTIFIER
        level_decl
        IMPLEMENTATION impl_decl_list
        END IDENTIFIER

level_decl:
        type_clause
        axiom_clause
        variable_clause
        constant_clause
        define_clause
        initial_clause
        invariant_clause
        constraint_clause
        schedule_clause
        further_clause
        trans_decl_list
```

## E.5. Transitions

```
trans_decl_list:
        trans_decl
    |   trans_decl_list trans_decl
```

```
trans_decl:
        TRANSITION trheading
        ENTRY IDENTIFIER OPENSQUARE TIME COLON duration CLOSESQUARE wff
        EXIT wff
        opt_except_list

trheading:
        IDENTIFIER
    |   IDENTIFIER OPENROUND id_type_list CLOSEROUND

id_type_list:
        id_list COLON any_type
    |   id_type_list COMMA id_list COLON any_type

id_list:
        IDENTIFIER
    |   id_list COMMA IDENTIFIER

any_type:
        INTEGER
    |   REAL
    |   BOOLEAN
    |   TIME
    |   ID
    |   IDENTIFIER

duration:
        IDENTIFIER
    |   INTEGER_CONST
    |   REAL_CONST

opt_except_list:
        /* empty */
    |   except_decl_list

except_decl_list:
        except_decl
    |   except_decl_list except_decl

except_decl:
        EXCEPT IDENTIFIER OPENSQUARE TIME COLON duration
            CLOSESQUARE wff
        EXIT wff
```

## E.6. Clauses

### E.6.1. Call Generation Clauses

```
cg_decl_list:
        CALL GENERATION cg_name wff
    |   cg_decl_list CALL GENERATION cg_name wff

cg_name:
        IDENTIFIER DOT IDENTIFIER
```

### E.6.2. Constant and Variable Clauses

```
constant_clause:
        /* empty */
    |   CONSTANT const_var_decl_list

variable_clause:
        /* empty */
    |   VARIABLE const_var_decl_list

const_var_decl_list:
        const_var_decl
    |   const_var_decl_list COMMA const_var_decl

const_var_decl:
        const_var_list COLON any_type

const_var_list:
        const_var
    |   const_var_list COMMA const_var

const_var:
        IDENTIFIER
    |   IDENTIFIER OPENROUND any_type_list CLOSEROUND

any_type_list:
        any_type
    |   any_type_list COMMA any_type

any_type:
        INTEGER
    |   REAL
    |   BOOLEAN
    |   TIME
    |   ID
    |   IDENTIFIER
```

### E.6.3. Define Clauses

```
define_clause:
        /* empty */
    |   DEFINE define_decl_list

define_decl_list:
        define_decl
    |   define_decl_list COMMA define_decl

define_decl:
        IDENTIFIER COLON any_type EQEQ wff
    |   IDENTIFIER OPENROUND id_type_list CLOSEROUND COLON any_type
            EQEQ wff

any_type:
        INTEGER
    |   REAL
    |   BOOLEAN
    |   TIME
    |   ID
    |   IDENTIFIER

id_type_list:
        id_list COLON any_type
    |   id_type_list COMMA id_list COLON any_type

id_list:
        IDENTIFIER
    |   id_list COMMA IDENTIFIER
```

### E.6.4. Formula Clauses

```
axiom_clause:
        /* empty */
    |   AXIOM wff

environment_clause:
        /* empty */
    |   ENVIRONMENT wff

impvar_clause:
        /* empty */
    |   IMPORTED VARIABLE wff

initial_clause:
        /* empty */
    |   INITIAL wff
```

```
invariant_clause:
        /* empty */
    |   INVARIANT wff

constraint_clause:
        /* empty */
    |   CONSTRAINT wff

schedule_clause:
        /* empty */
    |   SCHEDULE wff
```

### E.6.5. Further Assumptions Clauses

```
further_clause:
        /* empty */
    |   further_decl_list

further_decl_list:
        further_decl
    |   further_decl_list further_decl

further_decl:
        FURTHER ASSUMPTIONS POUND INTEGER_CONST
        furenv_clause
        furproc_clause

furenv_clause:
        /* empty */
    |   FURTHER ENVIRONMENT wff

furproc_clause:
        /* empty */
    |   FURTHER PROCESS ASSUMPTIONS constref_clause trsel_clause

constref_clause:
        /* empty */
    |   CONSTANT REFINEMENT wff

trsel_clause:
        /* empty */
    |   TRANSITION SELECTION trsel_decl_list

trsel_decl_list:
        trsel_decl
    |   trsel_decl_list COMMA trsel_decl
```

```
trsel_decl:
        et_decl_list AND wff IMPLIES ELIGIBLE_TRANS EQ basic_set
    |   et_decl_list AND wff IMPLIES ELIGIBLE_TRANS EQ basic_set
            INTERSECT ENABLED_TRANS

et_decl_list:
        et_decl
    |   et_decl_list AND et_decl_list
    |   et_decl_list OR et_decl_list
    |   OPENROUND et_decl_list CLOSEROUND

et_decl:
        ENABLED_TRANS SUBSET basic_set
    |   ENABLED_TRANS NSUBSET basic_set
    |   ENABLED_TRANS CONTAINED_IN basic_set
    |   ENABLED_TRANS NCONTAINED_IN basic_set
    |   ENABLED_TRANS SUPERSET trans_set
    |   ENABLED_TRANS NSUPERSET trans_set
    |   ENABLED_TRANS CONTAINS trans_set
    |   ENABLED_TRANS NCONTAINS trans_set
    |   ENABLED_TRANS EQ basic_set
    |   ENABLED_TRANS NEQ basic_set

trans_set:
        basic_set
    |   ANY_SUBSET OPENROUND basic_set CLOSEROUND

basic_set:
        ALL_TRANS
    |   OPENCURLY id_list CLOSECURLY
    |   ALL_TRANS SET_DIFF OPENCURLY id_list CLOSECURLY
```

### E.6.6. Implementation Clauses

```
impl_decl_list:
        impl_decl
    |   impl_decl_list COMMA impl_decl

impl_decl:
        lhs_impl_id EQEQ DO selseq OD
    |   lhs_impl_id EQEQ WHEN wff DO selseq OD
    |   lhs_impl_id EQEQ wff

lhs_impl_id:
        IDENTIFIER opt_id_list
    |   IDENTIFIER DOT INTEGER_CONST opt_id_list

opt_id_list:
        /* empty */
    |   OPENROUND id_list CLOSEROUND
```

```
id_list:
        IDENTIFIER
    |   id_list COMMA IDENTIFIER

selseq:
        rhs_impl_id
    |   selseq WHEN wff
    |   selseq IMPL_OR selseq
    |   selseq BEFORE selseq
    |   OPENROUND selseq CLOSEROUND

rhs_impl_id:
        IDENTIFIER opt_wff_list
    |   IDENTIFIER DOT INTEGER_CONST opt_wff_list

opt_wff_list:
        /* empty */
    |   OPENROUND wff_list CLOSEROUND

wff_list:
        wff
    |   wff_list COMMA wff
```

### E.6.7. Import and Export Clauses

```
import_clause:
        /* empty */
    |   IMPORT id_dot_list

id_dot_list:
        IDENTIFIER
    |   IDENTIFIER opt_n DOT IDENTIFIER
    |   id_dot_list COMMA IDENTIFIER
    |   id_dot_list COMMA IDENTIFIER opt_n DOT IDENTIFIER

opt_n:
        /* empty */
    |   OPENSQUARE INTEGER_CONST CLOSESQUARE

export_clause:
        /* empty */
    |   EXPORT id_list

id_list:
        IDENTIFIER
    |   id_list COMMA IDENTIFIER
```

### E.6.8. Processes Clauses

```
processes_decl_list:
        processes_decl
    |   processes_decl_list COMMA processes_decl

processes_decl:
        id_list COLON IDENTIFIER
    |   id_list COLON ARRAY OPENSQUARE id_integer CLOSESQUARE
            OF IDENTIFIER
    |   id_list COLON ARRAY OPENSQUARE id_integer DOT DOT id_integer
            CLOSESQUARE OF IDENTIFIER

id_list:
        IDENTIFIER
    |   id_list COMMA IDENTIFIER

id_integer:
        IDENTIFIER
    |   INTEGER_CONST
```

### E.6.9. Type Clauses

```
type_clause:
        /* empty */
    |   TYPE type_decl_list

type_decl_list:
        type_decl
    |   type_decl_list COMMA type_decl

type_decl:
        IDENTIFIER
    |   IDENTIFIER colon_is any_type
    |   IDENTIFIER SUBTYPE any_type
    |   IDENTIFIER colon_is LIST OF any_type
    |   IDENTIFIER colon_is SET OF any_type
    |   IDENTIFIER colon_is OPENROUND id_list CLOSEROUND
    |   IDENTIFIER colon_is STRUCTURE OF OPENROUND id_type_list CLOSEROUND
    |   IDENTIFIER colon_is TYPEDEF IDENTIFIER colon_is any_type
            OPENROUND wff CLOSEROUND

colon_is:
        COLON
    |   IS

any_type:
        INTEGER
    |   REAL
    |   BOOLEAN
```

| TIME
| ID
| IDENTIFIER

id_list:
IDENTIFIER
| id_list COMMA IDENTIFIER

id_type_list:
parm_id_list COLON any_type
| id_type_list COMMA parm_id_list COLON any_type

parm_id_list:
IDENTIFIER
| IDENTIFIER OPENROUND any_type_list CLOSEROUND
| parm_id_list COMMA IDENTIFIER
| parm_id_list COMMA IDENTIFIER OPENROUND any_type_list CLOSEROUND

any_type_list:
any_type
| any_type_list COMMA any_type

## E.7. Well-Formed Formulas

wff:
wff IFF wff
| wff NIFF wff
| wff IMPLIES wff
| wff NIMPLIES wff
| wff OR wff
| wff NOR wff
| wff AND wff
| wff NAND wff
| wff EQ wff
| wff NEQ wff
| wff LT wff
| wff NLT wff
| wff LTE wff
| wff NLTE wff
| wff GT wff
| wff NGT wff
| wff GTE wff
| wff NGTE wff
| wff CONTAINED_IN wff
| wff NCONTAINED_IN wff
| wff SUBSET wff
| wff NSUBSET wff
| wff CONTAINS wff
| wff NCONTAINS wff

```
    |   wff SUPERSET wff
    |   wff NSUPERSET wff
    |   wff ISIN wff
    |   wff NISIN wff
    |   wff UNION wff
    |   wff INTERSECT wff
    |   wff SET_DIFF wff
    |   wff SYM_DIFF wff
    |   wff CONCAT wff
    |   wff TIMES wff
    |   wff DIVIDE wff
    |   wff PLUS wff
    |   wff MINUS wff
    |   wff MOD wff
    |   wff DIV wff
    |   wff ALT wff
    |   NOT wff
    |   MINUS wff %prec UNARY_MINUS
    |   UNION wff %prec COLL_UNION
    |   INTERSECT wff %prec COLL_INTERSECT
    |   SYM_DIFF wff %prec COLL_SYM_DIFF
    |   SET_SIZE wff
    |   LIST_LEN wff
    |   IDTYPE OPENROUND wff CLOSEROUND
    |   LISTDEF OPENROUND wff_list CLOSEROUND
    |   OPENCURLY wff_list CLOSECURLY
    |   OPENCURLY SETDEF IDENTIFIER COLON any_type
    |   TRUE_
    |   FALSE_
    |   EMPTY
    |   NIL
    |   IF wff THEN wff ELSE wff FI
    |   OPENROUND wff CLOSEROUND
    |   becomes
    |   value
    |   self
    |   id_combo
    |   dot
    |   start_end_call
    |   change
    |   past
    |   nochange
    |   quantification

wff_list:
        wff
    |   wff_list COMMA wff
```

```
any_type:
        INTEGER
    |   REAL
    |   BOOLEAN
    |   TIME
    |   ID
    |   IDENTIFIER

becomes:
        IDENTIFIER OPENROUND wff_list CLOSEROUND BECOMES wff
    |   IDENTIFIER comp_spec BECOMES wff
    |   IDENTIFIER OPENROUND wff_list CLOSEROUND comp_spec BECOMES wff

value:
        REAL_CONST
    |   INTEGER_CONST
    |   NOW

self:
        SELF
    |   SELF DOT id_combo
    |   SELF DOT dot
    |   SELF DOT start_end_call

id_combo:
        IDENTIFIER
    |   IDENTIFIER PRIME
    |   IDENTIFIER OPENROUND wff_list CLOSEROUND
    |   IDENTIFIER PRIME OPENROUND wff_list CLOSEROUND
    |   IDENTIFIER comp_spec
    |   IDENTIFIER PRIME comp_spec
    |   IDENTIFIER OPENROUND wff_list CLOSEROUND comp_spec
    |   IDENTIFIER PRIME OPENROUND wff_list CLOSEROUND comp_spec

comp_spec:
        OPENSQUARE wff CLOSESQUARE
    |   OPENSQUARE wff CLOSESQUARE comp_spec

dot:
        id_combo DOT id_combo
    |   id_combo DOT start_end_call
    |   id_combo DOT dot

start_end_call:
        start_end_call_id opt_n OPENROUND IDENTIFIER opt_wff_list
            opt_wff CLOSEROUND
```

```
start_end_call_id:
        START
    |   END
    |   CALL

opt_n:
        /* empty */
    |   OPENSQUARE wff CLOSESQUARE

opt_wff_list:
        /* empty */
    |   OPENROUND wff_list CLOSEROUND

opt_wff:
        /* empty */
    |   COMMA wff

change:
        CHANGE opt_n OPENROUND wff opt_wff CLOSEROUND

past:
        PAST OPENROUND wff COMMA wff CLOSEROUND

nochange:
        NOCHANGE OPENROUND parm_id_list CLOSEROUND

parm_id_list:
        parm_id
    |   parm_id_list COMMA parm_id

parm_id:
        IDENTIFIER
    |   IDENTIFIER OPENROUND id_list CLOSEROUND

id_list:
        IDENTIFIER
    |   id_list COMMA IDENTIFIER

quantification:
        quantifier var_decl_list OPENROUND wff CLOSEROUND

quantifier:
        FORALL
    |   EXISTS
    |   UNIQUE

var_decl_list:
        id_list COLON any_type
    |   var_decl_list COMMA id_list COLON any_type
```

# Appendix F

# PVS Translation of Bakery Algorithm

## F.1. Global Theory

```
global: THEORY

BEGIN

astral_lib: LIBRARY = "/fs/rsl/pkgs/kolano/astral/astral9/lib/pvs"

IMPORTING astral_lib@astral_defs

process: TYPE = {proc}

id: NONEMPTY_TYPE

nonneg_int: TYPE = {i: integer | ((const(i)) >= (const(0)))(0)}

pos_int: TYPE = {i: integer | ((const(i)) > (const(0)))(0)}

n_procs: pos_int

procs: [{I1: int | I1 >= 1 AND I1 <= n_procs} -> id]

Id_Type(ID1: [time -> id])(T1: time): process = proc

procs_int: TYPE = {i: integer | (((const(1)) <= (const(i))) AND ((const(i)) <=
(const(n_procs))))(0)}

nonneg_real: TYPE = {r: real | ((const(r)) >= (const(0)))(0)}

pos_real: TYPE = {r: real | ((const(r)) > (const(0)))(0)}

i_parameter: TYPE = [# DUMMY: int #]

i_undef_parm: i_parameter

i_proc__choosing: [id -> [time -> boolean]]

i_proc__number: [id -> [time -> nonneg_int]]

i_proc__in_critical: [id -> [time -> boolean]]

exec_time: pos_real

i_Var_Changes(T1: time): bool =
        (EXISTS (PID1: id): Change1(i_proc__choosing(PID1), const(T1))(T1)) OR
        (EXISTS (PID1: id): Change1(i_proc__number(PID1), const(T1))(T1)) OR
        (EXISTS (PID1: id): Change1(i_proc__in_critical(PID1), const(T1))(T1))

global_axiom: AXIOM
        (const(TRUE))(0)

id_domain: AXIOM
        (FORALL (ID1: id):
                (EXISTS (I1: {K1: int | K1 >= 1 AND K1 <= n_procs}):
                        ID1 = procs(I1)))

id_unique: AXIOM
        (FORALL (I1, J1: {K1: int | K1 >= 1 AND K1 <= n_procs}):
                procs(I1) = procs(J1) IMPLIES
                        I1 = J1) AND
        TRUE

i_initial_state: AXIOM
        (LAMBDA (T1: time): (FA! (PID1: [time -> id]):
                Id_Type(PID1) = const(proc) IMPLIES
                ((((const(TRUE)) IMPLIES (NOT (i_proc__choosing(PID1(T1)))))
AND ((NOT ((i_proc__number(PID1(T1))) = (const(0)))) IMPLIES
(const(FALSE)))) AND ((const(TRUE)) IMPLIES (NOT
(i_proc__in_critical(PID1(T1)))))))(T1))(0) AND TRUE

END global
```

## F.2. Global_INV Theory

```
global_INV: THEORY

BEGIN

IMPORTING global

DELTA: posreal
T0: time
T1: VAR time

i_Invariant(T1: time): bool =
        (FA! (PID1: [time -> id]):
                Id_Type(PID1) = const(proc) IMPLIES
                ((((((i_proc__in_critical(PID1(T1))) IMPLIES (NOT
(i_proc__choosing(PID1(T1))))) AND ((i_proc__in_critical(PID1(T1))) IMPLIES
((i_proc__number(PID1(T1))) /= (const(0))))) AND ((FA! (i: [time -> procs_int]):
((Change1(i_proc__number(PID1(T1)), now)) AND ((i_proc__number(PID1(T1)))
/= (const(0)))) IMPLIES ((i_proc__number(PID1(T1))) >= (Past(((LAMBDA (T1:
time): i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))) +
(const(1)), (now) - (const(exec_time)))))))) AND
(((Change1(i_proc__number(PID1(T1)), now)) AND ((i_proc__number(PID1(T1)))
= (const(0)))) IMPLIES (NOT (i_proc__in_critical(PID1(T1))))))) AND
(((Change1(i_proc__number(PID1(T1)), now)) AND ((i_proc__number(PID1(T1)))
= (const(0)))) IMPLIES ((EX! (t: [time -> time]): ((((Changen(const(2),
i_proc__number(PID1(T1)))) < (t)) AND ((t) < (now))) AND
(Past(Change1(i_proc__in_critical(PID1(T1)), t), t))) AND
(Past(i_proc__in_critical(PID1(T1)), t)))))))(T1) AND TRUE

Invariant: [time -> bool] =
        const(TRUE)

END global_INV
```

## F.3. Global_SCH Theory

```
global_SCH: THEORY

BEGIN

IMPORTING global_INV

DELTA: posreal
T0: time
T1: VAR time

Environment: [time -> bool] =
    const(TRUE)

i_Environment(T1: time): bool =
    TRUE

i_Schedule(T1: time): bool =
    (FA! (PID1: [time -> id]):
        Id_Type(PID1) = const(proc) IMPLIES
        (((FA! (i: [time -> procs_int]): FA! (j: [time -> procs_int]):
((i_proc__in_critical(PID1(T1))) AND (((LAMBDA (T1: time): procs((j)(T1)))) =
(PID1))) IMPLIES (((((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1:
time): procs((i)(T1))))(T1))(T1))) = (const(0))) OR ((i_proc__number(PID1(T1))) <
((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1))))) OR ((i_proc__number(PID1(T1))) = ((LAMBDA (T1:
time): i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))))))) AND
((FA! (i: [time -> procs_int]): FA! (j: [time -> procs_int]):
((i_proc__in_critical(PID1(T1))) AND (((LAMBDA (T1: time): procs((j)(T1)))) =
(PID1))) IMPLIES (((((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1:
time): procs((i)(T1))))(T1))(T1))) = (const(0))) OR ((i_proc__number(PID1(T1))) <
((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1))))) OR ((j) < (i)))))))(T1) AND TRUE

s_Schedule(N1: int): [time -> bool] =
    IF N1 = 1 THEN
        (FA! (i: [time -> procs_int]): FA! (j: [time -> procs_int]): (((LAMBDA
(T1: time): i_proc__in_critical(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1)))
AND ((LAMBDA (T1: time): i_proc__in_critical(((LAMBDA (T1: time):
procs((j)(T1))))(T1))(T1)))) IMPLIES ((i) = (j)))
    ELSE const(TRUE)
    ENDIF

Schedule: [time -> bool] =
    s_Schedule(1)

% global schedule base case
global_SCH_base: THEOREM
    Invariant(0) AND
    Environment(0) AND
    i_Invariant(0) AND
    i_Schedule(0) IMPLIES
        Schedule(0)

% split global schedule induction case
global_SCH_ind_1: THEOREM
    (FORALL (T1):
        Invariant(T1) AND
        Environment(T1) AND
        i_Invariant(T1) AND
        i_Schedule(T1)) AND
    (FORALL (T1): T1 <= T0 IMPLIES Schedule(T1)) IMPLIES
        (FORALL (T1): T0 < T1 AND T1 < T0 + DELTA IMPLIES
            s_Schedule(1)(T1))

END global_SCH
```

## F.4. Proc_IV Theory

```
proc_IV: THEORY

BEGIN

IMPORTING global_SCH

T1: VAR time

i_Imported_Variable(N1: int)(T1: time): bool =
    IF N1 = 1 THEN
        (FA! (PID1: [time -> id]): Id_Type(PID1) = const(proc) IMPLIES
        ((FA! (i: [time -> procs_int]): ((LAMBDA (T1: time):
i_proc__in_critical(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))) IMPLIES
(NOT ((LAMBDA (T1: time): i_proc__choosing(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1)))))))(T1)
    ELSIF N1 = 2 THEN
        (FA! (PID1: [time -> id]): Id_Type(PID1) = const(proc) IMPLIES
        ((FA! (i: [time -> procs_int]): ((LAMBDA (T1: time):
i_proc__in_critical(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))) IMPLIES
(((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1))) /= (const(0))))))(T1)
    ELSIF N1 = 3 THEN
        (FA! (PID1: [time -> id]): Id_Type(PID1) = const(proc) IMPLIES
        ((FA! (i: [time -> procs_int]): FA! (j: [time -> procs_int]):
((Change1((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1)), now)) AND (((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))) /= (const(0))))
IMPLIES (((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1))) >= (Past(((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((j)(T1))))(T1))(T1))) + (const(1)),
(now) - (const(exec_time)))))))(T1)
    ELSIF N1 = 4 THEN
        (FA! (PID1: [time -> id]): Id_Type(PID1) = const(proc) IMPLIES
        ((FA! (i: [time -> procs_int]): ((Change1((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1)), now)) AND
(((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1))) = (const(0)))) IMPLIES (NOT ((LAMBDA (T1: time):
i_proc__in_critical(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1)))))))(T1)
    ELSIF N1 = 5 THEN
        (FA! (PID1: [time -> id]): Id_Type(PID1) = const(proc) IMPLIES
        ((FA! (i: [time -> procs_int]): ((Change1((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1)), now)) AND
(((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1))) = (const(0)))) IMPLIES ((EX! (t: [time -> time]):
(((((Changen(const(2), (LAMBDA (T1: time): i_proc__number(((LAMBDA (T1:
time): procs((i)(T1))))(T1))(T1)))) < (t)) AND ((t) < (now))) AND
(Past(Change1((LAMBDA (T1: time): i_proc__in_critical(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1)), t), t))) AND (Past((LAMBDA (T1: time):
i_proc__in_critical(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1)), t)))))))(T1)
    ELSE TRUE
    ENDIF

% split imported variable obligation
proc_IV_1: THEOREM
    (FORALL (T1):
        Invariant(T1) AND
        Environment(T1) AND
        i_Invariant(T1) AND
        i_Environment(T1)) IMPLIES
        (FORALL (T1): i_Imported_Variable(1)(T1))

% split imported variable obligation
proc_IV_2: THEOREM
    (FORALL (T1):
        Invariant(T1) AND
        Environment(T1) AND
        i_Invariant(T1) AND
        i_Environment(T1)) IMPLIES
        (FORALL (T1): i_Imported_Variable(2)(T1))

% split imported variable obligation
proc_IV_3: THEOREM
    (FORALL (T1):
        Invariant(T1) AND
        Environment(T1) AND
        i_Invariant(T1) AND
        i_Environment(T1)) IMPLIES
        (FORALL (T1): i_Imported_Variable(3)(T1))

% split imported variable obligation
proc_IV_4: THEOREM
```

```
        (FORALL (T1):
            Invariant(T1) AND
            Environment(T1) AND
            i_Invariant(T1) AND
            i_Environment(T1)) IMPLIES
            (FORALL (T1): i_Imported_Variable(4)(T1))

% split imported variable obligation
proc_IV_5: THEOREM
        (FORALL (T1):
            Invariant(T1) AND
            Environment(T1) AND
            i_Invariant(T1) AND
            i_Environment(T1)) IMPLIES
            (FORALL (T1): i_Imported_Variable(5)(T1))

END proc_IV
```

# F.5. Proc_L_Top_Level Theory

```
proc_L_top_level: THEORY

BEGIN

IMPORTING global

self: {ID1: id | Id_Type(const(ID1))(0) = proc}

transition: TYPE = {set_choose, set_number, reset_choose, for_loop, start_critical,
end_critical}

next_i: [time -> pos_int]

choosing: [time -> boolean]

number: [time -> nonneg_int]

in_critical: [time -> boolean]

delay: [time -> nonneg_real]

parameter: TYPE = [# DUMMY: int #]

undef_parm: parameter

Vars_No_Change(T1: time, T2: time): bool =
        next_i(T1) = next_i(T2) AND
        choosing(T1) = choosing(T2) AND
        number(T1) = number(T2) AND
        in_critical(T1) = in_critical(T2) AND
        delay(T1) = delay(T2)

Var_Changes(T1: time): bool =
        Change1(next_i, const(T1))(T1) OR
        Change1(choosing, const(T1))(T1) OR
        Change1(number, const(T1))(T1) OR
        Change1(in_critical, const(T1))(T1) OR
        Change1(delay, const(T1))(T1)

Base_Trans(TR1: transition): transition = TR1

Duration(TR1: transition): posreal =
        CASES TR1 OF
            set_choose: exec_time,
            set_number: exec_time,
            reset_choose: exec_time,
            for_loop: exec_time,
            start_critical: exec_time,
            end_critical: exec_time
        ENDCASES

Exported(BTR1: {TR1: transition | Base_Trans(TR1) = TR1}): bool = FALSE

Has_Parms(BTR1: {TR1: transition | Base_Trans(TR1) = TR1}): bool = FALSE

Num_Parms(BTR1: {TR1: transition | Base_Trans(TR1) = TR1}): nat = 0

IMPORTING astral_lib@astral_trans[transition, parameter, Duration, Base_Trans,
        Has_Parms, Exported]

Eval_Parms(BTR1: {TR1: transition | Base_Trans(TR1) = TR1 AND
        Has_Parms(TR1)}, N1: nat, P1: parameter, P2: parameter): bool =
        FALSE

Entry_No_Parms(TR1: {tr: transition | NOT Has_Parms(Base_Trans(tr))}):
                [time -> bool] =
        CASES TR1 OF
            set_choose:
(((now) >= (delay)) AND (NOT (choosing))) AND ((number) = (const(0))),
            set_number:
(choosing) AND ((FA! (t: [time -> time]): (Change1(number, t)) IMPLIES ((t) <
(Change1(choosing))))),
            reset_choose:
(choosing) AND ((FA! (t: [time -> time]): (Change1(number, t)) IMPLIES ((t) >
(Change1(choosing))))),
            for_loop:
(((((next_i) <= (const(n_procs))) AND (NOT (choosing))) AND ((number) /=
(const(0)))) AND (NOT ((LAMBDA (T1: time): i_proc__choosing(((LAMBDA (T1:
time): procs((next_i)(T1))))(T1))(T1))))) AND (((((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((next_i)(T1))))(T1))(T1))) =
(const(0))) OR ((number) < ((LAMBDA (T1: time): i_proc__number(((LAMBDA
(T1: time): procs((next_i)(T1))))(T1))(T1)))))) OR (((number) = ((LAMBDA (T1:
time): i_proc__number(((LAMBDA (T1: time): procs((next_i)(T1))))(T1))(T1))))
AND ((FA! (j: [time -> procs_int]): (((LAMBDA (T1: time): procs((j)(T1)))) =
(const(self))) IMPLIES ((j) <= (next_i))))))),
            start_critical:
((next_i) > (const(n_procs))) AND (NOT (in_critical)),
            end_critical:
in_critical
        ENDCASES

Entry_Parms(TR1: {tr: transition | Has_Parms(Base_Trans(tr))}, P1: parameter):
                [time -> bool] =
        const(TRUE)

Exit_No_Parms(TR1: {tr: transition | NOT Has_Parms(Base_Trans(tr))})
                (T1: {T1: time | T1 >= Duration(TR1)}): bool =
        (CASES TR1 OF
            set_choose:
(choosing) AND (LAMBDA (T1: time): next_i(T1) = next_i(T1 -
Duration(set_choose))) AND (LAMBDA (T1: time): number(T1) = number(T1 -
Duration(set_choose))) AND (LAMBDA (T1: time): in_critical(T1) = in_critical(T1 -
Duration(set_choose))) AND (LAMBDA (T1: time): delay(T1) = delay(T1 -
Duration(set_choose))),
            set_number:
(((FA! (i: [time -> procs_int]): (number) >= (((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1 -
Duration(set_number)))) + (const(1))))) AND ((EX! (i: [time -> procs_int]):
(number) = (((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1 - Duration(set_number)))) + (const(1)))))) AND (LAMBDA
(T1: time): next_i(T1) = next_i(T1 - Duration(set_number))) AND (LAMBDA (T1:
time): choosing(T1) = choosing(T1 - Duration(set_number))) AND (LAMBDA (T1:
time): in_critical(T1) = in_critical(T1 - Duration(set_number))) AND (LAMBDA
(T1: time): delay(T1) = delay(T1 - Duration(set_number))),
            reset_choose:
(NOT (choosing)) AND (LAMBDA (T1: time): next_i(T1) = next_i(T1 -
Duration(reset_choose))) AND (LAMBDA (T1: time): number(T1) = number(T1 -
Duration(reset_choose))) AND (LAMBDA (T1: time): in_critical(T1) =
in_critical(T1 - Duration(reset_choose))) AND (LAMBDA (T1: time): delay(T1) =
delay(T1 - Duration(reset_choose))),
            for_loop:
((next_i) = (((LAMBDA (T1: time): next_i(T1 - Duration(for_loop)))) + (const(1))))
AND (LAMBDA (T1: time): choosing(T1) = choosing(T1 - Duration(for_loop)))
AND (LAMBDA (T1: time): number(T1) = number(T1 - Duration(for_loop))) AND
(LAMBDA (T1: time): in_critical(T1) = in_critical(T1 - Duration(for_loop))) AND
(LAMBDA (T1: time): delay(T1) = delay(T1 - Duration(for_loop))),
            start_critical:
(in_critical) AND (LAMBDA (T1: time): next_i(T1) = next_i(T1 -
Duration(start_critical))) AND (LAMBDA (T1: time): choosing(T1) = choosing(T1 -
Duration(start_critical))) AND (LAMBDA (T1: time): number(T1) = number(T1 -
Duration(start_critical))) AND (LAMBDA (T1: time): delay(T1) = delay(T1 -
Duration(start_critical))),
            end_critical:
((((NOT (in_critical)) AND ((next_i) = (const(1)))) AND ((number) = (const(0))))
AND ((delay) >= (now))) AND (LAMBDA (T1: time): choosing(T1) = choosing(T1
- Duration(end_critical)))
        ENDCASES)(T1)
```

```
Exit_Parms(TR1: {tr: transition | Has_Parms(Base_Trans(tr))}, P1: parameter)
            (T1: {T1: time | T1 >= Duration(TR1)}): bool =
      TRUE

IMPORTING astral_lib@astral_trans_aux[transition, parameter, Duration,
      Base_Trans, Has_Parms, Exported, time, Entry_No_Parms, Exit_No_Parms,
      Entry_Parms, Exit_Parms, Eval_Parms, Num_Parms]

Initial: [time -> bool] =
      ((((next_i) = (const(1))) AND (NOT (choosing))) AND ((number) =
(const(0)))) AND (NOT (in_critical))

local_axiom: AXIOM
      (const(TRUE))(0)

self_imports: AXIOM
      i_proc__choosing(self) = choosing AND
      i_proc__number(self) = number AND
      i_proc__in_critical(self) = in_critical

END proc_L_top_level
```

# F.6. Proc_L_Top_Level_SG Theory

```
proc_L_top_level_SG: THEORY

BEGIN

IMPORTING proc_L_top_level
IMPORTING astral_lib@astral_change_aux
IMPORTING astral_lib@astral_lemmas[transition, parameter, time, Base_Trans,
            Duration, Has_Parms, Exported, Issued_Call, Entry, Exit,
            Enabled, Fired, Initial, Var_Changes, Vars_No_Change]

Initial_NOT_set_choose: THEOREM
      Not_Initial(set_choose)

set_choose_NOT_set_choose: THEOREM
      Not_Sequence(set_choose, set_choose, FALSE)

set_choose_NOT_set_number: THEOREM
      Not_Sequence(set_choose, set_number, FALSE)

set_choose_NOT_reset_choose: THEOREM
      Not_Sequence(set_choose, reset_choose, FALSE)

set_choose_NOT_for_loop: THEOREM
      Not_Sequence(set_choose, for_loop, FALSE)

set_choose_NOT_start_critical: THEOREM
      Not_Sequence(set_choose, start_critical, FALSE)

set_choose_NOT_end_critical: THEOREM
      Not_Sequence(set_choose, end_critical, FALSE)

Initial_NOT_set_number: THEOREM
      Not_Initial(set_number)

set_number_NOT_set_choose: THEOREM
      Not_Sequence(set_number, set_choose, FALSE)

set_number_NOT_set_number: THEOREM
      Not_Sequence(set_number, set_number, FALSE)

set_number_NOT_reset_choose: THEOREM
      Not_Sequence(set_number, reset_choose, FALSE)

set_number_NOT_for_loop: THEOREM
      Not_Sequence(set_number, for_loop, FALSE)

set_number_NOT_start_critical: THEOREM
      Not_Sequence(set_number, start_critical, FALSE)

set_number_NOT_end_critical: THEOREM
      Not_Sequence(set_number, end_critical, FALSE)

Initial_NOT_reset_choose: THEOREM
      Not_Initial(reset_choose)

reset_choose_NOT_set_choose: THEOREM
      Not_Sequence(reset_choose, set_choose, FALSE)

reset_choose_NOT_set_number: THEOREM
      Not_Sequence(reset_choose, set_number, FALSE)

reset_choose_NOT_reset_choose: THEOREM
      Not_Sequence(reset_choose, reset_choose, FALSE)

reset_choose_NOT_for_loop: THEOREM
      Not_Sequence(reset_choose, for_loop, FALSE)

reset_choose_NOT_start_critical: THEOREM
      Not_Sequence(reset_choose, start_critical, FALSE)

reset_choose_NOT_end_critical: THEOREM
      Not_Sequence(reset_choose, end_critical, FALSE)

Initial_NOT_for_loop: THEOREM
      Not_Initial(for_loop)

for_loop_NOT_set_choose: THEOREM
      Not_Sequence(for_loop, set_choose, FALSE)

for_loop_NOT_set_number: THEOREM
      Not_Sequence(for_loop, set_number, FALSE)

for_loop_NOT_reset_choose: THEOREM
      Not_Sequence(for_loop, reset_choose, FALSE)

for_loop_NOT_for_loop: THEOREM
      Not_Sequence(for_loop, for_loop, FALSE)

for_loop_NOT_start_critical: THEOREM
      Not_Sequence(for_loop, start_critical, FALSE)

for_loop_NOT_end_critical: THEOREM
      Not_Sequence(for_loop, end_critical, FALSE)

Initial_NOT_start_critical: THEOREM
      Not_Initial(start_critical)

start_critical_NOT_set_choose: THEOREM
      Not_Sequence(start_critical, set_choose, FALSE)

start_critical_NOT_set_number: THEOREM
      Not_Sequence(start_critical, set_number, FALSE)

start_critical_NOT_reset_choose: THEOREM
      Not_Sequence(start_critical, reset_choose, FALSE)

start_critical_NOT_for_loop: THEOREM
      Not_Sequence(start_critical, for_loop, FALSE)

start_critical_NOT_start_critical: THEOREM
      Not_Sequence(start_critical, start_critical, FALSE)

start_critical_NOT_end_critical: THEOREM
      Not_Sequence(start_critical, end_critical, FALSE)

Initial_NOT_end_critical: THEOREM
      Not_Initial(end_critical)

end_critical_NOT_set_choose: THEOREM
      Not_Sequence(end_critical, set_choose, FALSE)

end_critical_NOT_set_number: THEOREM
      Not_Sequence(end_critical, set_number, FALSE)

end_critical_NOT_reset_choose: THEOREM
      Not_Sequence(end_critical, reset_choose, FALSE)

end_critical_NOT_for_loop: THEOREM
      Not_Sequence(end_critical, for_loop, FALSE)

end_critical_NOT_start_critical: THEOREM
      Not_Sequence(end_critical, start_critical, FALSE)

end_critical_NOT_end_critical: THEOREM
```

```
    Not_Sequence(end_critical, end_critical, FALSE)
```

## F.7. Proc_L_Top_Level_INV Theory

```
proc_L_top_level_INV: THEORY

BEGIN

IMPORTING proc_L_top_level_SG

DELTA: posreal
T0: time
T1: VAR time

s_Invariant(N1: int): [time -> bool] =
    IF N1 = 1 THEN
        (in_critical) IMPLIES ((next_i) > (const(n_procs)))
    ELSIF N1 = 2 THEN
        ((next_i) > (const(1))) IMPLIES (NOT (choosing))
    ELSIF N1 = 3 THEN
        ((next_i) > (const(1))) IMPLIES ((number) /= (const(0)))
    ELSIF N1 = 4 THEN
        (in_critical) IMPLIES (NOT (choosing))
    ELSIF N1 = 5 THEN
        (in_critical) IMPLIES ((number) /= (const(0)))
    ELSIF N1 = 6 THEN
        (FA! (i: [time -> procs_int]): ((Change1(number, now)) AND ((number)
/= (const(0)))) IMPLIES ((number) >= (Past(((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((i)(T1)))(T1))(T1))) + (const(1)),
(now) - (const(exec_time))))))
    ELSIF N1 = 7 THEN
        ((Change1(number, now)) AND ((number) = (const(0)))) IMPLIES
(NOT (in_critical))
    ELSIF N1 = 8 THEN
        ((Change1(number, now)) AND ((number) = (const(0)))) IMPLIES
((EX! (t: [time -> time]): ((((Changen(const(2), number)) < (t)) AND ((t) < (now)))
AND (Past(Change1(in_critical, t), t))) AND (Past(in_critical, t))))
    ELSE const(TRUE)
    ENDIF

Invariant: [time -> bool] =
    s_Invariant(1) AND
    s_Invariant(2) AND
    s_Invariant(3) AND
    s_Invariant(4) AND
    s_Invariant(5) AND
    s_Invariant(6) AND
    s_Invariant(7) AND
    s_Invariant(8)

% local invariant base case
proc_L_top_level_INV_base: THEOREM
    Invariant(0)

% split local invariant induction case
proc_L_top_level_INV_ind_1: THEOREM
    (FORALL (TR1: transition): DELTA < Duration(TR1)) AND
    (FORALL (T1): T1 <= T0 IMPLIES Invariant(T1)) IMPLIES
        (FORALL (T1): T0 < T1 AND T1 < T0 + DELTA IMPLIES
            s_Invariant(1)(T1))

% split local invariant induction case
proc_L_top_level_INV_ind_2: THEOREM
    (FORALL (TR1: transition): DELTA < Duration(TR1)) AND
    (FORALL (T1): T1 <= T0 IMPLIES Invariant(T1)) IMPLIES
        (FORALL (T1): T0 < T1 AND T1 < T0 + DELTA IMPLIES
            s_Invariant(2)(T1))

% split local invariant induction case
proc_L_top_level_INV_ind_3: THEOREM
    (FORALL (TR1: transition): DELTA < Duration(TR1)) AND
    (FORALL (T1): T1 <= T0 IMPLIES Invariant(T1)) IMPLIES
        (FORALL (T1): T0 < T1 AND T1 < T0 + DELTA IMPLIES
            s_Invariant(3)(T1))

% split local invariant induction case
proc_L_top_level_INV_ind_4: THEOREM
    (FORALL (TR1: transition): DELTA < Duration(TR1)) AND
    (FORALL (T1): T1 <= T0 IMPLIES Invariant(T1)) IMPLIES
        (FORALL (T1): T0 < T1 AND T1 < T0 + DELTA IMPLIES
            s_Invariant(4)(T1))

% split local invariant induction case
proc_L_top_level_INV_ind_5: THEOREM
    (FORALL (TR1: transition): DELTA < Duration(TR1)) AND
    (FORALL (T1): T1 <= T0 IMPLIES Invariant(T1)) IMPLIES
        (FORALL (T1): T0 < T1 AND T1 < T0 + DELTA IMPLIES
            s_Invariant(5)(T1))

% split local invariant induction case
proc_L_top_level_INV_ind_6: THEOREM
    (FORALL (TR1: transition): DELTA < Duration(TR1)) AND
    (FORALL (T1): T1 <= T0 IMPLIES Invariant(T1)) IMPLIES
        (FORALL (T1): T0 < T1 AND T1 < T0 + DELTA IMPLIES
            s_Invariant(6)(T1))

% split local invariant induction case
proc_L_top_level_INV_ind_7: THEOREM
    (FORALL (TR1: transition): DELTA < Duration(TR1)) AND
    (FORALL (T1): T1 <= T0 IMPLIES Invariant(T1)) IMPLIES
        (FORALL (T1): T0 < T1 AND T1 < T0 + DELTA IMPLIES
            s_Invariant(7)(T1))

% split local invariant induction case
proc_L_top_level_INV_ind_8: THEOREM
    (FORALL (TR1: transition): DELTA < Duration(TR1)) AND
    (FORALL (T1): T1 <= T0 IMPLIES Invariant(T1)) IMPLIES
        (FORALL (T1): T0 < T1 AND T1 < T0 + DELTA IMPLIES
            s_Invariant(8)(T1))

END proc_L_top_level_INV
```

## F.8. Proc_L_Top_Level_CON Theory

```
proc_L_top_level_CON: THEORY

BEGIN

IMPORTING proc_L_top_level_INV

T1: VAR time

END proc_L_top_level_CON
```

## F.9. Proc_L_Top_Level_SCH Theory

```
proc_L_top_level_SCH: THEORY

BEGIN

IMPORTING proc_L_top_level_CON

DELTA: posreal
T0: time
T1: VAR time

Environment: [time -> bool] =
    const(TRUE)
```

```
Imported_Variable: [time -> bool] =
        (((FA! (i: [time -> procs_int]): ((LAMBDA (T1: time):
i_proc__in_critical(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))) IMPLIES
((NOT ((LAMBDA (T1: time): i_proc__choosing(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1)))) AND (((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))) /= (const(0))))))
AND ((FA! (i: [time -> procs_int]): FA! (j: [time -> procs_int]):
((Change1((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1)), now)) AND (((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))) /= (const(0))))
IMPLIES (((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1))) >= (Past(((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((j)(T1))))(T1))(T1))) + (const(1)),
(now) - (const(exec_time)))))))) AND ((FA! (i: [time -> procs_int]):
((Change1((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1)), now)) AND (((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))) = (const(0))))
IMPLIES ((NOT ((LAMBDA (T1: time): i_proc__in_critical(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1)))) AND ((EX! (t: [time -> time]): (((((Changen(const(2),
(LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1)))) < (t)) AND ((t) < (now))) AND
(Past(Change1((LAMBDA (T1: time): i_proc__in_critical(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1)), t), t))) AND (Past((LAMBDA (T1: time):
i_proc__in_critical(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1)), t))))))

Further_Environment(N1: int): [time -> bool] =
        IF N1 = 1 THEN
                const(TRUE)
        ELSE const(TRUE)
        ENDIF

Constant_Refinement(N1: int): [time -> bool] =
        IF N1 = 1 THEN
                const(TRUE)
        ELSE const(TRUE)
        ENDIF

Elgible_Set(N1: int)(T1: time): set[transition] =
        IF N1 = 1 THEN
                Enabled_Set(T1)
        ELSE emptyset
        ENDIF

Transition_Selection(N1: int)(T1: time): bool =
        IF N1 = 1 THEN
                (FORALL (TR1: transition):
                        Fired(TR1, T1) IMPLIES member(TR1, Elgible_Set(1)(T1)))
        ELSE TRUE
        ENDIF

s_Schedule(N1: int): [time -> bool] =
        IF N1 = 1 THEN
                (FA! (i: [time -> procs_int]): FA! (j: [time -> procs_int]): ((in_critical)
AND (((LAMBDA (T1: time): procs((j)(T1))))) = (const(self)))) IMPLIES
(((((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1))) = (const(0))) OR ((number) < ((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))))) OR ((number)
= ((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1))))))
        ELSIF N1 = 2 THEN
                (FA! (i: [time -> procs_int]): FA! (j: [time -> procs_int]): ((in_critical)
AND (((LAMBDA (T1: time): procs((j)(T1))))) = (const(self)))) IMPLIES
(((((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1))) = (const(0))) OR ((number) < ((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))))) OR ((j) < (i))))
        ELSIF N1 = 3 THEN
                (FA! (i: [time -> procs_int]): FA! (j: [time -> procs_int]):
(((Start1(for_loop, now)) AND (((LAMBDA (T1: time): procs((j)(T1)))) =
(const(self)))) AND ((i) < (next_i))) IMPLIES (((((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))) = (const(0)))
OR ((number) < ((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1))))) OR ((number) = ((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))))))
        ELSIF N1 = 4 THEN
                (FA! (i: [time -> procs_int]): FA! (j: [time -> procs_int]):
(((Start1(for_loop, now)) AND (((LAMBDA (T1: time): procs((j)(T1)))) =
(const(self)))) AND ((i) < (next_i))) IMPLIES (((((LAMBDA (T1: time):
i_proc__number(((LAMBDA (T1: time): procs((i)(T1))))(T1))(T1))) = (const(0)))
OR ((number) < ((LAMBDA (T1: time): i_proc__number(((LAMBDA (T1: time):
procs((i)(T1))))(T1))(T1))))) OR ((j) <= (i))))
        ELSE const(TRUE)
        ENDIF

Schedule: [time -> bool] =
        s_Schedule(1) AND
        s_Schedule(2) AND
        s_Schedule(3) AND
        s_Schedule(4)

% local schedule base case
proc_L_top_level_SCH_base_1: THEOREM
        Invariant(0) AND
        Environment(0) AND
        Imported_Variable(0) AND
        Further_Environment(1)(0) AND
        Constant_Refinement(1)(0) AND
        Transition_Selection(1)(0) IMPLIES
                Schedule(0)

% split local schedule induction case
proc_L_top_level_SCH_ind_1_1: THEOREM
        (FORALL (T1):
                Invariant(T1) AND
                Environment(T1) AND
                Imported_Variable(T1) AND
                Further_Environment(1)(T1) AND
                Constant_Refinement(1)(T1) AND
                Transition_Selection(1)(T1)) AND
        (FORALL (TR1: transition): DELTA < Duration(TR1)) AND
        (FORALL (T1): T1 <= T0 IMPLIES Schedule(T1)) IMPLIES
                (FORALL (T1): T0 < T1 AND T1 < T0 + DELTA IMPLIES
                        s_Schedule(1)(T1))

% split local schedule induction case
proc_L_top_level_SCH_ind_1_2: THEOREM
        (FORALL (T1):
                Invariant(T1) AND
                Environment(T1) AND
                Imported_Variable(T1) AND
                Further_Environment(1)(T1) AND
                Constant_Refinement(1)(T1) AND
                Transition_Selection(1)(T1)) AND
        (FORALL (TR1: transition): DELTA < Duration(TR1)) AND
        (FORALL (T1): T1 <= T0 IMPLIES Schedule(T1)) IMPLIES
                (FORALL (T1): T0 < T1 AND T1 < T0 + DELTA IMPLIES
                        s_Schedule(2)(T1))

% split local schedule induction case
proc_L_top_level_SCH_ind_1_3: THEOREM
        (FORALL (T1):
                Invariant(T1) AND
                Environment(T1) AND
                Imported_Variable(T1) AND
                Further_Environment(1)(T1) AND
                Constant_Refinement(1)(T1) AND
                Transition_Selection(1)(T1)) AND
        (FORALL (TR1: transition): DELTA < Duration(TR1)) AND
        (FORALL (T1): T1 <= T0 IMPLIES Schedule(T1)) IMPLIES
                (FORALL (T1): T0 < T1 AND T1 < T0 + DELTA IMPLIES
                        s_Schedule(3)(T1))

% split local schedule induction case
proc_L_top_level_SCH_ind_1_4: THEOREM
        (FORALL (T1):
                Invariant(T1) AND
                Environment(T1) AND
                Imported_Variable(T1) AND
                Further_Environment(1)(T1) AND
                Constant_Refinement(1)(T1) AND
                Transition_Selection(1)(T1)) AND
        (FORALL (TR1: transition): DELTA < Duration(TR1)) AND
        (FORALL (T1): T1 <= T0 IMPLIES Schedule(T1)) IMPLIES
                (FORALL (T1): T0 < T1 AND T1 < T0 + DELTA IMPLIES
                        s_Schedule(4)(T1))

END proc_L_top_level_SCH
```