Proof Assistance for Real-Time Systems Using an Interactive Theorem Prover

Paul Z. Kolano

Computer Science Department, University of California
Santa Barbara, CA 93106 U.S.A.
kolano@cs.ucsb.edu

Abstract. This paper discusses the adaptation of the PVS theorem prover for performing analysis of real-time systems written in the ASTRAL formal specification language. A number of issues were encountered during the encoding of ASTRAL that are relevant to the encoding of many real-time specification languages. These issues are presented as well as how they were handled in the ASTRAL encoding. A translator has been written that translates any ASTRAL specification into its corresponding PVS encoding. After performing the proofs of several systems using the encoding, PVS strategies have been developed to automate the proofs of certain types of properties. In addition, the encoding has been used as the basis for a transition sequence generator tool.

1 Introduction

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 controls where deviation from critical timing requirements can result in disastrous loss of lives and/or property. It is thus desirable to extensively test and verify the designs of these systems to gain assurance that such disasters will not occur. A number of formal methods for real-time systems have been proposed [14] that 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, nondeterminism, and dependence upon the external operating environment. Thus, the formal proofs of even simple real-time systems can be nontrivial. To make the verification of real-world real-time systems practical, mechanical proof assistance is necessary.

One such form of assistance is an interactive theorem prover. Interactive theorem provers provide mechanical support for deductive reasoning. Each theorem prover is associated with a specification language in which a system and associated theorems are expressed. A theorem prover uses a collection of axioms and inference rules about its specification language to reduce a high-level proof into simpler subproofs that can eventually be discharged by basic built-in decision procedures that support

arithmetic and boolean reasoning. Theorem provers provide a number of forms of assistance, including preserving the soundness of proofs, finishing off proof details automatically, keeping track of proof status, and recording proofs for reuse.

This paper discusses the adaptation of the PVS theorem prover [8] for performing analysis of real-time systems written in the ASTRAL [5] formal specification language. A number of issues were encountered during the encoding of ASTRAL that are relevant to the encoding of many real-time specification languages. These issues are presented as well as how they were handled in the ASTRAL encoding. A translator has been written that translates any ASTRAL specification into its corresponding PVS encoding. After performing the proofs of several systems using the encoding, PVS strategies have been developed to automate the proofs of certain types of properties. In addition, the encoding has been used as the basis for a transition sequence generator tool.

The remainder of this paper is organized as follows. In sections 2 and 3, brief overviews of ASTRAL and PVS are given. In section 4, the issues encountered during the encoding of ASTRAL are discussed. Section 5 describes the ASTRAL to PVS translator. Strategies for automating ASTRAL proofs and the use of PVS to develop a transition sequence generator are presented in section 6. Section 7 discusses related work. Finally, section 8 provides some conclusions and directions for future research.

2 ASTRAL

In ASTRAL [5], a real-time system is described as a collection of state machine specifications, each of them representing a process type of which there may be multiple statically generated instances. There is also a *global specification*, which 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.

An ASTRAL *process specification* consists of a sequence of *levels*. Each level is an abstract data type view of the system 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. Figure 1 shows one of the process types of an elevator control system. The Elevator_Button_Panel process represents the button panel located within an elevator car.

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, where *entry assertions* describe the constraints that state variables must satisfy in order for the transition to fire, and *exit assertions* describe the constraints that are fulfilled by state variables after the transition has fired. Variables are changed atomically at the end of a transition's execution with variables not referenced in the exit assertion remaining unchanged. An explicit non-null duration is associated with each transition. A transition is executed as soon as it is enabled (i.e. when its entry assertion is satisfied), assuming no other transition for that process

instance is executing. In the Elevator_Button_Panel process, the clear_floor_request transition is enabled when the elevator is currently stopped with its door opening at a floor that has been requested.

```
PROCESS Elevator_Button_Panel
    IMPORT
        floor, request_dur, clear_dur,
        elevator, elevator.position,
        elevator.door_open,
        elevator.door_moving
    EXPORT
        floor_requested, request_floor
    VARIABLE
        floor_requested(floor): boolean
    INITIAL
        FORALL f: floor
            (~floor_requested(f))
    TRANSITION request_floor(f: floor)
        ENTRY   [TIME:  request_dur]
            ~floor_requested(f)
        EXIT
            floor_requested(f)
                Becomes TRUE
TRANSITION clear_floor_request
    ENTRY   [TIME:  clear_dur]
        floor_requested(elevator.position)
    &   ~elevator.door_open
    &   elevator.door_moving
    EXIT
        floor_requested(elevator.position)
            Becomes FALSE
INVARIANT
    FORALL f: floor
        (   Change(floor_requested(f), now)
        &   ~floor_requested(f)
    →       EXISTS t: time
                (   Change2(floor_requested(f)) < t
                &   t ≤ now
                &   past(elevator.position, t) = f
                &   ~past(elevator.door_open, t)
                &   past(elevator.door_moving, t)))
```

Fig. 1. The Elevator_Button_Panel process

Every process can export both state variables and transitions; as a consequence, the former are readable by other processes and the external environment while the latter are executable from the external environment. Interprocess communication is accomplished by broadcasting the values of exported variables and the start and end times of exported transitions. In the Elevator_Button_Panel process, the floor_requested variable and the request_floor transition are exported. The position, door_open, and door_moving variables of the elevator process are imported.

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 about 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 requirement and assumption clauses are expressed using a combination of first-order logic and ASTRAL-specific constructs. The main constructs are the timed operators used to express timing requirements. The *start* operator, Start(trans1, t1), takes a transition trans1 and a time t1 and returns true iff the last start of trans1 was at t1. Similarly, the *end* and *call* operators, End(trans1, t1) and Call(trans1, t1), return true iff the last end or the last call of trans1 was at t1. The *change* operator,

Change(A, t), takes an expression A and a time t and returns true iff the last time A changed value was at t. The *past* operator, past(A, t), takes an expression A and a time t and returns the value of A at t. In addition to these operators, a special global variable *now* is used to denote the current time, where the time domain is the nonnegative real numbers.

Using these operators, a variety of complex properties can be expressed. For example, the invariant of the Elevator_Button_Panel process states that between a change to floor_requested(f) and a change back to ~floor_requested(f) for any floor f, the elevator has been at f and its door has started opening. An introduction and complete overview of the ASTRAL language can be found in [5]. For the complete description and specification of the elevator system, see [16].

Rather than implementing a theorem prover for ASTRAL from scratch, it was decided to take advantage of an existing general-purpose theorem prover adapted for use with ASTRAL. PVS was considered ideal for ASTRAL given its powerful typing system, higher-order facilities, heavily automated decision procedures, and ease of use. Other theorem provers were also considered, including HOL [12] and ACL2 [15]. HOL does not have the usability of PVS and its decision procedures are not as powerful [11]. ACL2 is also not as usable as PVS and has limited or no support for arbitrary quantification and real numbers [20].

3 PVS

The Prototype Verification System (PVS) [8] is a powerful interactive theorem prover based on typed higher-order logic. A PVS specification consists of a modular collection of *theories*, where a theory is defined by a set of type, constant, axiom, and theorem declarations. 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 "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, for the declaration "e_plus_2(e: even_int): even_int = e + 2", PVS generates the following TCC:

```
% 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. Besides types and constants, a theory declaration contains axioms, which are basic "truths" of the theory and theorems, which are hypotheses that are thought to be true, but that need to be proven with the prover.

When the PVS prover 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_m are consequents in the current sequent, then the current goal is $(A_1 \& ... \& A_n) \rightarrow (C_1 \mid ... \mid C_m)$. 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 is true, all 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.

4 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.

4.1 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 [1], an uninterpreted "TRIO_formula" type is introduced to handle this issue. In TRIO, 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) is another real-time language that has been encoded into PVS [18]. 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 $\rightarrow$ 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(trans1, t1) takes a transition trans1 and a time t1 and returns true iff the last start of trans1 was at t1. Its PVS counterpart, Start1(trans1, at1) takes a transition trans1 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 iff the last start of trans1 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(trans1: transition, at1: [time → time])(t1: {t1: time | at1(t1) ≤ t1}): bool =
    Fired(trans1, at1(t1)) AND
    (FORALL (t2):
        at1(t1) < t2 AND t2 ≤ t1 IMPLIES
            NOT Fired(trans1, t2))
```

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.

4.2 Partial Functions

Some specification languages such as Z [19] 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 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 needed 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 $\rightarrow$ 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.

4.3 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*. Called(trans1, t1) is true iff transition trans1 was called from the external environment at time t1. Fired(trans1, t1) is true iff trans1 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.

Two of the ten axioms of the ASTRAL axiomatization are shown below. The axiomatization of ASTRAL into PVS is a much revised and expanded version of the ASTRAL axiomatization of [7] and includes corrections for both soundness and completeness. The full version of the semantics presented in this paper defines the current formal semantics of the ASTRAL language. The trans_fire axiom states that if some transition is enabled and the process is idle (i.e. no transition in the middle of execution), then some transition fires. The trans_mutex axiom states that whenever a transition fires, no other transition can fire until its duration has elapsed (i.e. until the transition ends). This axiom combined with trans_fire is sufficient to show that a single unique transition fires on a particular process instance when some transition is enabled and the process is idle. Note that since the semantics cannot be represented by a sequence of transitions as in an interleaved model, it is necessary to assure that a process is actually idle in order for a transition to fire.

```
trans_fire: AXIOM
 (FORALL (t1):
    (EXISTS (trans1):
       Enabled(trans1, t1)) AND
    (FORALL (trans2, t2):
       t1 - Duration(trans2) < t2 AND
       t2 < t1 IMPLIES
          NOT Fired(trans2, t2)) IMPLIES
    (EXISTS (trans1):
       Fired(trans1, t1)))
```

```
trans_mutex: AXIOM
 (FORALL (trans1, t1):
    Fired(trans1, t1) IMPLIES
    (FORALL (trans2):
       trans2 ≠ trans1 IMPLIES
          NOT Fired(trans2, t1)) AND
    (FORALL (trans2, t2):
       t1 < t2 AND
       t2 < t1 + Duration(trans1) IMPLIES
          NOT Fired(trans2, t2)))
```

Since ASTRAL is based on noninterleaved concurrency, the intra-level proof obligations [7] (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): T1 ≤ T0 IMPLIES Invariant(T1)) IMPLIES
      (FORALL (T1): 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 $T0 + \Delta$. 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 (trans1: transition): $\Delta <$ Duration(trans1))" 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 6.1.

5 PVS Library and Translator

The axiomatization and operator definitions discussed in section 4 have been incorporated into an ASTRAL-PVS library. The library contains the specification-independent core of the ASTRAL language. In the axiomatization and operators, 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 the appropriate definitions.

The major obstacle in translating ASTRAL specifications 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" and "struct1: structure of (l_one(integer): 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((s1)(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.

For the full details of the axiomatization of the ASTRAL abstract machine, the operator definitions, and the ASTRAL to PVS translator, see [16].

6 Proof Assistance and Automation

After a specification is translated, the user must prove the inductive proof obligations discussed in section 4.3. In general, the proof obligations are undecidable so they require a fair amount of interaction with the prover. For timed properties, this interaction usually consists of setting up the sequences of transitions that are possible within the prover, proving that each sequence is indeed possible, and then showing that the time of the sequence is less than the required time. Portions of these proofs can be automated with appropriate PVS strategies [16], but the user must still direct PVS during much of the reasoning. There are some property types, however, that can oftentimes be proven fully automatically by PVS. After performing the proofs of several systems using the encoding, PVS strategies have been developed to assist the user in proving these types of properties.

6.1 Untimed Formulas

The *try-untimed* strategy was written to attempt the proofs of properties that do not involve time and only deal with combinations of state variables of a single process instance. For example, in the Elevator process type, one such property in the invariant section is "elevator_moving → ~door_moving". That is, whenever the elevator car is moving, the elevator door should not be in the process of opening or closing. This property was proved completely unassisted by the try-untimed strategy.

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, as discussed, Δ has been limited to a constant less than the duration of any transition, only a single transition end can occur within the interval. Figure 2 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.

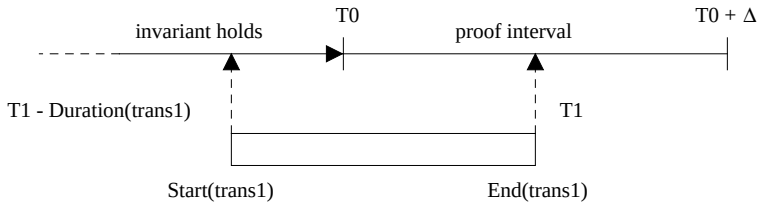

Fig. 2. Proof interval

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. Once again, since Δ was limited to less than the duration of any transition, the start of the transition occurred before T0, thus the property held at the start of the transition. From this point, 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 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.

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. For example, consider the elevator property "elevator_moving → ~door_open". That is, the door must be closed while the elevator car is moving. After running the try-untimed strategy, all the transition cases are proved except for the "door_stop" case. The door_stop transition, shown below, stops the door in either the open or closed position after a suitable length of time from when the door started 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′
```

The strategy fails for this case because it is possible for door_open to be set to true in the exit assertion and yet the value of elevator_moving is not stated in the entry assertion so can possibly be true if door_stop follows a transition in which elevator_moving is true. If elevator_moving is true and door_open is false when door_stop begins firing, then the formula will hold at the start of execution yet will not hold at the end of execution. In order to complete the proof of this property, it is necessary to consider the transitions that can fire immediately before door_stop. If the proof still cannot be completed, transitions must be considered further and further back in time. Eventually, the formula will be provable or a violation will occur.

6.2 Transition Sequence Generator

Since sequencing is so important to proving some properties, it is useful to provide the user with a tool to view the transition sequences that can occur in a given process type. Such a tool can be used to estimate time delays between states, help the user visualize the operation of the system, and in some cases can be used to prove simple system properties. Unlike graphical state-machine languages in which the successor information is part of the specification, in textual languages such as ASTRAL, sequencing cannot be determined without more in-depth analysis. In addition, determining whether one transition is the successor of another in ASTRAL 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.

The sequence generator first eliminates as many transition successors as possible. This is done by attempting the proof of an obligation trans1_not_trans2 for each pair

of transitions (trans1, trans2) as shown below. Note that this obligation only states that some transition must end between trans1 and trans2 and does not exclude trans1 or trans2 from firing. The obligation is sufficient, however, to prove that a transition besides trans1 and trans2 must fire in between any firing of trans1 and trans2. If only trans1 and trans2 fire in between t1 and t2, then since t2 - t1 is finite and the durations of trans1 and trans2 are constant and non-null, eventually a contradiction can be achieved by applying the theorem below repeatedly on an ever shortening interval.

```
trans1_not_trans2: THEOREM
    (FORALL (t1, t2):
        t1 + Duration(trans1) ≤ t2 AND
        Fired(trans1, t1) AND
        Fired(trans2, t2) IMPLIES
        (EXISTS (trans3, t3):
            t1 + Duration(trans1) <
                t3 + Duration(trans3) AND
            t3 + Duration(trans3) ≤ t2 AND
            Fired(trans3, t3))))
```

```
initial_not_trans1: THEOREM
    (FORALL (t1):
        Fired(trans1, t1) IMPLIES
        (EXISTS (trans2, t2):
            t2 + Duration(trans2) ≤ t1 AND
            Fired(trans2, t2)))
```

An obligation initial_not_trans1, as shown above, is also attempted to prove that each transition is not the first to fire after the initial state. The PVS strategies *try-seq-gen* and *try-seq-gen-0* were written to automatically discharge these obligations. The try-seq-gen strategy uses abstract machine axioms to introduce the entry and exit assertions of trans1, the entry assertion of trans2, and the fact that if nothing ended between the end of trans1 and the start of trans2, then all variable values remained constant during this time. Once all of this information is present, the strategy invokes the modified grind command as discussed for the try-untimed strategy. The try-seq-gen-0 strategy is similar but uses the initial clause of the process in place of the information about trans1.

Table 1 shows the results of using these strategies to compute the successors for each process type of a set of testbed systems developed in [16], which includes the elevator control system. 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.

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 door_stop transition, shown in section 6.1, constrains the value of door_moving, but does not constrain the value of elevator_moving.

When proving that the arrive transition, shown below, cannot follow door_stop, PVS does not have information about the value of elevator_moving at the start of door_stop, which is only derivable from the transitions preceding door_stop. Thus, PVS must assume an arbitrary symbolic value for elevator_moving. It is possible that elevator_moving is true, thus PVS cannot eliminate the possibility that arrive immediately follows door_stop. 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 arrive can immediately follow door_stop. The only possible predecessors to door_stop are open_door and close_door. Open_door sets elevator_moving to false in

its exit assertion, thus if open_door immediately precedes door_stop, arrive cannot follow door_stop. Similarly, it is possible to show that close_door must be preceded by door_stop, which is preceded by open_door. Thus, arrive cannot follow door_stop.

```
TRANSITION arrive
    ENTRY   [TIME:  arrive_dur]
            elevator_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 order to improve the accuracy 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 non-terminating procedure, however, whenever the second transition of a successor obligation 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, the end operator is used in the arrive transition to prevent arrive from following itself. In the proof of the successor obligation arrive_not_arrive, arrive fires at t1 and t2 and no other transition fires in between. By the last conjunct of arrive's entry assertion, there must be an end to move_up or move_down between the last time arrive ended (t1 + arrive_dur) and the next time it fires (t2), which contradicts the fact that no transition fires in between t1 and t2. This proof cannot be carried out without the use of the end operator. The definition of the end operator within PVS, however, is quite complex with several quantifiers, thus there is little hope that PVS could automatically prove such an obligation. For this reason, the modified grind used in the try-seq-gen strategies does not expand any of the timed operators, which prevents work from being wasted.

Table 1. Transition successors of testbed systems

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	420	84	198
	Sensor	6	3	3
Total		1978	585	986

When a transition is parameterized, such as the request_floor transition of the Elevator_Button_Panel process shown in section 2, 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 do not usually give any helpful information to the user since essentially any transition can follow any other.

Since the standard sequence generator proof obligations do not ordinarily produce a useful result for parameterized transitions, 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 successor 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. This is useful for finding the sequence of transitions in a single "thread". For example, by keeping the parameters fixed in the Central_Control process of the phone system of [5], the sequences of transitions that make up the evolution of a call for a particular phone can be computed. The numbers in Table 1 were computed using the parameterized extension. The numbers for the Elevator_Button_Panel,

Central_Control, and Controller processes are the only processes affected by this extension.

After the successors have been computed, the sequence generator constructs transition sequences based on input from the user, which includes 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. The sequence generation process is completely automatic and is available as a component of the ASTRAL Software Development Environment (SDE) [17]. The ASTRAL 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 from previous proof attempts can be quickly displayed.

For each sequence generated, an approximate running time of the sequence is constructed by analyzing the entry assertion of each transition. Entry assertions depend on the values of local and imported variables, the call/start/end times of local and imported transitions, and the current time in the system. Transitions that only depend on local variables and/or the start/end times of local transitions will always fire immediately after another transition. Transitions that reference the current time, however, may be delayed some amount of time before firing. For example, the door_stop transition, shown in section 6.1, fires at least t_move_door after the door starts moving. Similarly, transitions may wait indefinitely for a change to an imported variable, a call/start/end to an imported transition, or a call to a local transition from the external environment. The three types of delays are denoted *delay_T* for a time delay, *delay_O* for a delay because of the other processes in the system, and *delay_E* for a delay due to the external environment.

The sequence generator is complete (i.e. if a sequence is possible it will appear as a result) without the parameterized extension 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. In this case, utility was chosen over completeness.

The accuracy of the sequence generator can be improved by manually performing the proofs of those successor obligations that actually can 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.

As a simple example of a sequence generator query, consider the door_stop case that failed in the try-untimed proof of "elevator_moving $\rightarrow$ ~door_open" in section 6.1. The user may wish to view the predecessors to door_stop to see if the proof can be completed quickly or if a violation is possible involving the door_stop transition. Figure 3 shows the sequence generator dialog box and the second of the three sequences generated from the query.

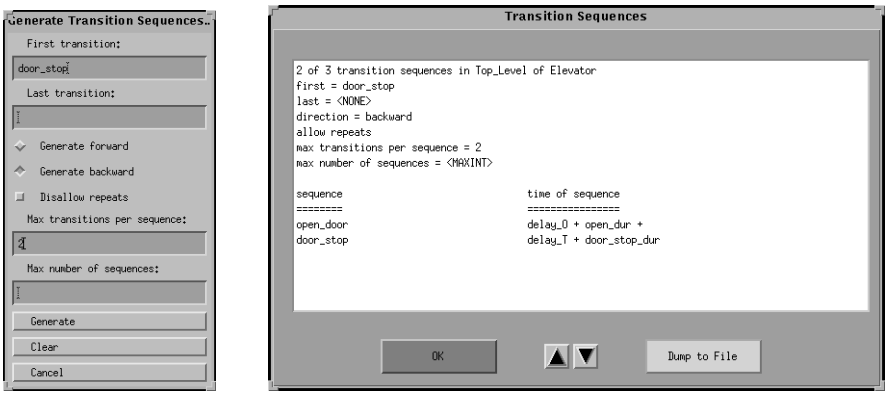

Fig 3. Sequence generator dialog box and query result

Three sequences are returned to the user, which show three possible predecessors to door_stop: close_door, open_door, and arrive. If close_door fires before door_stop, the door is closed when door_stop completes firing, thus the property trivially holds. The open_door transition sets elevator_moving to false, thus the property also trivially holds if open_door fires before door_stop. The arrive transition, shown earlier, requires the elevator car to be moving to fire. By the inductive hypothesis, the door is closed when it fires, thus if arrive precedes door_stop, the invariant can be violated because the elevator car is moving and door_stop sets door_open to true. Therefore, the user knows that to complete the proof, it must be shown that arrive cannot fire immediately before door_stop. The arrive case is another example of a successor that the sequence generator could not eliminate automatically and yet is not actually possible after further analysis. Thus, the user must consider the predecessors of arrive and continue the proof process in a similar manner until the property is proved. Additional uses of the transition sequence generator can be found in [16].

7 Related Work

The temporal logics TRIO [1] and DC [18] have been encoded into PVS as discussed in section 4.1. 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 [2] and Timed Transition Systems into HOL [13]. 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 [3] and [4], but this encoding is based on a definition of ASTRAL that has been developed independently at Delft University based on earlier ASTRAL work in [9] and [10]. The ASTRAL definition in [9] and [10] 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 [5] and [7] 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 axiomatization of the ASTRAL abstract machine. 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.

8 Conclusions and Future Work

This paper has discussed the adaptation of the PVS theorem prover for performing analysis of real-time systems written in the ASTRAL formal specification language. A number of issues were encountered during the encoding of ASTRAL that are relevant to the encoding of many real-time specification languages. These issues were presented as well as how they were handled in the ASTRAL encoding. A translator has been written that translates any ASTRAL specification into its corresponding PVS encoding. After performing the proofs of several systems using the encoding, PVS strategies have been developed to automate the proofs of certain types of properties. In addition, the encoding has been used as the basis for a transition sequence generator tool.

A number of issues still need to be addressed in future work. The implementation clause of ASTRAL, which is used to map relationships between upper and lower level specifications, needs to be incorporated into the translator, as well as the inter-level proof obligations used to show that an implementation is consistent with the level above. Currently, the refinement mechanism described in [6] is in a transitional phase, so its translation was postponed until the new refinement mechanism is in place.

A number of enhancements to the sequence generator can be added. For instance, it is useful to provide a more powerful interface. For example, a query interface could be added to answer queries such as whether a given transition can ever occur between two other specified transitions. It is also possible to construct a symbolic expression for the values of the state variables at the end of each sequence by examining the entry and exit assertions of each transition.

In general, more proofs need to be performed for different ASTRAL systems using their PVS translations. In studying the proofs performed for many systems, more proof patterns may be discovered that can be incorporated into suitable PVS strategies. The patterns may also lead to the definition of useful lemmas that can be proven in advance and added to the ASTRAL-PVS library for future use. It is also worthwhile to investigate whether the structure of the ASTRAL specification determines which lemmas and strategies are most applicable to a given formula type.

References

1. Alborghetti, A., A. Gargantini, and A. Morzenti. Providing automated support to deductive analysis of time critical systems. *Proc. 6th European Software Engineering Conf.*, 1997.
2. Archer, M. and C. Heitmeyer. Mechanical verification of timed automata: a case study. *Proc. Real-Time Technology and Applications Symp.*, pp. 192-203, 1996.

3. Bun, L. Checking properties of ASTRAL specifications with PVS. *Proc. 2nd Annual Conf. of the Advanced School for Computing and Imaging*, pp. 102-107, 1996.
4. Bun, L. Embedding Astral in PVS. *Proc. 3rd Annual Conf. of the Advanced School for Computing and Imaging*, pp. 130-136, 1997.
5. Coen-Porisini, A., C. Ghezzi, and R.A. Kemmerer. Specification of realtime systems using ASTRAL. *IEEE Transactions on Software Engineering*, 23(9): 572-598, 1997.
6. Coen-Porisini, A., R.A. Kemmerer, and D. Mandrioli. A formal framework for ASTRAL inter-level proof obligations. *Proc. 5th European Software Engineering Conf.*, pp. 90-108, 1995.
7. Coen-Porisini, A., R.A. Kemmerer, and D. Mandrioli. A formal framework for ASTRAL intralevel proof obligations. *IEEE Transactions on Software Engineering*, 20(8): 548-561, 1994.
8. Crow, J., S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. *Workshop on Industrial-Strength Formal Specification Techniques*, 1995.
9. Ghezzi, C. and R.A. Kemmerer. ASTRAL: an assertion language for specifying realtime systems. *Proc. 3rd European Software Engineering Conf.*, pp. 122-140, 1991.
10. Ghezzi, C. and R.A. Kemmerer. Executing formal specifications: the ASTRAL to TRIO translation approach. *Proc. Symp. on Testing, Analysis, and Verification*, 1991.
11. Gordon, M. Notes on PVS from a HOL perspective. Available at <http://www.cl.cam.ac.uk/users/mjcg/PVS.html>, 1995.
12. Gordon, M.J.C. and T.F. Melham (eds.). Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press, 1993.
13. Hale, R., R. Cardell-Oliver, and J. Herbert. An embedding of timed transition systems in HOL. *Formal Methods in System Design*, 3(1-2): 151-174, 1993.
14. Heitmeyer, C. and D. Mandrioli (eds.). Formal methods for real-time computing. John Wiley, 1996.
15. Kaufmann, M. and J. Strother Moore. ACL2: an industrial strength version of Nqthm. *Proc. 11th Annual Conf. on Computer Assurance*, pp. 23-34, 1996.
16. Kolano, P.Z. Tools and techniques for the design and systematic analysis of real-time systems. Ph.D. Thesis, University of California, Santa Barbara, 1999.
17. Kolano, P.Z., Z. Dang, and R.A. Kemmerer. The design and analysis of real-time systems using the ASTRAL software development environment. *Annals of Software Engineering*, 7, 1999.
18. Skakkebaek, J.U. and N. Shankar. Towards a duration calculus proof assistant in PVS. *3rd Int. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp. 660-679, 1994.
19. Spivey, J.M. Specifying a real-time kernel. *IEEE Software*, 7(5): 21-28, 1990.
20. Young, W.D. Comparing verification systems: interactive consistency in ACL2. *Proc. 11th Annual Conf. on Computer Assurance*, pp. 35-45, 1996.