Running Title:

The ASTRAL Software Development Environment

Authors:

Paul Kolano
Computer Science Department
University of California
Santa Barbara, CA 93106
805-893-4394
805-893-8553 (fax)
kolano@cs.ucsb.edu

Zhe Dang
Computer Science Department
University of California
Santa Barbara, CA 93106
805-893-4394
805-893-8553 (fax)
dang@cs.ucsb.edu

Richard Kemmerer
Computer Science Department
University of California
Santa Barbara, CA 93106
805-893-4232
805-893-8553 (fax)
kemm@cs.ucsb.edu

The Design and Analysis of Real-Time Systems Using the ASTRAL Software Development Environment

Paul Z. Kolano

Zhe Dang

Richard A. Kemmerer

Reliable Software Group
Computer Science Department
University of California
Santa Barbara, CA 93106 USA

Abstract

ASTRAL is a formal specification language for real-time systems. It is intended to support formal software development and, therefore, has been formally defined. The structuring mechanisms in ASTRAL allow one to build modularized specifications of complex systems with layering. A real-time system is modeled by a collection of state machine specifications and a single global specification.

This paper discusses the ASTRAL Software Development Environment (SDE), which is an integrated set of design and analysis tools based on the ASTRAL formal framework. The tools that make up the support environment are a syntax-directed editor, a specification processor, a verification condition generator, a browser kit, a model checker, and a mechanical theorem prover.

1. Introduction

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 standalone 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 formal software development environment presented in this paper is based on ASTRAL, which is a formal specification language for real-time systems. ASTRAL is intended to support formal software development and, therefore, has been formally defined. The structuring mechanisms in ASTRAL allow one to build modularized specifications of complex systems with layering. A real-time system is modeled by a collection of state machine specifications and a single global specification. The ASTRAL Software Development Environment (SDE) is a tool for the ASTRAL language, which assists in the design, analysis, and reuse of ASTRAL specifications.

The main intent of this paper is to give the reader an overview of the ASTRAL SDE and a status report on its development. In addition, some updates to the ASTRAL language that have not appeared in previously published work, but that have been incorporated into the ASTRAL SDE, are discussed. In section 2, a brief overview of the ASTRAL specification language is presented. In section 3, the ASTRAL SDE is presented. In section 4, related systems are discussed. Finally, in section 5, conclusions drawn from this work and further areas of research are presented.

2. ASTRAL Overview

A railroad crossing is used as a pedagogical example throughout the remainder of this paper to illustrate various features of ASTRAL and of the ASTRAL SDE. The system description is taken from [Heitmeyer and Lynch 1994]. The system consists of a set of railroad tracks that intersect a street where 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 entire region between the sensors and the crossing exit is denoted by R and the crossing itself, which is a subinterval of R, is denoted by I. Figure 1 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. The complete ASTRAL specification of the railroad system is located in the appendix.

In ASTRAL, 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. Each process instance in the system executes concurrently and asynchronously with all the other process instances. In the railroad specification, there are two process types, with one instance of the Gate process type and n instances of the Sensor process type, where n is the number of tracks. 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. Figure 2 presents the syntactic structure for an ASTRAL specification.

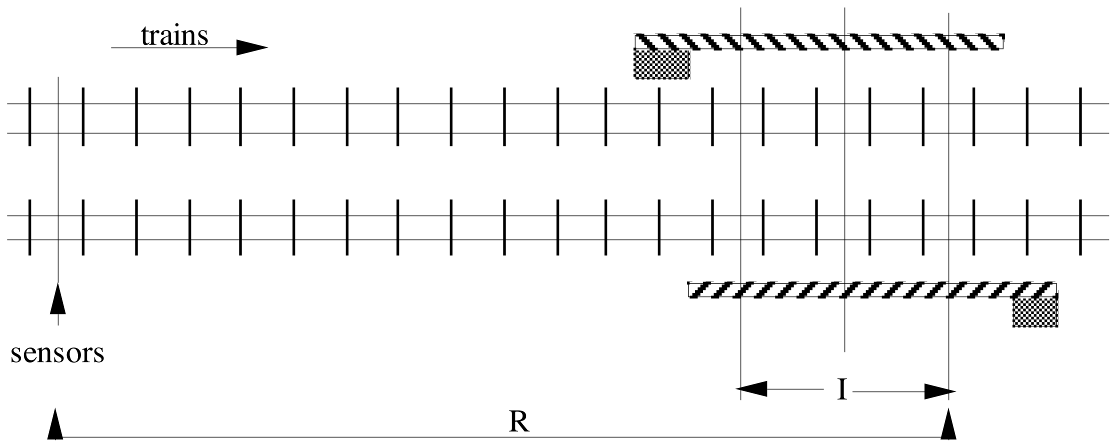


Figure 1. The railroad crossing.

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.

The process being specified is thought of as being in various *states*, with one state differentiated from another by the values of the *state variables.* In the railroad specification, the Gate and Sensor process types have a single variable each. The position variable in the Gate process type is an enumerated variable that specifies the current position of the gate. The train_in_R variable in the Sensor process type is a Boolean variable that indicates whether a train has been detected by the sensor.

State variables of a given process instance can only be changed by *state transitions* of that process instance. 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. 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. Only one transition may be executing on a given process instance at any time. A transition is executed as soon as it is enabled assuming no other transition for 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.

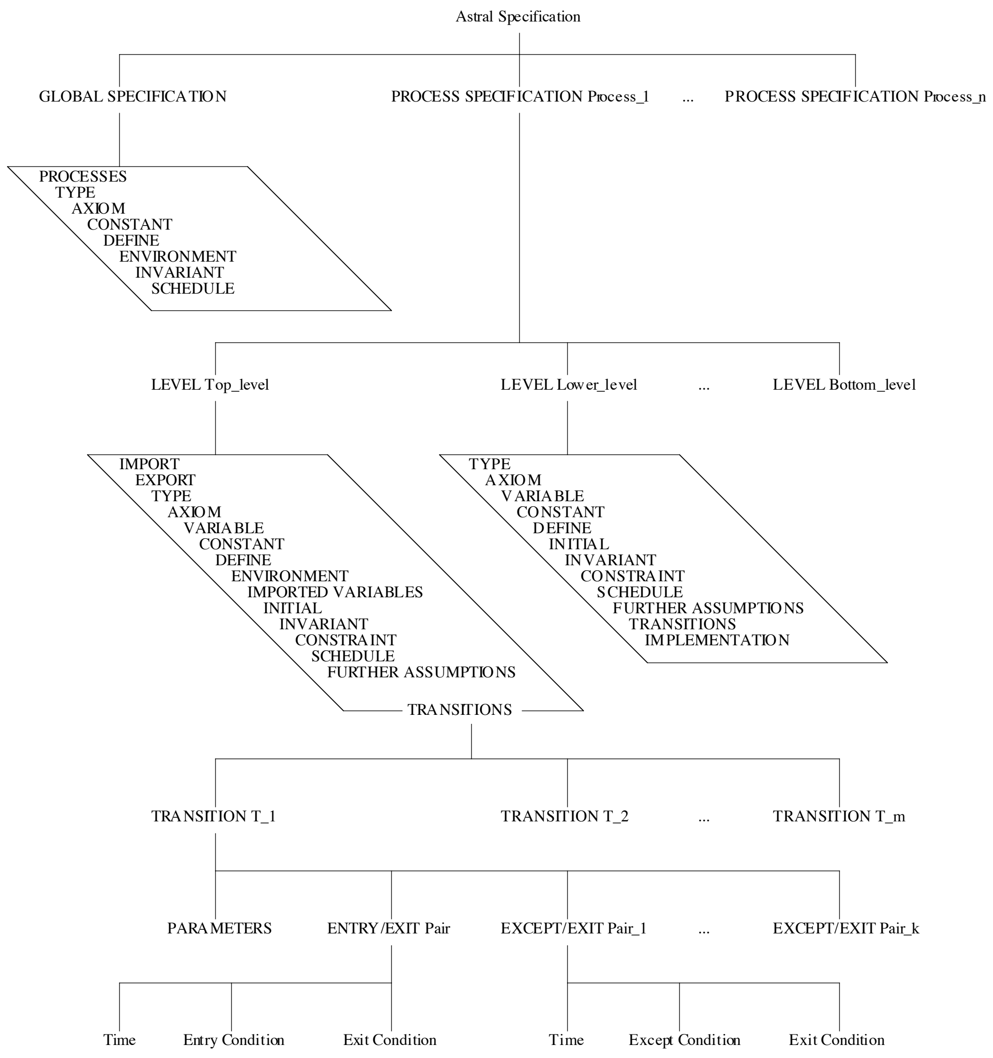


Figure 2. The ASTRAL hierarchy.

Every process can export both state variables and transitions; as a consequence, the state variables are readable by other processes while the transitions are executable from the external environment. For example, the Sensor process type exports the variable train_in_R, which is referenced by the Gate process type, and the transition enter_R, which is invoked when a train trips the sensor. Inter-process communication is accomplished by broadcasting the values of exported variables and the start and end times of exported transitions. A special variable called *now* is used to denote the current time. The time domain in ASTRAL is the non-negative real numbers. The value of now is zero at system initialization time. Now progresses independently of the execution of any process instance in the system.

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 both the other processes in the system and the external environment that interacts with the system. The behavior of other processes in the system is expressed by means of *imported variable clauses*, which describe patterns of changes to the values of imported variables and timing information about transitions exported from other processes. For example, the imported variable clause of the Gate process type states that once a sensor reports a train, it will keep reporting a train at least as long as it takes the fastest train to cross the region. This assumption is needed to guarantee the liveness requirement of 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 that can be reached 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 other system processes and the external environment behave as assumed. For example, the global schedule states the safety and liveness requirements of the system. That is, that the gate will be down before the fastest train can reach the crossing and that the gate will be up within a reasonable amount of time from when the slowest train could have exited the crossing.

Invariants and Schedules are proved over all possible executions of a system. A system execution is a set of process executions that contains one process execution for each process instance in the system. A process execution for a given process instance is a history of events on that instance. There are four types of events in ASTRAL. A call event occurs for an exported transition tr1 at a time t1 iff tr1 was called at t1. A start event occurs for a transition tr1 at a time t1 iff tr1 fires at t1. Similarly, an end event occurs if tr1 ends at t1. The last type of event is a change event. A change event occurs for a variable v1 at a time t1 iff v1 changes value at t1. Note that change events can only occur when an end event occurs for some transition.

An example of a transition is the exit_I transition of the Sensor process type:

```
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
```

Exit_I is enabled when a train is in R and enough time has elapsed since the train first entered R for the slowest train to get past the crossing ((dist_R_to_I + dist_I_to_out) / min_speed – exit_dur). In ASTRAL, *Call(T)* indicates when an exported transition T is invoked and *Start(T)* indicates when it actually fires. Note that the call time of enter_R is actually when the train entered R, but under the assumptions of the Sensor environment, enough time elapses between successive calls of enter_R to guarantee that the start time is equivalent to the call time in this case. Start was used in place of call so that the local invariant, which must hold regardless of the behavior of the external environment, could be proved. The duration of exit_I is exit_dur and is indicated in the "TIME" expression of the transition. When exactly exit_dur time has elapsed since a start of exit_I, the train_in_R variable is reset to false.

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. A type, constant, variable, or transition of P_L representing the implementation of a corresponding term in P_U is referred to as a *mapped* type, constant, variable, or transition. 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 is of the form $T_U == A_1 \& T_{L.1} \mid A_2 \& T_{L.2} \mid ... \mid A_n \& T_{L.n}$. This is defined such that when the upper level transition T_U fires, one and only one lower level transition $T_{L.j}$ fires. In addition, $T_{L.j}$ can only fire when both its entry assertion and its associated "guard," A_j, are true.

A sequence mapping is of the form T_U == WHEN Entry_L DO $T_{L.1}$ BEFORE $T_{L.2}$ BEFORE ... BEFORE $T_{L.n}$ OD. This 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, to preserve this property, mapped variables of P_L can be modified only when the sequence implementing T_U ends. The proof obligations for transition mappings are discussed in a later section. The details of transition mappings and process refinement are presented in [Coen-Porisini *et al*. 1995].

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 3a shows two systems, S_1 and S_2. S_1 exports transitions T1 and T2 and state variables x1, x2 and x3, while S_2 exports transition T3 and state variables y1 and y2. 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 3b, transitions T1 and T3 are no longer exported, since the events that trigger them are now represented by particular values of y2 and x1, x3, respectively. Thus, the composed system, C, will export only transition T2. That is, the external environment of C can call only transition T2 (see figure 3c).

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 in a later section. The details of the composition clause and the process of automatically composing ASTRAL specifications are presented in [Coen-Porisini and Kemmerer 1993].

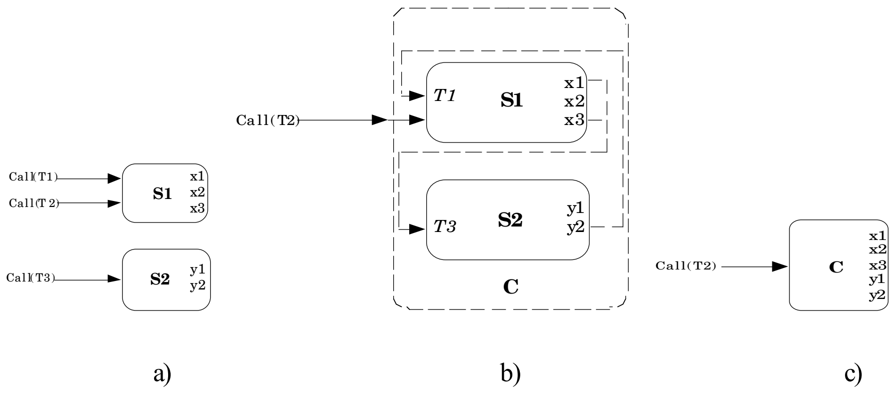


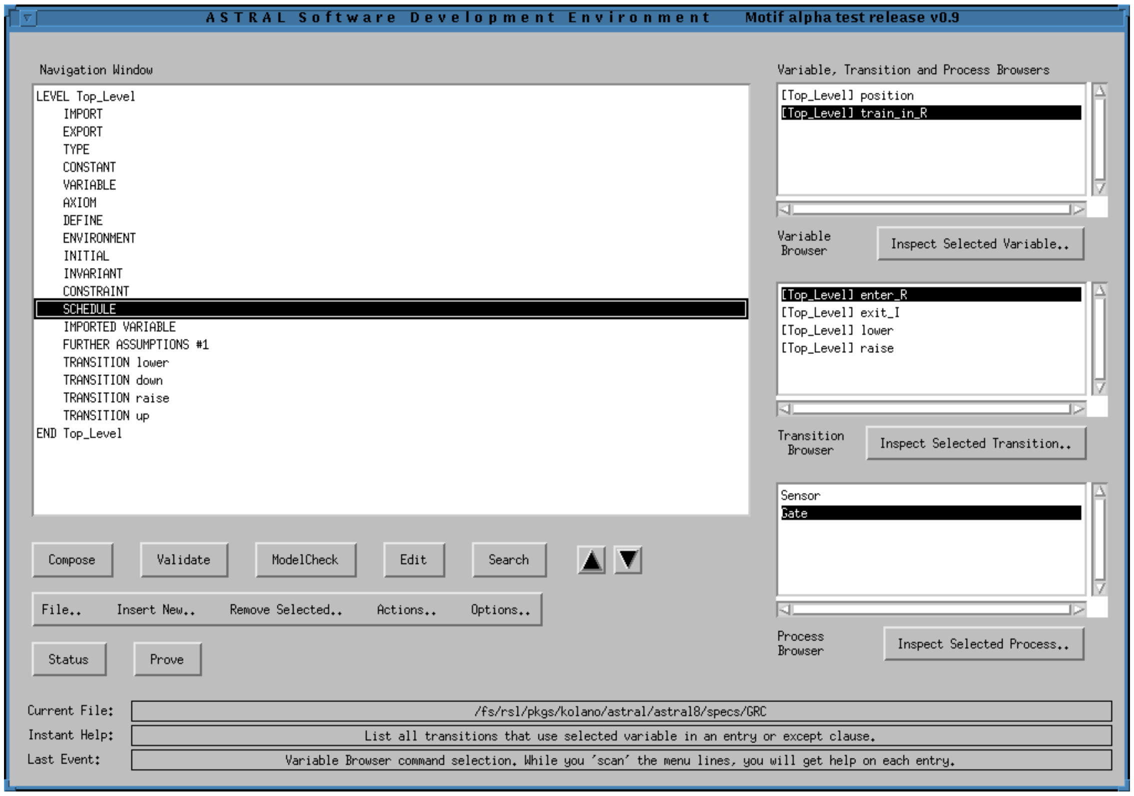
Figure 3. The composition of S_1 and S_2 into C.

An introduction and complete overview of the ASTRAL language can be found in [Coen-Porisini *et al.* 1997].

3. SDE Overview

Figure 4 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. For instance, figure 4 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 4, 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.

The top row of buttons in the middle of the SDE are for the most commonly invoked operations. Clicking on "Compose" allows the user to work with multiple specifications and composition specific clauses. 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 "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 "Edit" button brings up the syntax-directed editor on the section highlighted in the navigation window. Finally, the "Search" button brings up the search window, which can be used to search and replace expressions throughout the current specification. The lower row of buttons is for checking the status of and invoking a prover on the specification. The "Status" button brings up the proof manager, which keeps track of changes made to the specification and the current status of proofs. The "Prove" button generates the PVS translation of the specification and invokes the PVS theorem prover.


A S T R A L   S o f t w a r e   D e v e l o p m e n t   E n v i r o n m e n t
Motif alpha test release v0.9
Navigation Window
Variable, Transition and Process Browsers
Variable Browser
Inspect Selected Variable..
Transition Browser
Inspect Selected Transition..
Process Browser
Inspect Selected Process..
Compose
Validate
ModelCheck
Edit
Search
File..
Insert New..
Remove Selected..
Actions..
Options..
Status
Prove

```
LEVEL Top_Level
    IMPORT
    EXPORT
    TYPE
    CONSTANT
    VARIABLE
    AXIOM
    DEFINE
    ENVIRONMENT
    INITIAL
    INVARIANT
    CONSTRAINT
    SCHEDULE
    IMPORTED VARIABLE
    FURTHER ASSUMPTIONS #1
    TRANSITION lower
    TRANSITION down
    TRANSITION raise
    TRANSITION up
END Top_Level
```

```
[Top_Level] position
[Top_Level] train_in_R
```

```
[Top_Level] enter_R
[Top_Level] exit_I
[Top_Level] lower
[Top_Level] raise
```

```
Sensor
Gate
```

Current File: /fs/rsl/pkgs/kolano/astral/astral8/specs/GRC
Instant Help: List all transitions that use selected variable in an entry or except clause.
Last Event: Variable Browser command selection. While you 'scan' the menu lines, you will get help on each entry.

Figure 4. The ASTRAL SDE.

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 doesn't 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.

3.1. Editor

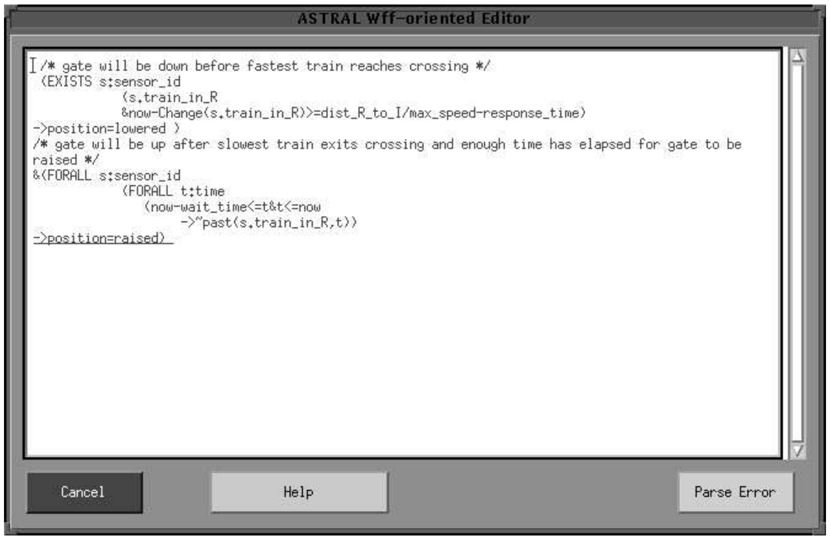
The SDE editor provides only the most basic functionality of common general purpose editors such as vi or emacs; however, it is rarely the case that an ASTRAL section is so large as to require the additional functionality provided by general purpose editors. More importantly, the syntax checking, automatic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

3.1.1. 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 shows the popup window that appears when the edit function is invoked on the section highlighted in the navigation window of figure 4. 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. 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 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.

3.1.2. Formatting

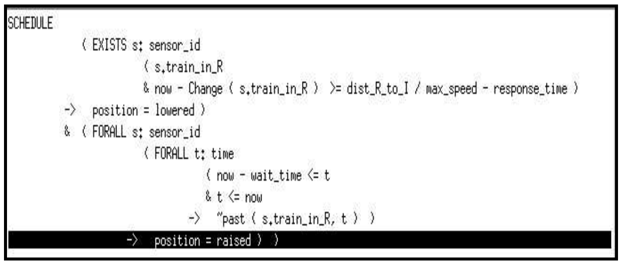
When first writing specifications, it is more important to concentrate on the content rather than the readability of the specification. This doesn't mean, however, that readability is unimportant. In fact, an unreadable specification is likely to contribute unnecessary confusion, errors, and additional 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.



```
/* 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)
```

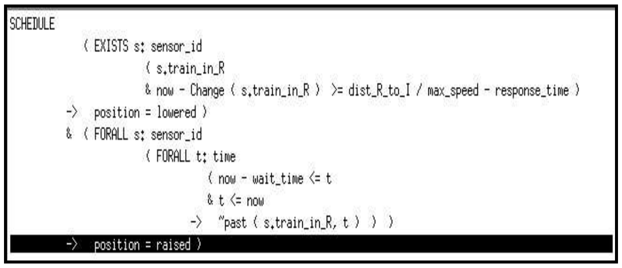
Figure 5. Editor window for the Gate schedule.

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. This parenthesis can be placed in a number of different locations to syntactically fix the problem. Figure 6a shows the result of formatting in the edit window of figure 5 (after adding the missing parenthesis to the end). As can be seen from figure 6a, by adding a parenthesis to the end of the text the highlighted implication is placed in the wrong scope. Figure 6b shows the same formula with the parenthesis correctly placed.



```
SCHEDULE
         ( EXISTS s: sensor_id
                  ( s.train_in_R
                  & now - Change ( s.train_in_R )  >= dist_R_to_I / max_speed - response_time )
       ->  position = lowered )
       &  ( FORALL s: sensor_id
                  ( FORALL t: time
                          ( now - wait_time <= t
                          & t <= now
                        ->  ~past ( s.train_in_R, t )  )
                ->  position = raised )  )
```

a)

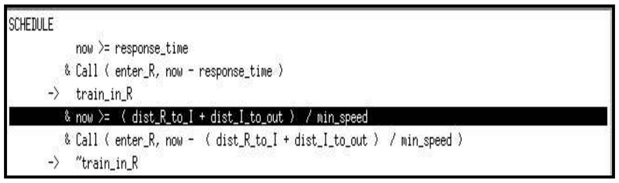


```
SCHEDULE
         ( EXISTS s: sensor_id
                  ( s.train_in_R
                  & now - Change ( s.train_in_R )  >= dist_R_to_I / max_speed - response_time )
       ->  position = lowered )
       &  ( FORALL s: sensor_id
                  ( FORALL t: time
                          ( now - wait_time <= t
                          & t <= now
                        ->  ~past ( s.train_in_R, t )  )  )
       ->  position = raised )
```

b)

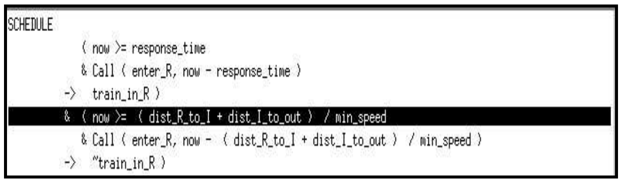
Figure 6. Formatted forms of Gate schedule with misplaced and correctly placed parenthesis

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 7a, the highlighted conjunction incorrectly binds more tightly than the two implications surrounding it. In figure 7b, 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.



```
SCHEDULE
          now >= response_time
        & Call ( enter_R, now - response_time )
     ->   train_in_R
        & 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
```

a)



```
SCHEDULE
          ( now >= response_time
          & Call ( enter_R, now - response_time )
       ->  train_in_R )
       &  ( 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 )
```

b)

Figure 7. Formatted forms of Sensor invariant with scoping error and correction.

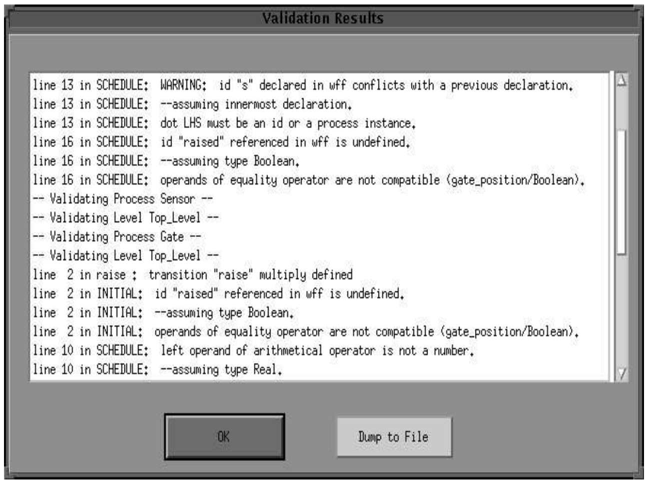
3.1.3. Search and Replace

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.

3.2. Validation

One of the most valuable tools that the SDE has to offer is the validation mechanism. 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 8 shows a sample validation results window, which demonstrates some of the different types of errors that are reported.



```
line 13 in SCHEDULE:  WARNING:  id "s" declared in wff conflicts with a previous declaration.
line 13 in SCHEDULE:  --assuming innermost declaration.
line 13 in SCHEDULE:  dot LHS must be an id or a process instance.
line 16 in SCHEDULE:  id "raised" referenced in wff is undefined.
line 16 in SCHEDULE:  --assuming type Boolean.
line 16 in SCHEDULE:  operands of equality operator are not compatible (gate_position/Boolean).
-- Validating Process Sensor --
-- Validating Level Top_Level --
-- Validating Process Gate --
-- Validating Level Top_Level --
line  2 in raise :  transition "raise" multiply defined
line  2 in INITIAL:  id "raised" referenced in wff is undefined.
line  2 in INITIAL:  --assuming type Boolean.
line  2 in INITIAL:  operands of equality operator are not compatible (gate_position/Boolean).
line 10 in SCHEDULE:  left operand of arithmetical operator is not a number.
line 10 in SCHEDULE:  --assuming type Real.
```

Figure 8. 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.

3.3. Composing ASTRAL Specifications

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 [Coen-Porisini and Kemmerer 1993]) 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 only required to perform one step in the construction of 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 9 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.

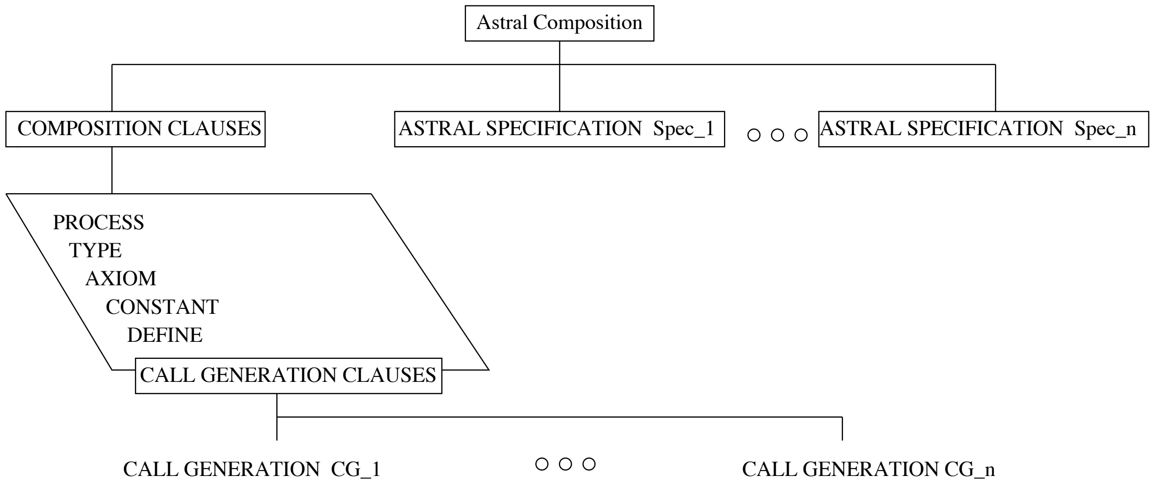


Figure 9. The composition hierarchy.

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 [Coen-Porisini and Kemmerer 1993]. 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 [Coen-Porisini *et al*. 1997]):

```
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, one must check 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. To perform this process, the environment clause is first modified according to the call generation clauses. It is then transformed into an equivalent CNF expression and each conjunct is checked for calls to exported transitions. If no such calls are found in a conjunct, then the conjunct is conjoined to the imported variable clause. If a call is found, then the conjunct remains in the environment 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 [Coen-Porisini and Kemmerer 1993]. Thus, the user is completely relieved of the burden of producing the composite specification by the build function of the SDE.

3.4. Proof Obligations

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 the specification of level i is consistent and satisfies the stated critical requirements for each of the processes, as well as for the global system. 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 [Coen-Porisini *et al*. 1994], [Coen-Porisini *et al*. 1995], and [Coen-Porisini and Kemmerer 1993], 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 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).

The inter-level proofs consist of showing that the mapping for 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 duration of each $T_{L.j}$ is equal to the duration of T_U, and that the effect of each $T_{L.j}$ is equivalent to the effect of T_U. For sequences, it must be shown that the sequence is enabled if and only if T_U is enabled, that the duration of the sequence (including any initial delay after $Entry_L$ is true) is equal to the duration of T_U, and that the effect of the sequence is equivalent to the effect of T_U. The exact proof obligations for simple sequences and selections are given in [Coen-Porisini *et al*. 1995], 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 == WHEN Entry_L DO  (   A_0 &  (   A_1 & T_L.1
                                        |   A_2 & (T_L.2 BEFORE T_L.3))
                            |   A_4 & T_L.4) BEFORE T_L.5 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($T_{L.j}$). This is another example where even though the exact details of these proof obligations were not discussed in previous work, the updated definition of the ASTRAL language has been incorporated into the proof obligation generator of the SDE.

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.

3.5. Browsers

The process, transition, and variable browsers in the right portion of figure 4 enable the user to view various relationships between the three types of items. The complete impact that the browsers will have on the specification writing process has not yet been realized. It is clear, however, that there will certainly be times when a user that is modifying a system may wish to know which processes import a particular variable, which transitions set the same variables that are set by a selected transition, or any other number of relationships that may be helpful at a particular moment. Uncovering such relationships manually, however, can be a time consuming task. The browsers make use of symbol tables maintained during editing to easily ascertain and display the appropriate information. To illustrate the types of queries available, figure 10 shows the transition browser queries.

A query is executed by selecting an item (i.e. variable, transition, or process) in one of the browsers and then choosing one of the queries in the “Inspect Selected..” menu beneath that browser. 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.

The browsers in figure 4 demonstrate the results of a sample browser session. First, all processes declared in the current specification were listed with the special "All Processes Defined" process browser query, which does not require any browser item to be highlighted. 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. In a pedagogical specification such as the railroad crossing, the information obtained by the browsers could most likely be obtained manually in a similar amount of time. In larger specifications, however, the difference in speed and accuracy between obtaining the information manually and obtaining it using the browsers will be much more significant.

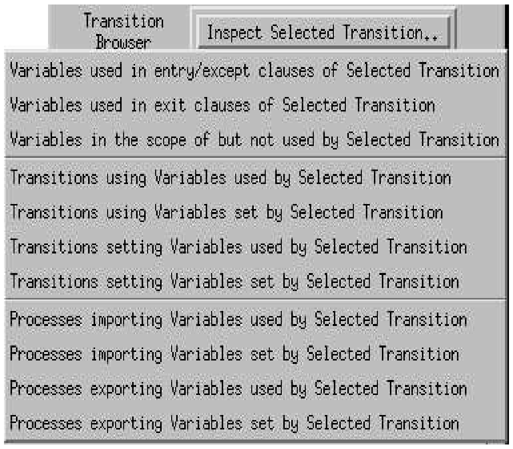

Transition Browser
Inspect Selected Transition..
Variables used in entry/except clauses of Selected Transition
Variables used in exit clauses of Selected Transition
Variables in the scope of but not used by Selected Transition
Transitions using Variables used by Selected Transition
Transitions using Variables set by Selected Transition
Transitions setting Variables used by Selected Transition
Transitions setting Variables set by Selected Transition
Processes importing Variables used by Selected Transition
Processes importing Variables set by Selected Transition
Processes exporting Variables used by Selected Transition
Processes exporting Variables set by Selected Transition

Figure 10. Transition browser queries.

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. For example, consider an invariant with a property of the form P → var = val, for some ASTRAL predicate P, variable var, and value val. The prover can select var in the variable browser and query the transitions that use var in an exit assertion. Each transition that is displayed in the transition browser can then be double clicked on and its specification will be shown in the navigation window. This allows the exit assertion to be examined to determine which transitions change var to a value other than val. The prover can then check whether P implies the entry assertion of any of those transitions and if such a transition is found, then the invariant can be violated if P still holds at the end of that transition.

3.6. Theorem Prover

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. Rather than implementing a theorem prover for ASTRAL from scratch, it was decided to take advantage of an existing general purpose theorem prover modified for use with ASTRAL. After investigating a number of theorem provers, the Prototype Verification System (PVS) [Crow *et al*. 1995] was considered to be ideal for ASTRAL because of its powerful typing system, higher-order facilities, heavily automated decision procedures, and intuitive reasoning style.

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. 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 $C_1, ..., C_n$ 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 also allows the user to define *strategies*, which are collections of prover commands that can be developed to automate frequently occurring proof patterns.

To use PVS to prove properties of ASTRAL systems, the semantics of ASTRAL needed to be encoded in the PVS logic. These semantics include both the axiomatization of the ASTRAL abstract machine as well as the precise definition of each ASTRAL operator. The abstract machine describes admissible system executions, where a system execution consists of the call, start, and end times of each transition. For example, one axiom of the abstract machine states that if no transition has ended within a given interval on a particular process, then each variable of that process keeps the same value throughout the interval. This axiom is called the *vars_no_change* axiom and defines the ASTRAL property that variables can only change values when transitions end.

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

Note that the function Vars_No_Change(t1, t2) in the vars_no_change definition is a specification-dependent function constructed by the translator stating that the value of each variable of the process is the same at t1 and t2. The vars_no_change axiom is one of ten axioms in the axiomatization of the ASTRAL abstract machine.

In addition to the abstract machine axioms, the ASTRAL semantics also include the definitions of each ASTRAL operator. Since ASTRAL expressions can have different values depending on the current time in the system (i.e. now), each ASTRAL operator has been defined such that the evaluation of its operands are delayed until a temporal context (i.e. a value of now) is given. For example, the past operator, which normally takes an ASTRAL formula of type T and a time is defined as:

```
Past(v1: [time → T], at1: [time → time])
      (t1: {t1: time | at1(t1) ≤ t1}): T = v1(at1(t1))
```

When a time is applied to a past expression, the time operand, which can be an arbitrary formula, is evaluated to a specific time and then the value of the formula operand is evaluated at the resulting time. Note that the type of t1 has been limited such that at1(t1) ≤ t1 to enforce the semantics of the past operator, which says that past cannot be applied at times beyond now (e.g., past(v, now+1) is not allowed). The standard Boolean and arithmetic connectives have been defined similarly to the ASTRAL operators in that their normal operands of type T become functions of type [time → T]. For example, conjunction is defined as:

```
AND(b1: [time → bool], b2: [time → bool])(t1: time): bool = b1(t1) AND b2(t1)
```

With these definitions, expressions in the encoding can be constructed exactly as they are in ASTRAL. When an evaluation time is applied to an expression, the time is propagated to each individual operator and operand in the expression. For example, (A AND B)(t) becomes A(t) AND B(t). The translator component of the SDE can translate any ASTRAL specification into its PVS representation. To translate an ASTRAL expression, its parse tree is traversed and the appropriate PVS definition is substituted for each operator. The two formulas below show the environment clause of the Sensor process in ASTRAL and PVS forms. The left formula is in ASTRAL notation and the right formula is the PVS translation that was automatically generated from the ASTRAL form.

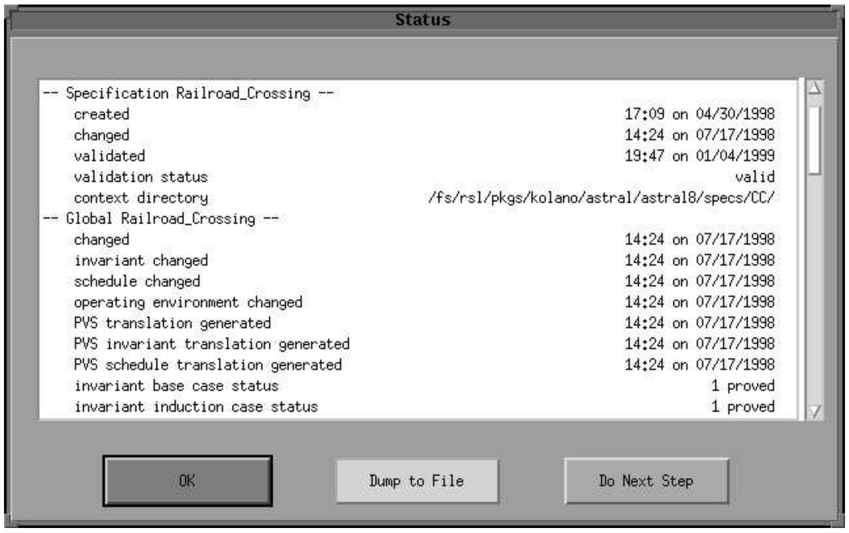
```
Environment                                 Environment: [time → bool] =
   Call(enter_R, now) &                        Call1(enter_r, now) AND
   EXISTS t: time (                            (EX! (t: [time → time]):
      t ≥ 0 &                                     t ≥ const(0) AND
      t ≤ now &                                   t ≤ now AND
      Call2(enter_R, t)) →                        Calln(const(2), enter_r, t)) IMPLIES
         Call(enter_R) - Call2(enter_R) >            Call1(enter_r) - Calln(const(2), enter_r) >
            (dist_R_to_I + dist_I_to_out) /             (const(dist_r_to_i) + const(dist_i_to_out)) /
               min_speed                                   const(min_speed)
```

Note that the function "const" is used to denote numeric and symbolic values that are constant over time. When the above expression is evaluated at a specific time, the const "drops out". Also note that user-defined identifiers such as "enter_R" have been changed to a lower case form in the translation. This is because PVS is case-sensitive while ASTRAL is case-insensitive, thus the choice was made to always translate user-defined identifiers into a lower case form. The axiomatization and the operator definitions have been incorporated into an ASTRAL-PVS library, included as part of the SDE. For a complete description of the axiomatization and the PVS encoding, see [Kolano 1998].

A prototype proof manager has been implemented that tracks changes to ASTRAL specifications and automatically generates PVS translations when appropriate. The "Prove" button of the SDE generates the translations for those sections of the specification that have changed more recently than they have been translated. The "Prove" button then invokes PVS on the appropriate context directory and retrieves the PVS proof status. In the future, the proof manager will perform other tasks such as guiding the user as to the design or analysis step to perform next. The

"Status" button brings up the proof manager window, which is shown in figure 11 for the railroad crossing specification.

PVS was used to prove a portion of the requirements of the railroad crossing specification. These proofs were the first extensive test of the ASTRAL-PVS encoding on real-time properties spanning a fairly large time interval (i.e. the time between an event and its required response). The proof of the gate safety property was proven for one of the two worst case scenarios. Essentially, to prove that the gate is lowered in time, the proof can be split into cases such that when a train arrives, the gate can be either idle with position being in one of four states, or the gate can be executing a transition. The longest time for the gate to be lowered occurs when either the up or raise transition is firing (depending on the values of up_dur, raise_dur, and t_lower). The raise case was proved using 258 prover commands and required 25 lemmas to be provided to the prover during the course of the proof. The invariant of the sensor process, which is used to guarantee that the sensors act as assumed in the gate's imported variable clause, was completely proved using 490 prover commands and providing 38 lemmas.



```
-- Specification Railroad_Crossing --
    created                                          17:09 on 04/30/1998
    changed                                          14:24 on 07/17/1998
    validated                                        19:47 on 01/04/1999
    validation status                                              valid
    context directory           /fs/rsl/pkgs/kolano/astral/astral8/specs/CC/
-- Global Railroad_Crossing --
    changed                                          14:24 on 07/17/1998
    invariant changed                                14:24 on 07/17/1998
    schedule changed                                 14:24 on 07/17/1998
    operating environment changed                    14:24 on 07/17/1998
    PVS translation generated                        14:24 on 07/17/1998
    PVS invariant translation generated              14:24 on 07/17/1998
    PVS schedule translation generated               14:24 on 07/17/1998
    invariant base case status                                  1 proved
    invariant induction case status                             1 proved
```

Figure 11. The proof manager window.

Although nothing was found that could not be derived using only the abstract machine axioms, the proofs were sometimes overly complicated by the lack of supporting lemmas. Perhaps the most pervasive example of this was in applying the vars_no_change axiom. To show that no variables change between two times, it must be shown that no transition ends between those two times. To show that a transition does not end, its entry assertion is usually compared to the known state in the system to achieve a contradiction. To determine the state of the system at a given time, however, it is almost always necessary to apply the vars_no_change axiom again. This creates a circular scenario, which, although resolvable with complex case analysis on the firing times and durations of transitions, makes the proofs long and complex and causes much repetition and waste of time. The discovery of this problem during the railroad proofs led to the definition of the no_trans_fire lemma shown below:

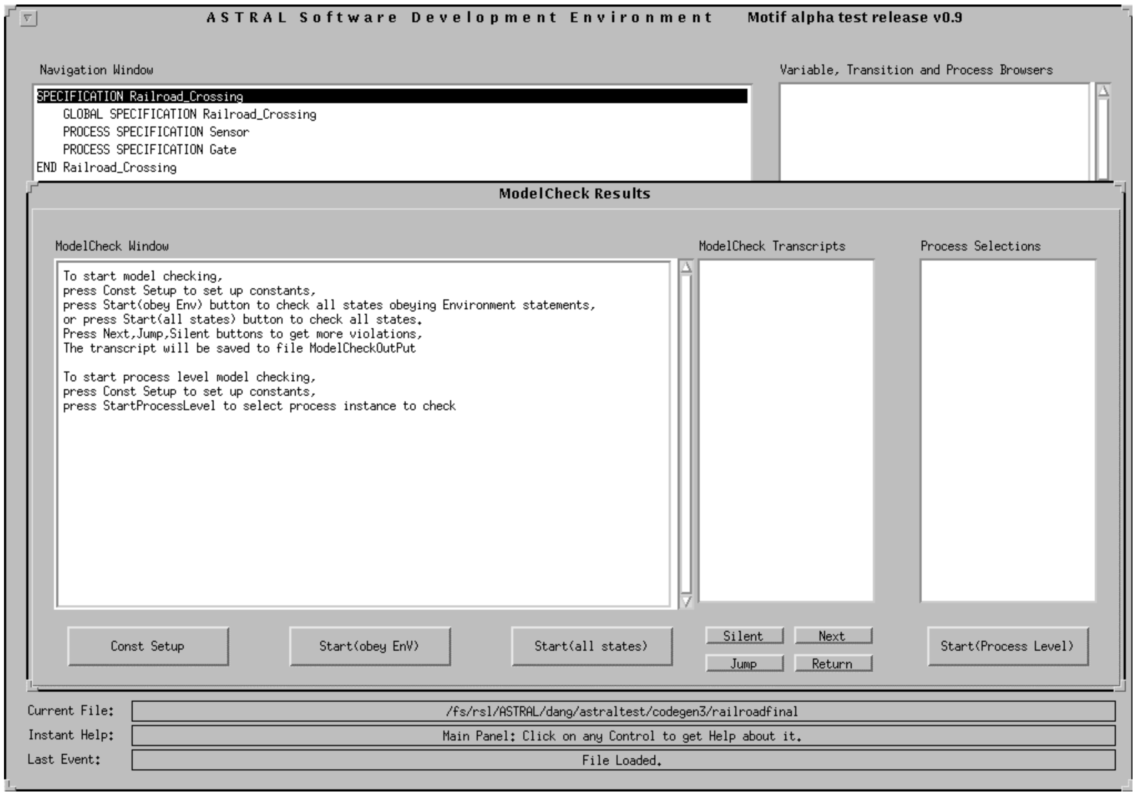
```
no_trans_fire: LEMMA
   (FORALL (t0, t3):
      t0 ≤ t3 AND
      (EXISTS (tr1):
         t0 ≥ Duration(tr1) AND
         Fired(tr1, t0 - Duration(tr1))) AND
      (FORALL (tr1, t1):
         t0 ≤ t1 AND t1 ≤ t3 AND
         (FORALL (t2):
            t0 ≤ t2 AND t2 ≤ t1 IMPLIES
               Vars_No_Change(t0, t2)) AND
         (FORALL (tr2, t2):
            t0 ≤ t2 AND t2 < t1 IMPLIES
               NOT Fired(tr2, t2)) IMPLIES
            NOT Fired(tr1, t1)) IMPLIES
         (FORALL (tr1, t1):
            t0 ≤ t1 AND t1 ≤ t3 IMPLIES
               NOT Fired(tr1, t1)))
```

This lemma can be used to prove the subgoal of the vars_no_change axiom requiring that no transition ends in the given interval. This lemma takes advantage of several facts about ASTRAL proofs. First, in order to show that no transition fires in an interval, it is sufficient to show that no transition is the *first* to fire in that interval. When showing that no transition is the first to fire at a time t in the given interval, it can be assumed that no transition fires between the beginning of the interval up until t. Another observation is that the vars_no_change axiom is almost always applied on an interval beginning with the end of some transition. Since it is assumed that no transition fires up until t and a transition just ended to start the interval, it can also be assumed that no variable changes from the beginning of the interval up until and including t. With this assumption, the circular application of axioms is eliminated when using the vars_no_change axiom, and the proofs are significantly simplified. In addition to the no_trans_fire lemma, 9 other lemmas were developed during the proofs of the railroad crossing.

3.7. Model Checker

Since performing proofs with a theorem prover can be a long and arduous task and finding an error within a specification during the theorem proving process can negate days or even weeks worth of work, it is to the user's advantage to be as confident as possible that the requirements of a specification hold before invoking the theorem prover. One way to gain this confidence is through the use of a model checker. The model checker can check the requirements of a specification over a finite time interval and for a given set of system constants. Figure 12 shows the model checker window that is brought up by clicking the "ModelCheck" button in the SDE as shown in figure 4.

The current ASTRAL model checker is an updated version of the prototype reported in [Dang and Kemmerer 1997]. In the prototype, the model checking process was realized by simulating the ASTRAL abstract machine and enumerating all possible execution branches (up to a time bound) in order to check the critical requirements of a specification. The new model checker, in contrast, generates customized C++ code for each specification. This code is actually a prototype implementation of the specified system and a control module to enumerate all the branches of execution of this implementation up to a system time bound set by the user. This code generator approach takes advantage of the fact that ASTRAL is a modularized specification language and that the mapping from an ASTRAL specification to C++ classes is quite natural. For example, an ASTRAL process instance can be translated into a C++ class instance, and ASTRAL data types, like LIST and SET, can be implemented efficiently in C++. Using the code generator makes it possible to check large specifications within a reasonable amount of time.



```
SPECIFICATION Railroad_Crossing
    GLOBAL SPECIFICATION Railroad_Crossing
    PROCESS SPECIFICATION Sensor
    PROCESS SPECIFICATION Gate
END Railroad_Crossing
```

```
To start model checking,
press Const Setup to set up constants,
press Start(obey Env) button to check all states obeying Environment statements,
or press Start(all states) button to check all states.
Press Next,Jump,Silent buttons to get more violations,
The transcript will be saved to file ModelCheckOutPut

To start process level model checking,
press Const Setup to set up constants,
press StartProcessLevel to select process instance to check
```

Current File: /fs/rsl/ASTRAL/dang/astraltest/codegen3/railroadfinal
Instant Help: Main Panel: Click on any Control to get Help about it.
Last Event: File Loaded.

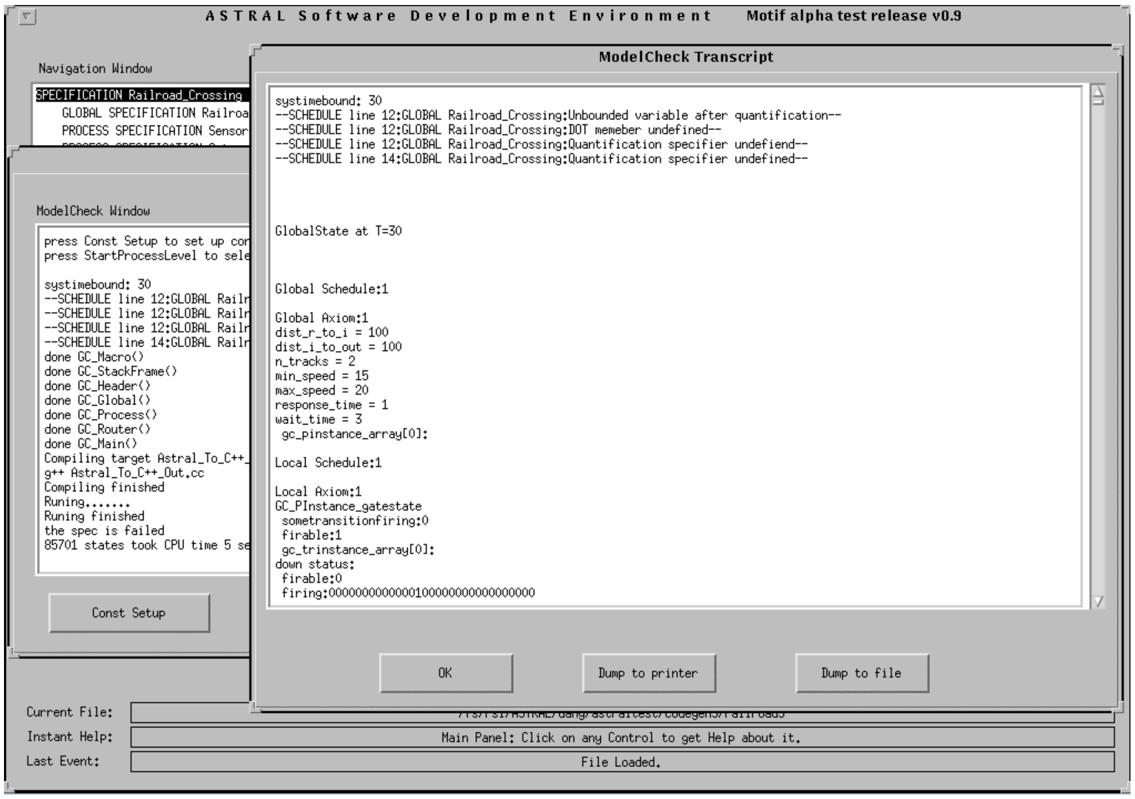
Figure 12. The model checker window.

The model checking procedure in the SDE 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 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 satisfy the environment clauses. 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, among others, the truth values of all the critical requirements, and the values of all local variables and the status information of all transition instances for every process instance. The user can easily follow the trace to figure out where the specification goes wrong, since each trace is for a single execution branch of the specification and is presented at the specification level. Figure 13 shows the transcript of a violating trace of states from an earlier version of the railroad specification. Besides performing system level model checking as described above, the model checker can also perform process level model checking by enumerating all reachable states of a process instance. In this case, the states explored are restricted by the imported variable clause of the process.

When using the ASTRAL model checker to analyze the railroad system, it was necessary to change the duration constants in the railroad specification from real to integer, because the model checker only deals with discrete time. The model checking results from the railroad specification demonstrated that the model checker is a useful tool for finding errors in a specification. It was able to successfully uncover a number of errors in *earlier* versions of the railroad specification, and when it was eventually run on the specification presented in this paper, which is the seventh version, no errors were found after searching 8,932,382 states in 200.7 seconds. The time bound used was 25, and the constants were set as follows:

```
dist_R_to_I = dist_I_to_out = 100
min_speed = 15
max_speed = 20
wait_time = 3.
```

All other constants were set to the value 1. Note that the constants need to be chosen carefully to assure meaningful results. In general, this means that the choice of constants must be sufficient to assure that something "interesting" occurs within the chosen time interval. For instance, for the railroad system, the interesting events are trains entering and exiting the crossing. If (dist_R_to_I + dist_I_to_out) / max_speed were chosen larger than the time bound, the requirements would hold, but essentially nothing would have been tested because no train could reach the crossing within the time interval.


A S T R A L S o f t w a r e D e v e l o p m e n t E n v i r o n m e n t   Motif alpha test release v0.9
Navigation Window
SPECIFICATION Railroad_Crossing
GLOBAL SPECIFICATION Railroa
PROCESS SPECIFICATION Sensor
ModelCheck Window
Const Setup
ModelCheck Transcript
OK
Dump to printer
Dump to file
Current File:
Instant Help:
Main Panel: Click on any Control to get Help about it.
Last Event:
File Loaded.

```
press Const Setup to set up cor
press StartProcessLevel to sele

systimebound: 30
--SCHEDULE line 12:GLOBAL Railr
--SCHEDULE line 12:GLOBAL Railr
--SCHEDULE line 12:GLOBAL Railr
--SCHEDULE line 14:GLOBAL Railr
done GC_Macro()
done GC_StackFrame()
done GC_Header()
done GC_Global()
done GC_Process()
done GC_Router()
done GC_Main()
Compiling target Astral_To_C++_
g++ Astral_To_C++_Out.cc
Compiling finished
Runing.......
Runing finished
the spec is failed
85701 states took CPU time 5 se
```

```
systimebound: 30
--SCHEDULE line 12:GLOBAL Railroad_Crossing:Unbounded variable after quantification--
--SCHEDULE line 12:GLOBAL Railroad_Crossing:DOT memeber undefined--
--SCHEDULE line 12:GLOBAL Railroad_Crossing:Quantification specifier undefiend--
--SCHEDULE line 14:GLOBAL Railroad_Crossing:Quantification specifier undefined--




GlobalState at T=30



Global Schedule:1

Global Axiom:1
dist_r_to_i = 100
dist_i_to_out = 100
n_tracks = 2
min_speed = 15
max_speed = 20
response_time = 1
wait_time = 3
 gc_pinstance_array[0]:

Local Schedule:1

Local Axiom:1
GC_PInstance_gatestate
 sometransitionfiring:0
 firable:1
 gc_trinstance_array[0]:
down status:
 firable:0
 firing:00000000000001000000000000000000
```

Figure 13. Transcript of a violating trace of states.

Once a suitable degree of assurance was obtained by running the model checker on the railroad specification with the constant values and time interval, as indicated above, the theorem proving component of the SDE was used to prove that selected portions of the requirements held for all constant values and for all times.

4. Related Work

A number of other development environments are available for other specification languages. MT [Clements *et al*. 1993] is an integrated development environment for the Modechart language, which incorporates components similar to those of the SDE. It includes the ability to hierarchically traverse specifications and invoke the editor on the displayed portion. MT's Consistency and Completeness Checker is similar in function to the SDE's validation procedure, performing a variety of static checks on the current specification. MT also provides a simulator which allows users to set up various conditions and display the results of a single execution path. Finally, MT has a verifier component to verify certain types of properties such as starvation and reachability.

STATEMATE [Harel *et al*. 1990] is another graphical environment for specifying reactive systems based on statecharts. It includes a number of different editors supporting statecharts, activity-charts, and module-charts which, like the SDE editor, check for syntactic errors immediately upon input. STATEMATE can also perform various consistency, completeness, and static logic tests at any time during a session. A simulator component can be run either interactively or according to a program specified in a simulation control language. Simulator overhead can be avoided by automatically generating a rapid prototype of the specified system in C or Ada code.

The Graphical Interval Logic Toolset [Kutty *et al*. 1993] provides support for editing and verifying formulas written in Graphical Interval Logic. Like the SDE, the editor in the toolset is syntax-directed to prevent syntactic errors. Formulas can be easily composed to create more complex formulas. Given a set of predicates from the user that supposedly implies a formula, the toolset uses a theorem prover to search for and display an appropriate counterexample, if one exists.

StateTime [Ostroff 1997] is a toolset supporting the design and analysis of specifications written in the Timed Transition Model/Real-Time Temporal Logic (TTM/RTTL) framework. StateTime consists of a graphical front-end to construct TTM specifications. Like the SDE, StateTime has both model checking and theorem proving capabilities. Additionally, a translator component can translate TTM/RTTL specifications to the language of the STeP prover [Bjorner *et al*. 1997].

5. Conclusions and Future Work

The SDE offers features that reduce errors and facilitate use throughout all stages of the specification development process. In the initial specification phase, the editor prevents syntax errors and the formatter enhances readability. In the middle phase, the validation function reports type errors, scoping errors, missing parameters, etc., and the VCG component generates the proof obligations needed to prove the specification correct with respect to its critical requirements. In the analysis phase, the model checker and theorem prover components allow these requirements to be proven. Finally, the browsers and compose/build features provide for easy maintenance and reuse of specifications.

ASTRAL has been used by both its developers and others to specify a number of interesting real-time concurrent systems. In [Coen-Porisini *et al*. 1994] the results of using it to formally specify a Consultative Committee on International Telephony and Telegraphy (CCITT) system that consists of a packet assembler process and several input processes is reported. In [Coen-Porisini *et al*. 1997] a phone system is composed with a switch to generate a wide-area phone system. The use of ASTRAL as a hardware description language was demonstrated in [Buonanno *et al*. 1992], where it was used to formally specify a checksum generator and a universal asynchronous receiver transmitter (UART) between a modem and a microprocessor. At Delft University of Technology (The Netherlands) ASTRAL was used to specify a robot control system [Brink *et al*. 1995]. The ASTRAL model checker has also been used to demonstrate flaws in several encryption protocols [Dang and Kemmerer 1997]. These case studies demonstrate the expressiveness and the power of the language. They also show ASTRAL's usefulness for specifying varying types of real-time systems from basic hardware to complete communication systems. The packet assembler, the wide-area phone system, the railroad crossing, an elevator controller, a production cell, an Olympic boxing scoring system, and a number of encryption protocols have all been specified and validated using the SDE.

This paper has shown how the ASTRAL SDE eliminates a large portion of the burden placed on the ASTRAL developer. It allows formal specifications to be written more quickly and with less errors and it provides the user with tools for analyzing the specifications. It has also shown two of the updates to the ASTRAL language that have been incorporated into the SDE, which makes these features readily available to the system designers using the environment. The model checker allows the user to quickly check for design errors in a specific time interval and the theorem prover provides assistance for discharging the general proof obligations.

The current version of the ASTRAL SDE is available for public use. The system can be obtained by anonymous ftp at ftp.cs.ucsb.edu in the directory /pub/rsg-distrib. The file name to download is in the form "astral-mmmyy.tar.gz," where mmm is the first three letters of the month and yy is the last two digits of the year of the most recent version of the system.

The model checking technique used by the model checker in this paper is explicit state exploration. It handles only a subset of ASTRAL, for the richness of complete ASTRAL makes validity checking of a general ASTRAL formula undecidable. Strictly speaking, the model checker is only a testing tool when the specification considered has infinite states, as was discussed in [Dang and Kemmerer 1997]. A symbolic model checker that could handle a larger subset of ASTRAL is highly desirable. This is one of the ongoing efforts in the Reliable Software Group at UCSB.

One tool to be added to the ASTRAL SDE is a symbolic executor. This would allow the specifier to build a rapid prototype and observe the behavior of the system under explicit scenarios without the cost of building the actual system first. Another SDE component, which has been partially implemented, is a specification management tool. This tool keeps track of tasks that still need to be performed on each specification before it can be considered complete. The proof manager discussed in section 3.6 is one portion of this tool. In the future, more functionality will be added, such as providing direction to the user as to which actions should be performed next and with which tools.

A number of issues still need to be addressed in the ASTRAL-PVS translator and encoding. To strengthen the semantic foundation of the ASTRAL axioms, the proofs of soundness and completeness should be performed. In order to use the theorem prover for inter-level proofs, 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 necessary to show that an implementation is consistent with the level above. Currently, the refinement mechanism described in [Coen-Porisini *et al*. 1995] is in a transitional phase, thus its translation was postponed until the new refinement mechanism is in place.

In general, more proofs need to be performed for different ASTRAL systems using their PVS translations. In studying the proofs performed for many systems, it can be determined if recurring patterns exist in the proofs. These patterns can then be incorporated into suitable PVS strategies. The patterns discovered 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. Work is also in progress to investigate whether the structure of an ASTRAL specification affects which lemmas and strategies are most useful during the proofs for that specification.

6. Acknowledgments

The authors would like to thank Richard Lee and Marco Mussini who worked on an earlier version of the ASTRAL Software Development Environment. This research was partially supported by the National Science Foundation under Grant No. CCR-9204249.

References

Bjorner, N.S., Z. Manna, H.B. Sipma, and T.E. Uribe (1997), "Deductive Verification of Real-time Systems Using SteP," *Proceedings of the Fourth AMAST Workshop on Real-Time Systems*, Springer-Verlag, Berlin, Germany, pp. 22-43.

Brink, K., L. Bun, J. van Katwijk, and W.J. Toetenel (1995), "Hybrid Specification of Control Systems," *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems*, IEEE Computer Society Press, Los Alamitos, CA, pp. 149-152.

Buonanno G., A. Coen-Porisini, and W. Fornaciari (1992), "Hardware Specification Using the Assertion Language ASTRAL," *Proceedings of the Advanced Research Workshop on Correct Hardware Design Methodologies*, North-Holland, Amsterdam, Netherlands, pp. 335-358.

Clements, P.C., C.L. Heitmeyer, B.G. Labaw, and A.T. Rose (1993), "MT: A Toolset for Specifying and Analyzing Real-Time Systems," *Proceedings of the Real-Time Systems Symposium,* IEEE Computer Society Press, Los Alamitos, CA, pp. 12-22.

Coen-Porisini, A., C. Ghezzi, and R.A. Kemmerer (1997), "Specification of Realtime Systems Using ASTRAL," *IEEE Transactions on Software Engineering*, 23, 9, pp. 572-598.

Coen-Porisini, A. and R.A. Kemmerer (1993), "The Composability of ASTRAL Realtime Specifications," *Proceedings of the International Symposium on Software Testing and Analysis*, ACM, New York, NY, pp. 128-138.

Coen-Porisini, A., R.A. Kemmerer, and D. Mandrioli (1994), "A Formal Framework for ASTRAL Intra-level Proof Obligations," *IEEE Transactions on Software Engineering*, 20, 8, pp. 548-561.

Coen-Porisini, A., R.A. Kemmerer, and D. Mandrioli (1995), "A Formal Framework for ASTRAL Inter-level Proof Obligations," *Proceedings of the Fifth European Software Engineering Conference*, Springer-Verlag, Berlin, Germany, pp. 90-108.

Crow, J., S. Owre, J. Rushby, N. Shankar, and M. Srivas (1995), "A Tutorial Introduction to PVS," *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques*, Available at http://www.csl.sri.com/wift_tutorial.html.

Dang, Z. and R.A. Kemmerer (1997), "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, New Brunswick, NJ.

Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot (1990), "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering*, 16, 4, pp. 403-414.

Heitmeyer, C. and N. Lynch (1994), "The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems," *Proceedings of the Real-Time Systems Symposium*, IEEE Computer Society Press, Los Alamitos, CA, pp. 120-131.

Kolano, P.Z. (1998), "A Theorem Prover for ASTRAL," Technical Report TRCS 98-01, Department of Computer Science, University of California, Santa Barbara, CA.

Kutty, G., Y.S. Ramakrishna, L.E. Moser, L.K. Dillon, and P.M. Melliar-Smith (1993), "A Graphical Interval Logic Toolset for Verifying Concurrent Systems," *Proceedings of the Conference on Computer-Aided Verification*, Springer-Verlag, Berlin, Germany, pp. 138-153.

Ostroff, J.S. (1997), "A Visual Toolset for the Design of Real-Time Discrete-Event Systems," *IEEE Transactions on Control Systems Technology*, 5, 3, pp. 320-337.

Appendix

```
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
                        &  Call2(enter_R, t))
            →       Call(enter_R) - Call2(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 cross 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 cross 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
```