A Theorem Prover for ASTRAL

Paul Z. Kolano

University of California, Santa Barbara

Abstract

The ASTRAL real-time formal specification language has been encoded into the PVS theorem prover. A translator has been developed to completely translate any single-level ASTRAL specification into its corresponding PVS encoding. The semantics of the ASTRAL abstract machine have been revised and expanded for use with PVS. This paper describes the encoding and semantics and explains their use along with providing other possible applications of the encoding.

Introduction

A real-time system is a system whose actions must be performed within certain time bounds in order to guarantee correct behavior. With the advent of cheap processing power and increasingly sophisticated consumer demands, real-time systems have become commonplace in everything from refrigerators to automobiles. Besides such numerous everyday uses, real-time systems are also being employed in more complex and potentially deadly applications such as weapons systems and nuclear reactor control where deviation from critical timing requirements can result in disastrous loss of lives and/or property. It is thus desirable to extensively test and verify the designs of these systems to gain assurance that such disasters will not occur. A number of formal methods for real-time systems have been proposed [HM 96] that provide a framework under which developers can eliminate ambiguity, reason rigorously about system design, and prove that critical requirements are met using well-defined mathematical techniques. Real-time systems are characterized by concurrency, asynchrony, nondeterminism, and dependence upon the external operating environment. Thus, the formal proofs of even simple real-time systems can be nontrivial. To make the verification of real-world real-time systems practical, mechanical proof assistance is necessary.

One such form of assistance is an interactive theorem prover. Interactive theorem provers provide mechanical support for deductive reasoning. Each theorem prover is associated with a specification language in which a system and associated theorems are expressed. A theorem prover uses a collection of axioms and inference rules about its specification language to reduce a high-level proof into simpler subproofs that can eventually be discharged by basic built-in decision procedures that support arithmetic and boolean reasoning. Theorem provers provide a number of forms of assistance, 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. A number of general purpose theorem provers were considered, including PVS [COR 95], ACL2 [KM 96], and HOL [GM 93]. Among these, PVS was considered ideal for ASTRAL given its powerful typing system, higher-order facilities, heavily automated decision procedures, and intuitive reasoning style. Comparisons between PVS and other theorem provers can be found in [CM 95], [Gor 95], and [You 96].

This paper discusses the adaptation of the PVS theorem prover for performing analysis of real-time systems written in the formal specification language ASTRAL [CGK 97]. ASTRAL formulas may contain arbitrary first-order logic expressions involving ASTRAL-specific operators and variables of complex types. A translator has been written that can completely translate any single-level ASTRAL specification into its corresponding PVS encoding. The semantics of ASTRAL have been revised and expanded for use with PVS. Some small example systems have been proved using the encoding. Additionally, the encoding has been used as the basis for other tools, such as a transition sequence generator. The translator and PVS add theorem proving capabilities to the ASTRAL Software Development Environment [KK 97], which additionally consists of a syntax-directed editor, a specification processor, a specification testing component, and a browser kit.

PVS

The Prototype Verification System (PVS) [OSR 93c] is a powerful interactive theorem prover based on typed higher-order logic. A PVS specification [OSR 93b] consists of a modular collection of *theories*. A theory may be parameterized to support polymorphism. Declarations in one theory can be referenced in another theory by using an *importing clause*. Parameterized theories can be imported either with explicit parameters or without parameters. If left without parameters, PVS attempts to instantiate the theory based on the use of its declarations within the importing theory. Most single parameter theories can be instantiated automatically by PVS, but theories with complex or multiple parameters often need to be instantiated explicitly in the referring theory.

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

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

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

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

PVS generates the following TCC for the definition of i_to_e2:

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

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

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

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

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

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

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

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

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

When the PVS prover [OSR 93a] 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 $(A_1 \,\&\, ... \,\&\, A_n) \rightarrow (C_1 \mid ... \mid C_n)$. Thus, a sequent is true if (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 $A_i = C_j$. A sequent may be split into two or more subgoals by splitting the conjuncts of the sequent using the *split* command or by introducing cases explicitly with the *case* command. PVS maintains a proof tree, consisting of all of the subgoals generated during a proof. Initially, when the prover is invoked on a theorem, the proof tree consists of only the sequent form of that theorem. As the proof proceeds, subgoals may be generated and proved. To prove

that a particular goal in the proof tree holds, all its subgoals must be proved to hold. PVS allows the user to define *strategies*, which are collections of prover commands that can be developed to automate frequently occurring proof patterns.

ASTRAL Semantics

Before ASTRAL could be encoded into PVS, it was necessary to determine its precise formal semantics. Since the semantics of ASTRAL had been addressed previously in [CKM 94] and [CSK 94], the first thought was to simply translate these definitions. Unfortunately, both sets of semantics were determined to be inadequate for translation into PVS. In [CKM 94], an axiom system is introduced for the ASTRAL abstract machine. While investigating its definition for use with PVS, however, the axiom system given was determined to be neither sound nor complete. For the invariant proofs, three axioms are given stating (1) that the end of a transition occurs at its given duration after the last start, (2) that when a process is idle and some transition is enabled, then some transition fires, and (3) that transitions are nonoverlapping on a single process instance. For the schedule proofs, four axioms are given, which include the three axioms of the invariant proofs, with the definition of "enabled" in (2) modified to include requiring a call to have been issued, along with a fourth axiom describing what it means for a call to have been issued. Consider the following process:

```
Process P
        Export
                T
        Constant
                dur: posreal
        Transition T   [TIME: dur]
                Entry
                        TRUE
                Exit
                        TRUE
End P
```

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

$$\text{Start}(T, \text{now} - \text{dur}) \rightarrow \text{Start}(T, \text{now})$$

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

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

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

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

In addition to the difficulty just described, it was also somewhat undesirable to have to encode all of the axioms and inference rules for the ASTRAL base logic, since most of them are essentially just those of first-order logic. PVS already has first-order logic axioms and inference rules defined internally so it was preferable to take advantage of the already available PVS framework. Unfortunately, unlike some other theorem provers, PVS does not support partial functions, so the undefined element used in the three-valued logic definition of ASTRAL cannot be defined directly in the underlying PVS logic without introducing a new three-valued domain. As mentioned earlier, however, PVS does have a very powerful predicate subtyping system that allows functions to be declared with domains of only those elements satisfying a given predicate, such as only those elements for which a function is well defined.

Another consideration in the encoding of ASTRAL into PVS was encoding the ASTRAL operators that take formulas as operands (i.e. those operators that do not immediately evaluate their operands). The *past* operator is a prime example. The past operator, past(A, t), takes an expression A and a time t and returns the value that A had at t. Unlike a "traditional" function, such as addition, in which the operands can be evaluated immediately, the evaluation of the expression operand of the past operator must be delayed until its *parent context* is determined. In a past expression, the parent context refers to the value of "now" in the expression. For example, let now_0 be the current time in the system. The parent context of the expression "past(past(x, now-1), now-1)" is now_0. The parent context of the operand "past(x, now-1)", however, is now_0 - 1, and similarly for "x", the parent context is now_0 - 2. Thus, "past(past(x, now-1), now-1)" should evaluate to the value of x at time now_0 - 2. [CSK 94] defines the axioms associated with the past operator, so that the correct value can be derived.

Previous PVS Efforts

Two different approaches have been used to handle unevaluated expression operands in PVS. In encoding the TRIO language [GMM 90] into PVS, a similar problem was encountered. The TRIO *dist* operator takes a TRIO formula and a time and returns the value that the formula had at the time given. dist is very similar to past in that the expression given as the first operand cannot be evaluated until its context is fully defined in the parent expression. In [AGM 97], an uninterpreted "TRIO_formula" type is introduced to encode dist. The dist operator is defined as a function of type [[TRIO_formula, time] → TRIO_formula]. Thus, dist(dist(A, t1), t2) is a well-defined function producing a TRIO_formula. Then, axioms similar to those defined for the past operator in [CSK 94] are defined to transform elements of type TRIO_formula to other elements of type TRIO_formula. Eventually, there must be some valuation from TRIO_formulas to "real-world" values (i.e. booleans, integers, etc.). Thus, a function "now" (not related to the ASTRAL now) is defined that takes a TRIO_formula and produces the corresponding boolean value assuming an initial context of the current time instant.

The TRIO-PVS approach could also have been utilized for the ASTRAL-PVS encoding. It was felt, however, that there were several disadvantages to this approach in the case of ASTRAL. In order to evaluate expressions involving operators such as past, axioms of the form:

```
eval_past_plus: AXIOM
        eval(Past(f1 + f2, t)) = eval(Past(f1, t)) + eval(Past(f2, t))
```

would need to be added for every operator in the language. Unlike the TRIO dist operator, which always returns a boolean, the past operator can return any type defined in a specification. Thus, although the use of parameterized theories would help somewhat, the writing of these axioms, not to mention their eventual use, would be a very onerous task. Besides all of these "eval" axioms, all axioms and inference rules of first-order logic would also have to be encoded, since PVS can't explicitly manipulate uninterpreted expressions. It is undesirable to introduce so many axioms as it becomes more and more confusing to the

user as to which axiom is appropriate for which situation. In the ASTRAL encoding, the desire was to keep the axiom set as small and manageable as possible and to rely on the existing PVS framework.

Another disadvantage of the TRIO encoding is that many of the typing facilities of PVS cannot be utilized. Since the declaration of an "ASTRAL_formula" type would need to be uninterpreted, PVS would not be able to decide anything about them. This means that instead of utilizing the PVS subtyping system to restrict operator domains appropriately, the three-valued logic of ASTRAL would have to be explicitly encoded into PVS, resulting in more axioms. It is also more difficult to use function definitions with uninterpreted types, because PVS cannot know when something should be evaluated or not because it does not know anything about its type.

In TRIO, the current time is always implicit, hence the TRIO-PVS encoding was designed to keep the current time implicit. In ASTRAL, however, the current time can be referenced explicitly by using the variable now. Additionally, the past operator references an absolute time and not a time with respect to the current time. That is, in TRIO, dist(A, d) is the value of the expression A at d time from the current time into the past or future depending on whether d is negative or positive. In ASTRAL, however, past(A, t) refers to the value of the expression A at time t in the system. Note that there is no reference to the current time in this definition. Thus, in fact, the current context of an ASTRAL expression must in some cases be known explicitly.

The Duration Calculus (DC) [ZHR 91] is another real-time language that has been encoded into PVS. Like the ASTRAL past operator and the TRIO dist operator, the DC "dur" operator can only be evaluated when the interval of its parent context has been established. In the DC-PVS encoding [SS 94], DC operators are defined as curried PVS functions, which when given their "original" operands, return a function from an "Interval" to the original range of the operator. For example, the disjunction operator "\/" is defined as "\/(A, B)(i): bool = A(i) OR B(i)", where A and B are of the type [Interval → bool] and i is of type Interval. Using this technique, the resulting functions can be combined normally, while still delaying the evaluation of the whole expression until the context is given. This is possible because the functions return the same type as their operands. For example, "A \/ B" has the same type as A and B individually, thus they can be combined arbitrarily. Eventually, when an interval is given, an actual boolean value is obtained.

Like TRIO, DC is an implicit time temporal logic. In the encoding, however, an explicit interval is eventually needed to evaluate a DC expression and return a simply typed value that can be used by the decision procedures of PVS. To keep the reasoning similar to that of unencoded DC proofs, a special front-end to PVS was developed in [SS 94] to hide the details of the encoding from the user. The front-end essentially strips out all of the "interval indices" from a PVS formula to get the corresponding DC

formula. Such a mechanism does not make sense for ASTRAL since the context of each expression must be known explicitly even before encoding, so there's no need to hide it.

ASTRAL also differs from TRIO and DC in that it is a state-machine based language and not a temporal logic. Much of the effort in the DC encoding was given to supporting the special inference rules for the modal operators of DC. ASTRAL also requires inference rules for reasoning in its base logic, but these are, for the most part, the inference rules of first-order logic and are not specifically tailored for the ASTRAL language. The proofs of ASTRAL systems rely not so much on these standard definitions, but instead are heavily dependent on the axiomatization of the ASTRAL abstract machine. Since PVS already supports first-order logic within its prover, much of the effort of axiomatization could be directed toward the definition of the abstract machine.

An encoding of ASTRAL into PVS was reported in [Bun 96] and [Bun 97], but this encoding is based on a definition of ASTRAL that has been developed independently at Delft University based on earlier ASTRAL work in [GK 91a] and [GK 91b]. This definition has diverged from the work reported in [CGK 97] and [CKM 94] and has essentially become a different language, although with similar syntax. The encoding from [Bun 96] and [Bun 97] is similar to the work presented here only in that a variable is declared as a function from time to its ASTRAL domain. It does not include any of the semantics revisions, abstract machine axioms, operator definitions, arbitrary type translations, etc. that are discussed in this paper.

Other real-time state machine languages have been encoded into theorem provers. The Timed Automata Model has been encoded into PVS [AH 96] and Timed Transition Systems into HOL [HCH 93]. These languages are based on interleaved concurrency, however, which makes their semantics simpler than those of ASTRAL. Additionally, they are not defined in terms of arbitrary first-order logic expressions and do not have the complex subtyping mechanisms that are available in ASTRAL.

ASTRAL Encoding

Each ASTRAL process specification is defined as a separate PVS theory. The transitions of a process are defined by a set of five declarations. The "transition" type is an enumerated type that consists of all transition names as well as an identifier "trans__i" for each ith exception of all transitions "trans". Throughout the discussion, examples from the translated elevator specification will be used. A portion of the translation is included in appendices C, D, and E. The ASTRAL specification of the elevator can be found in appendix B. The transition declaration for the Elevator_Button_Panel process would be:

```
transition: TYPE = {request_floor, clear_floor_request}
```

If request_floor had an exception condition, there would be an additional element called "request_floor__1". It was sometimes necessary in the definitions of the ASTRAL transition operators (i.e. start, end, and call) to quantify over all the elements of the transition type that are associated with a

particular *base transition*, where the base transitions are all the non-exception elements of the transition type. Thus, the function "Base_Trans(TR1: transition)" is defined to be "trans" for TR1 = trans__i and TR1 otherwise. Each process has a function "Duration" that defines the duration of each transition in the transition type. For each base transition, the function "Exported" returns true or false if that transition is exported or not, respectively. Similarly, "Has_Parms" returns true or false if the given base transition has parameters or not.

The handling of parameterized transitions was one of the more difficult issues that arose during the encoding. One of the main goals of the ASTRAL translation was to be able to define a special PVS library that contained the axiomatization as well as the definitions of all of the ASTRAL operators, which could be proven (i.e. TCCs and lemmas proven) independently and be included with any distribution. It was desired to provide as much of the language in the library as possible, defined and rigorously reasoned about beforehand, so the translator would not be relied on too heavily and/or the user burdened later on. The translator could then be focused on the specification-dependent items. For the most part, this was achieved in a straightforward manner, but parameterized transitions presented a number of obstacles. The main difficulty arose in the definition of the transition operators. In ASTRAL, for a single transition T with n parameters (p_1, ..., p_n), all of the following are legal expressions: Start(T), Start(T(p_1)), Start(T(p_1, p_2)), ..., Start(T(p_1, ..., p_n)). This, combined with the fact that transition parameters may be of arbitrary number and type, seemed to indicate that Start could not be predefined, but must be constructed separately for each use within a given specification.

A solution was found, however, that while not particularly elegant, avoids this very undesirable complication. For each process specification, a "parameter" type is introduced that is a record containing the parameter names and types of *all* transitions in the process. In the Elevator_Button_Panel process, the definition of parameter is:

```
parameter: TYPE = [# request_floor__f: floor #]
```

To avoid name conflicts between transitions, it was necessary to rename parameters. "request_floor__f" refers to the f parameter of the request_floor transition. The idea of this scheme is that all entry/exit assertions and transition operator definitions can reference the same type (i.e. parameter) and use only those parts of a parameter instance appropriate in the given situation. The parts of a parameter that are not used in an expression for all intents and purposes do not exist for that expression. For example, an entry assertion may reference parameters that are passed to it when called from the external environment. The entry assertion only references its own declarations within the parameter type, thus only constrains those portions of the parameter. The unreferenced elements of the parameter type can have any value, thus they do not affect the reasoning. The parameter definition allows the transition operators to be defined uniformly as part of the ASTRAL-PVS library. A single definition of the Start operator handles arbitrary numbers and types of transition parameters.

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

Besides the transition declarations, a process theory also consists of declarations of types, constants, variables, and definitions. Since PVS has the ability to declare predicate subtypes as described earlier, the translation of types was very straightforward. For example, the definition of "floor", which is a typedef is:

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

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

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

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

For the most part, these translations were straightforward. One difference between ASTRAL and PVS, however, is that in ASTRAL, there is no ordering implied in the declarations of the specification. That is, it is not necessary (in fact, not possible by the structure of ASTRAL specifications) to declare constants, variables, etc. before they are used. Thus, declarations in the type section may refer to declarations in the constant and definition sections and possibly vice-versa, without producing a typecheck error. In PVS, however, an item can only be referenced after it has been declared. Thus, in order to translate the declarations in an ASTRAL specification correctly into a corresponding PVS specification, the ordering of

declarations needed to be explicitly determined. To handle this, the translator first constructs the dependencies of all type declarations. As the dependency lists are created, any constants, definitions, and process instances encountered are added to that type's dependency list and their dependencies constructed. If a time-dependent expression, such as a variable or a past expression, is encountered during the construction of the dependencies, an error is reported, since a type cannot be time-dependent. Eventually, when all dependencies lists have been created, the necessary ordering can be determined by declaring the items without dependencies, removing those dependencies from the remaining items, and repeating until all items have been declared. Any circular dependencies encountered result in an error.

All well-formed formula clauses of ASTRAL, such as invariants, entry assertions, definitions, etc. are translated identically. All of the operators of the ASTRAL language have been encoded as interpreted functions, so given their operands, they evaluate to specific values. The parse tree of the ASTRAL formula is traversed and the appropriate PVS definition is substituted for each ASTRAL operator. Additionally, all ASTRAL operator function definitions are curried functions from their normal operand domains to the type [time → range]. For example, "Start1(tr1, at1)" takes a transition tr1 and an operand at1 of type [time → time] and returns a function of type [time → bool] such that when the evaluation time t1 is given will return true if and only if the last start of tr1 at time t1 was at time at1(t1). In the Start1 definition, as well as the definitions of all ASTRAL operators that take a time operand, the time operand is itself of type [time → time] and is only evaluated after an evaluation context is provided. Since it is not known whether at1(t1) will be a valid operand or not (i.e. will cause the expression to be undefined), t1 is limited by the PVS typing system to be greater than or equal to at1(t1). It is then the user's job to show via a TCC obligation that any evaluation times of a Start1 expression occurring in a specification are permissible. The Start1 operator is defined:

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

The operator is called Start1 instead of Start to be consistent with the definition of End, which could not be defined as "End" since it is a keyword of PVS. The "1" indicates that the operator refers to the last start. Later, the Startn definition will be shown which refers to the nth start in the past. The definition of Start1 is complicated by the need to handle the definition of transition exceptions. In ASTRAL, it is not possible to assert anything about the start of an exception. That is, assertions can only be made about the start time of the base transition, so stating Start(trans, t1) actually means that trans fired because of its entry assertion *or any one of its exceptions*. It is convenient to think of exceptions as separate transitions

in the encoding, however, because just like a transition with a single entry assertion, they each have a precondition, a postcondition, and a duration, and must be mutually exclusive within the same process instance. The definition of Start1 takes this into account. It says that for at1(t1) to be the last time btr1 started, its entry or one of its exceptions must have fired and no entry or exception has fired from that time up until the evaluation time t1. Start1 is only well-defined in a context in which the time of the last start at1(t1) is greater than or equal to 0 and less than or equal to the evaluation time t1.

A less intuitive definition is that of the Startn operator (i.e. the nth start in the past).

```
Startn_0(ai1: [time → posint], btr1: base_transition, at1: [time → time])(t1: time):
                RECURSIVE bool =
        (IF at1(t1) > t1 THEN FALSE
        ELSIF ai1(t1) = 1 THEN Start1(btr1, at1)(t1)
        ELSE (EXISTS (t2):
                        t2 <= t1 AND
                        Start1(btr1, const(t2))(t1)) AND
                (EXISTS (t2):
                        t2 < Start1(btr1)(t1) AND
                        Start1(btr1, const(t2))(t2)) AND
                Startn_0(const(ai1(t1) - 1), btr1, const(at1(t1)))
                        (epsilon! (t2):
                                t2 < Start1(btr1)(t1) AND
                                Start1(btr1, const(t2))(t2) AND
                                (FORALL (t3):
                                        t2 < t3 AND t3 < Start1(btr1)(t1) IMPLIES
                                                NOT Start1(btr1, const(t3))(t3)))
        ENDIF)
MEASURE (LAMBDA (ai1, btr1, at1): (LAMBDA (t1): ai1(t1)))

Startn(ai1, btr1, at1)(t1: {t1 | at1(t1) <= t1}): bool =
        Startn_0(ai1, btr1, at1)(t1)
```

The definition of Startn is split into two functions, Startn and its internally invoked counterpart, Startn_0. This separation is so that a TCC is only generated for the Startn reference occurring in the original ASTRAL specification and not for the subsequent recursive references in the internal definition. That is, the user only has to prove that the time argument in the original specification is between 0 and the evaluation time so that the result is well-defined. In the subsequent recursive calls, however, it may be that the evaluation time given by the epsilon expression may actually be before the time the user gave (at1) and yet still be a well defined result. In that case, it would mean that the nth start in the past did not occur at the given time since that time was "passed up" by the recursive calls before the nth start occurred and hence the expression is false and not undefined. For a similar reason, the existence clauses have been added to the main body of Startn_0 rather than restricting the domain of the evaluation time t1. If there have not yet been the number of starts specified in the integer argument, then the expression is false and not undefined. The existence clauses also serve to trivially satisfy the TCCs generated for the Start1(btr1)(t1) used in the definition.

This definition also illustrates the use of the PVS epsilon function, which is used in many of the transition operator and Change operator definitions. epsilon is a "choice function", which given a predicate p of type [T → bool], where T is an arbitrary type, returns a element e of type T such that p(e) holds. In the Startn definition, it is used to select the second time in the past that the given transition has started. In other words, Startn(n, trans, time) is true if there has been a start and additionally Startn(n-1, trans, t1) holds, where t1 is the second time in the past that trans has started. Note that t1 must be a time before the last start or else the same time would be returned every iteration. The *measure* at the end of the Startn_0 definition must be given in every PVS recursive function definition. It has the same signature as the associated function and defines an expression that decreases in each recursive iteration. It is used to prove the termination of the function. The definitions of Endn, Calln, and Changen are defined similarly to Startn.

With the operators defined in this manner, it is then possible to combine ASTRAL operators in standard ways to achieve an expression that will only be evaluated once its temporal context is given and must be proven to be well-defined. Defining the operators this way also has the advantage that all expressions translated from ASTRAL to PVS can be easily expanded and reduced via the built-in mechanisms of PVS such as beta reduction. Additionally, such an encoding forces every operator to be examined very closely for its exact meaning. This encoding, however, does have some drawbacks. One is that by using curried functions and variables from time to their domains, the reasoning is no longer in ASTRAL. Instead of reasoning about past expressions as is done in the original semantics of [CSK 94] and [CKM 94], the user must now work directly with the underlying PVS logic. A more pressing concern is that it becomes more difficult to translate the results gleaned from PVS back into the ASTRAL context. This difficulty arises because many of the conjuncts of the operator definitions will be split apart and "flattened" during the proof process. Thus, although all the components of an operator definition may be present, it is non-trivial to reform them correctly into an expression equivalent to that of the operator. This becomes an issue when an error is found during the proofs and the original ASTRAL specification needs to be changed. The counterexample found may no longer be recognizable back in the ASTRAL context. This is a valid concern, but upon closer examination, the transition between the PVS translation and the original ASTRAL specification is not as great as it may seem. The logic of PVS is very similar to that of ASTRAL. Additionally, any result found will be in terms of variables and Fired expressions. As discussed previously, v(t) in the encoding is equivalent to past(v, t) in ASTRAL. There is also a very fine line between Fired(tr1, t1) in the encoding and Start(tr1, t1) in ASTRAL, thus the difference in reasoning between proving or disproving that tr1 Fired at t1 or Started at t1 is very small. It may still be difficult to see where the counterexample fits in the main proof and how the original specification needs to be changed, but this can hopefully be gleaned from an examination of the complete PVS proof tree.

Another concern is that restricting time operand domains rather than directly encoding an undefined value and the associated axioms uses the *strict semantics* of quantification over time rather than the *loose semantics*, as defined in [CSK 94]. These semantics define the values of quantified expressions in the presence of undefined values. In the strict semantics, "FORALL t: time (A)" is true if A evaluates to true for all values of t, false if A evaluates to false for at least one value of t and A is not undefined for any value of t, and undefined otherwise. In the loose semantics, "FORALL t: time (A)" is false if A does not evaluate to true for any value of t and A evaluates to true for at least one value of t, false if A evaluates to false for at least one value of t, and undefined otherwise. For example, "FORALL t: time (End(tr, t + duration(tr)) → Start(tr, t))" is a true statement in the loose semantics, but in the translation, a context must be provided to evaluate the expression and since the domain of time ranges over values past any context that can be given, the formula must be written instead as "FORALL t: time (t <= now → (End(tr, t + duration(tr)) → Start(tr, t)))".

The major obstacle in translating arbitrary well-formed formulas was translating identifiers with types involving lists and structures. In ASTRAL, it is possible to define arbitrary combinations of structures and lists as types, thus references to variables of these types can become quite complex. For example, consider the following type declaration:

```
list1:  list of integer,
struct1:  structure of (l_one: list1),
list2:  list of struct1,
struct2:  structure of (l_two(integer): list2)
```

If s2 is a variable of type struct2, valid uses of s2 would include s2 by itself, s2[l_two(5)], s2[l_two(5)][9], s2[l_two(5)][9][l_one], and s2[l_two(5)[9][l_one][2]. The translation of expressions such as these must result in a curried time function, so that it can be used with the definitions of the curried boolean and arithmetic operators. The expression in each bracket can be time-dependent, so it is necessary to define the translation such that an evaluation context (i.e. time) given to the expression as a whole is propagated to all expressions in brackets.

In the translation, s2 is a function of type [time → struct2] and struct2 is a record [# l2: [integer → list2] #]. The translation of "s2[l_two(5)][9]", for example, is:

```
(λ(T1: time): nth(((λ(T1: time): l_two((s2)(T1))((const(5))(T1))))(T1), (const(9))(T1)))
```

The lambdas are added to propagate the temporal context in which the formula is to be evaluated. Although the lambda expression generated for s2 looks very difficult to decipher, translated expressions will never actually be used in "raw" form. In the proof obligations and elsewhere, a translated expression is never used "as is", but is first evaluated in some context. Once this evaluation occurs, all the lambdas drop out and the formula is simplified to a combination of variables and predicates. For example, the expression above evaluated at time t becomes:

```
nth(l_two((s2)(t))(5), 9)
```

First, the value of the variable s2 is evaluated at time t. Then the record member l_two is obtained from the resulting record. This member is parameterized, so is given a parameter of 5. Finally, element 9 of the resulting list is obtained.

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

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

even though the actual exit assertion is:

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

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

Currently, the ASTRAL-PVS translator can completely translate any single-level ASTRAL specification. One issue that has not been resolved satisfactorily, however, is the translation of set cardinality expressions. The SET_SIZE operator in ASTRAL returns the cardinality of sets of any type. PVS also provides this functionality in its cardinality library. To use the cardinality operator, however, the cardinality theory must be instantiated with a suitable mapping from the type of the set to the natural numbers. Given the ability in ASTRAL to define arbitrary predicate subtypes, automatically generating such a mapping from an ASTRAL specification is a non-trivial matter. At present, a blank IMPORTING template is added in the generated PVS specifications that must be filled in by the user if the SET_SIZE operator is used in the ASTRAL specification.

In addition to a theory for each ASTRAL process type specification, there is an additional PVS theory generated for the global specification. The declarations of the global theory are constructed similarly to those of the process theories. The global theory also defines the process instances in the system as well as all exported variables and transitions that are used in the system. Each process definition may contain references to imported variables. The imported variables must be declared before they are referenced, so it's not possible to import a variable from an instantiated process theory because that process may in turn import a variable or transition from the importing process, resulting in a circular dependency between

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

ASTRAL Axioms

The execution history of a process is represented by the predicates *Fired* and *Called*, which each take a transition and a time and return whether or not the transition has fired or been called at that time, respectively. Additionally, the functions *Fire_Parms* and *Call_Parms* are defined to record the history of transition parameters. These functions are defined:

```
Fired:  [[transition, time] → bool]
Called:  [[{btr1: base_transition | Exported(btr1)}, time] → bool]

Fire_Parms:  [[btr1: {btr1: base_transition | Has_Parms(btr1) AND NOT Exported(btr1)},
              {t1: time | (EXISTS (tr1: transition): Base_Trans(tr1) = btr1 AND
                 Fired(tr1, t1))}] → parameter]
Call_Parms:  [[etr1: {etr1: exported_transition | Has_Parms(etr1)},
              {t1 | Called(etr1, t1)}] → parameter]
```

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

There are seven core axioms based on these four functions that describe the ASTRAL abstract machine. *trans_fire* is the only way to derive that a transition fired. It states that if some transition is enabled and the process is idle (i.e. no other transition is in the middle of execution), then some transition will fire. Note that Enabled requires that the transition's entry assertion holds and that if the transition is exported, then it has been called. The exact definition of Enabled is in appendix A.

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

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

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

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

```
trans_exit: AXIOM
        (FORALL (tr1, t1):
                t1 >= Duration(tr1) AND
                Fired(tr1, t1 - Duration(tr1)) IMPLIES
                        Exit(tr1, t1))
```

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

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

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

```
trans_mutex: AXIOM
        (FORALL (tr1, t1):
                Fired(tr1, t1) IMPLIES
                        (FORALL (tr2):
                                tr2 /= tr1 IMPLIES
                                        NOT Fired(tr2, t1)) AND
                        (FORALL (tr2, t2):
                                t1 < t2 AND t2 < t1 + Duration(tr1) IMPLIES
                                        NOT Fired(tr2, t2)))
```

These five axioms describe the dynamic execution of transitions. Besides the start, end, and call times of transitions, the other time-dependent entities are variables. The axioms so far only describe variables implicitly in the Entry, Exit, and Enabled functions used in them. Thus, the value of a variable is only known at the time a transition starts and when it ends. In ASTRAL, however, it is also known that a variable only changes value when a transition ends. Thus, the *var_changes* axiom states this fact. Specifically, it states that for any interval in which a transition has not ended, all variables keep a single

value throughout the interval. The Vars_No_Change function is process-dependent and is constructed by the translator based on the variables declared in each process. It states that the value of all variables of the process have the same value at t1 and t3. This axiom was missing from [CKM 94].

```
var_changes: 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
                        Vars_No_Change(t1, t3))
```

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

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

In addition to the core axioms, there are three axioms dealing with imported transitions. For the most part, nothing can be inferred about imported variables and transitions. It is not known when imported variables will change, nor what the duration of an imported transition is, nor what held when an imported transition started or ended, and so on. If any of these items are required to hold to prove a schedule, they must be explicitly stated in an imported variable clause. There are, however, a few things that can be deduced about all imported transitions, regardless of context. None of these axioms were dealt with in [CKM 94] or [CSK 94]. The imported axioms are expressed in terms of *Started*, *Ended*, and *Called*. The exact duration between a start and an end of an imported transition is not known globally or in other processes because the duration is implementation dependent. The duration may have different values depending on the number and durations of exceptions to each transition. Thus, Started and Ended had to be defined separately, rather than the single Fired of local process definitions. *i_trans_mutex* states that for any process id and in any interval such that an imported transition started at the beginning of the interval and has not yet ended, no imported transition can have started or ended on the process associated with that process id within the interval (excluding the first instant).

```
i_trans_mutex: AXIOM
        (FORALL (id1, itr1, t1, t3):
                t1 < t3 AND
                Started(id1, itr1, t1) AND
                (FORALL (t2):
                        t1 < t2 AND t2 <= t3 IMPLIES
                                NOT Ended(id1, itr1, t2)) IMPLIES
```

```
                (FORALL (itr2, t2):
                        t1 < t2 AND t2 <= t3 IMPLIES
                                NOT Started(id1, itr2, t2) AND
                                NOT Ended(id1, itr2, t2)))
```

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

```
i_trans_end: AXIOM
        (FORALL (id1, itr1, t3):
                Ended(id1, itr1, t3) IMPLIES
                        (FORALL (itr2):
                                itr2 /= itr1 IMPLIES
                                        NOT Ended(id1, itr2, t3)) AND
                        (EXISTS (t1):
                                t1 < t3 AND
                                Started(id1, itr1, t1) AND
                                (FORALL (t2):
                                        t1 < t2 AND t2 < t3 IMPLIES
                                                NOT Ended(id1, itr1, t2))))
```

i_trans_start is similar to i_trans_end, except that it states that if an imported transition starts, then no other imported transition started at the same time and that the transition was called since the last time it started.

```
i_trans_start: AXIOM
        (FORALL (id1, itr1, t3):
                Started(id1, itr1, t3) IMPLIES
                        (FORALL (itr2):
                                itr2 /= itr1 IMPLIES
                                        NOT Started(id1, itr2, t3)) AND
                        (EXISTS (t1):
                                t1 <= t3 AND
                                Called(id1, itr1, t1) AND
                                (FORALL (t2):
                                        t1 <= t2 AND t2 < t3 IMPLIES
                                                NOT Started(id1, itr1, t2))))
```

There are some axioms that are specification-dependent and must be constructed during translation. In the local and global cases, the axiom section from the specification is translated as an axiom. The axiom section is a time-independent clause, but rather than implementing a separate translation procedure for it, the standard formula translation is used and then the formula is evaluated at time 0. If the axiom clause is written correctly and is time-independent, the 0 will drop out and only assertions about constants will remain. There are two additional axioms constructed in the global specification. The type "id" is declared as a NONEMPTY_TYPE, thus nothing is known about it by PVS, except that id has at least one item in its domain. The axiom *id_domain* further refines this domain by stating that every id must correspond to some process instance declared in the processes section. The second axiom *id_unique* states that every process instance corresponds to a unique id. Thus, the type id is declared to be exactly the set of process instances. There is also an additional axiom in each process definition. The axiom

self_imports states the relationship between exported, but locally named variables and their global counterparts of the form i_var(id). In addition to stating that var = i_var(self), self_imports also relates the local values of Fired, Called, and Call_Parms to the global definitions of Started, Ended, Called, and i_Call_Parms for exported transitions. The definitions of these axioms for the elevator system can be found in appendices C and D.

Proof Obligations

The local invariant and schedule proof obligations are a straightforward translation of those presented in [CKM 94]. The obligations for the elevator process are located in appendix E. The global obligations look very similar to the local obligations but translating them correctly, however, is not as easy. In the global obligations, it is not possible to reason about properties in quite the same way as in the local case, because the global proofs cannot use any information that is not exported by each process. That is, at the global level, nothing is known about implementational details such as transition entry and exit assertions or the values of local variables. Thus, it's not possible to use the style of reasoning employed in local proofs, such as when a transition must fire, what held before and after it fired, and so on. Instead, the global proofs must be performed by using the local invariants and schedules as lemmas to prove properties of the system as a whole. Local proof obligations contain references to local variables that are not visible at the global level. It is necessary, however, to be able to use the proofs performed at the local level as lemmas in the proof of the global properties. It is a non-trivial problem to determine which portions of the local formulas can be removed "safely" (i.e. still preserve the property). Currently, a separate lemma with the desired properties must be introduced by the user with a proof that should follow immediately from the local proof obligations and the self_imports axiom.

Example Proof Session

Initially, this section was going to contain a small proof session based on the elevator specification in order to demonstrate how many of the axioms are used. Unfortunately, the attempt of a few small examples quickly turned into hundreds of pages of PVS output. Since it is not practical to include the bulk of the proofs, only a few of the major subgoals will be given along with some explanation of which axioms were used to finish the proofs. The example that was proved was the following local invariant of the elevator process:

End(arrive, now) & the_elevator_buttons.floor_requested(position) →
    Start(open_door, now)

This states that if the elevator arrives at a floor and there is a request from inside the elevator to stop at that floor, then the elevator will stop and open the doors. This example illustrates the basic approach to proving that a transition fires at a given time. After expanding the ASTRAL definitions and doing some skolemization and simplification, the sequent to be proved is:

[-1] (t2!1 >= arrive_dur)

```
[-2]   Fired(arrive, (t2!1 - arrive_dur))
[-3]   (FORALL (t2):
       t2!1 < t2 AND t2 <= t2!1
          IMPLIES
        (FORALL (tr1):
          tr1 = arrive AND t2 >= Duration(tr1)
             IMPLIES NOT Fired(tr1, t2 - Duration(tr1))))
[-4]   i_floor_requested(the_elevator_buttons)((position)(t2!1))(t2!1)
  |-------
{1}    Fired(open_door, t2!1)
```

In this sequent, since none of the antecedents are contradictory, it needs to be shown that the single consequent holds. That is, that open_door fires at time t2!1. As mentioned, the only way to prove that a transition fires is to use the axiom trans_fire. After introducing this axiom as a lemma and instantiating it with open_door and t2!1, three main subgoals result. In the first subgoal:

```
{-1}   (EXISTS (tr1): Fired(tr1, t2!1))
[-2]   (t2!1 >= arrive_dur)
[-3]   Fired(arrive, (t2!1 - arrive_dur))
[-4]   (FORALL (t2):
       t2!1 < t2 AND t2 <= t2!1
          IMPLIES
        (FORALL (tr1):
          tr1 = arrive AND t2 >= Duration(tr1)
             IMPLIES NOT Fired(tr1, t2 - Duration(tr1))))
[-5]   i_floor_requested(the_elevator_buttons)((position)(t2!1))(t2!1)
  |-------
[1]    Fired(open_door, t2!1)
```

the antecedent "(EXISTS (tr1): Fired(tr1, t2!1))" has been added. Thus, it is known that some transition fired at t2!1, but has not yet been shown that the transition was open_door. To complete the proof of this subgoal, it must be shown that open_door is the only transition that is enabled at t2!1. The new antecedent is existentially quantified, so can be skolemized to remove the quantifier. Thus, antecedent [-1] becomes "Fired(tr1!1, t2!1)". From here, the proof is broken up into a separate case for each possible value of tr1!1 (i.e. each transition of the elevator). The case for open_door is trivially performed since "Fired(open_door, t2!1)" will be both an antecedent and a consequent. open_door is shown to be enabled at t2!1 in the second main subgoal. For the remaining transitions, trans_entry is used to introduce the entry assertion of each transition as an antecedent and then a contradiction obtained. In some cases, these proofs become rather involved because in order to show that the entry assertion is false, the predecessors of the transition must be determined. For example, it is not possible to achieve a contradiction for close_door until it is shown that move_up or move_down must have fired before arrive so that the door is already closed and cannot be closed again. In these cases, the implied nochange expressions come into play and the var_changes axiom must be used to show that certain variables keep the same value from when the predecessor's entry assertion held.

In the second subgoal:

```
[-1]   (t2!1 >= arrive_dur)
```

```
[-2]   Fired(arrive, (t2!1 - arrive_dur))
[-3]   (FORALL (t2):
       t2!1 < t2 AND t2 <= t2!1
          IMPLIES
        (FORALL (tr1):
          tr1 = arrive AND t2 >= Duration(tr1)
             IMPLIES NOT Fired(tr1, t2 - Duration(tr1))))
[-4]   i_floor_requested(the_elevator_buttons)((position)(t2!1))(t2!1)
  |-------
{1}    (EXISTS (tr1): Enabled(tr1, t2!1))
[2]    Fired(open_door, t2!1)
```

the consequent "(EXISTS (tr1): Enabled(tr1, t2!1))" has been added. Thus, it must be shown that some transition was enabled at t2!1. This proof is carried out similarly to that of the first subgoal except that only a single case needs to be proven. Since the existential quantifier appears as a consequent, the user may instantiate it with a transition of choice. Once the quantifier has been instantiated, Enabled is expanded and it is attempted to match actual process conditions in the antecedents with the enabling conditions in the consequents. Note that it is only necessary to show that some transition is enabled at t2!1 and not specifically open_door. In this particular proof, however, if any transition besides open_door is enabled, then it will not be possible to achieve a contradiction for that transition in the proof of the first subgoal. trans_exit may be used to introduce conditions that hold in the process at t2!1. In this proof, trans_exit was instantiated with the arrive transition to show that position changes and that moving stays the same, since it must be shown that the last change of position must be greater than the last change of moving in the entry assertion of open_door. Like the first subgoal, this subgoal may become lengthy because some conditions necessary for the chosen transition to fire may not be explicitly stated in the sequent, so must be derived by determining its possible predecessors. In this proof, open_door requires ~door_moving and ~door_open to hold, which are not asserted in the entry or exit of arrive, so thus it must be shown that the only possible predecessors to arrive are move_up and move_down, which both assert those conditions in their entry assertions.

In the third subgoal:

```
[-1]   (t2!1 >= arrive_dur)
[-2]   Fired(arrive, (t2!1 - arrive_dur))
[-3]   (FORALL (t2):
       t2!1 < t2 AND t2 <= t2!1
          IMPLIES
        (FORALL (tr1):
          tr1 = arrive AND t2 >= Duration(tr1)
             IMPLIES NOT Fired(tr1, t2 - Duration(tr1))))
[-4]   i_floor_requested(the_elevator_buttons)((position)(t2!1))(t2!1)
  |-------
{1}    (FORALL (tr2, t2):
       t2!1 - Duration(tr2) < t2 AND t2 < t2!1 IMPLIES NOT Fired(tr2, t2))
[2]    Fired(open_door, t2!1)
```

a consequent has been added that requires the process to be idle at t2!1. In particular, it must be shown that no transition is in the middle of firing at t2!1. This subgoal is proved mainly by using the trans_mutex axiom. The universal quantifier is first skolemized to remove the quantifier and consequent [1] becomes an expression in terms of skolem variables tr2!1 and t2!2. The proof is then broken into two cases: t2!2 >= t2!1 - arrive_dur and t2!2 < t2!1 - arrive_dur, which correspond to tr2!1 firing before arrive fires and after, respectively. In the former case, trans_mutex is instantiated with arrive to show that tr2!1 cannot fire after arrive fires and before t2!1. In the latter case, trans_mutex is instantiated with tr2!1 to show that arrive couldn't have fired at t2!1 - arrive if tr2!1 fired at a time before t2!1 - arrive_dur. In both cases, a contradiction is achieved and the subgoal is proved.

Other Applications

Now that ASTRAL has been encoded into a theorem prover and a translator has been developed, the encoding can be used as a basis for other useful analysis tools. One such tool is a transition sequence generator. That is, a tool that can construct sequences of transitions that are possible in a process and that satisfy given properties (e.g. a certain length, between two specific transitions, etc.). This can be used to estimate time delays between states, help the user visualize the operation of the system, and in some cases can be used to prove simple system properties.

A first step towards such a tool is determining the possible successors of each transition. Once this information has been obtained, the sequences with the desired properties can be constructed. Determining whether one transition is the successor of another, however, is undecidable since transition entry/exit assertions may be arbitrary first-order logic expressions. Many successors, however, can be eliminated based only on the simpler portions of the entry/exit assertions, such as boolean and enumerated variables. The translator produces a file of proof obligations to automatically eliminate as many transition successors as possible. For each pair of transitions (tr1, tr2), an obligation tr1_not_tr2 is added as shown below.

```
tr1_not_tr2: THEOREM
        NOT (FORALL (t1): (FORALL (t2):
                t1 + Duration(tr1) <= t2 AND
                Fired(tr1, t1) AND Fired(tr2, t2) AND
                (FORALL (tr3, t3):
                        t1 + Duration(tr1) < t3 + Duration(tr3) AND
                        t3 + Duration(tr3) <= t2 IMPLIES
                                NOT Fired(tr3, t3))))
```

Note that the theorem is written in a somewhat peculiar negated form to keep the third quantifier in the antecedent portion of the theorem's sequent representation, which simplifies the definition of the strategy to discharge these obligations discussed below. Also note that this theorem only states that some transition must end between tr1 and tr2 and does not exclude tr1 or tr2 from firing. The above obligation is sufficient, however, to prove that a transition besides tr1 and tr2 must fire in between any firing of tr1 and tr2. If only tr1 and tr2 fire in between t1 and t2, then since t2 - t1 is finite and the durations of tr1

and tr2 are constant and non-null, eventually a contradiction can be achieved by applying the above theorem repeatedly on an ever shortening interval. Such a proof is non-trivial to perform in PVS, thus a lemma is available in the ASTRAL-PVS library stating that if tr1_not_tr2 holds, then a transition other than tr1 and tr2 must fire between any times that tr1 and tr2 fire.

A PVS strategy has been defined to automatically discharge these obligations. This strategy uses abstract machine axioms to introduce the entry and exit assertions of tr1, the entry assertion of tr2, and the fact that if nothing ended between the end of tr1 and the start of tr2, then all variable values remained constant during this time. Once all of this information is present, the strategy invokes a modified version of the PVS "grind" strategy that expands all ASTRAL definitions except those of quantifiers, the change operator, and the transition operators.

In addition to the theory with the successor proof obligations in it, the translator also constructs a PVS proof script file consisting of a proof attempt for each obligation that uses the predefined strategy. Thus, the proofs of all the obligations can be attempted without human interaction by a single PVS command (M-x prove-theory) and the results of all the proofs are displayed when PVS has finished. The file of results can then be read by another tool to construct transition sequences based on the displayed results. Using PVS as the basis for a sequence generation tool has a number of benefits. Not only are the results obtained at minimal cost, but they are also of use as lemmas during the full system proofs. Even though the results may not be complete since some successors may not be eliminated without human interaction, the results are known to be sound because they are carried out within the framework of the encoding.

The table below shows the results of running the proof script generated for the elevator system. For each process type, the table shows the maximum number of successors, the number of successors that are actually possible, and the number that were computed automatically with the proof script generated for that process. The difference in the elevator process results from two factors. First, many of the transition entry assertions contain timed operators to define the delay between certain operations. Since the strategy does not expand these operators to control complexity, this information cannot be used to eliminate successors. Additionally, many entry assertions do not constrain all of the state variables, thus it is sometimes necessary to look at sequences of length three or more to be able to eliminate certain possibilities.

process type	max # of succ	actual # of succ	computed # of succ
Elevator	36	11	22
Elevator_Buttons	4	4	4
Floor_Buttons	16	12	12

Conclusions and Future Work

An ASTRAL to PVS translator has been developed that can completely translate any single-level ASTRAL specification. The translator includes a library of ASTRAL definitions and axioms that has been completely proved and rigorously reasoned about. The library also contains strategies to assist in rewriting ASTRAL definitions during proof attempts. A few small examples have been completed using the encoding.

A number of issues still need to be addressed in future work. To strengthen the semantic foundation of the ASTRAL axioms, the proofs of soundness and correctness should be performed. For the translator and encoding, 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 interlevel proof obligations necessary to show that an implementation is consistent with that of the level above. Currently, the refinement mechanism described in [CKM 95] is in a transitional phase, so its translation was postponed until the new refinement mechanism is in place. Additionally in the translator, the cardinality translation needs to be studied further based on more experience in doing proofs to see if any general guidelines can be given as to instantiating the cardinality theory. A better solution to using the results from local proofs in the proofs of global properties is also desirable.

In general, more proofs need to be performed for different ASTRAL systems using their PVS translations. In studying the proofs performed of many systems, it can be determined if reoccurring patterns exist in the proofs. These patterns can then be incorporated into suitable PVS strategies. One such strategy will be to attempt TCCs encountered during proofs. From the proofs completed so far, it was noticed that many of the TCCs generated that were not automatically completed by the prover, could have been completed if a suitable rewriting scheme was available to the basic PVS TCC strategy. Thus, a new TCC strategy incorporating the expand-astral strategy will be developed. The patterns may also lead to the definition of useful lemmas that can be proven in advance and added to the ASTRAL-PVS library for future use. It is also useful to investigate whether the structure of the ASTRAL specification determines which lemmas and strategies are most useful.

One item that will be integrated shortly into the SDE is the ability to perform “on-the-fly” translations. That is, the user can write a formula in ASTRAL and the SDE will give the PVS translation of the formula, which can be cut-and-pasted into the PVS prover window, back to the user. This will be helpful in introducing lemmas to the prover. With this facility available, the user can formulate helpful lemmas in the ASTRAL language, while still being able to take advantage of the PVS prover.

Bibliography

[AGM 97] Alborghetti, A.; Gargantini, A.; Morzenti, A. "Providing automated support to deductive analysis of time critical systems". *Proceedings of the 6th European Software Engineering Conference*, Zurich, Switzerland, 22-25 Sept. 1997.

[AH 96] Archer, M.; Heitmeyer, C. "Mechanical verification of timed automata: a case study". *Proceedings Real-Time Technology and Applications*, Brookline, MA, USA, 10-12 June 1996. Edited by: Jeffay, K.; Zhao, W. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1996. p. 192-203.

[AK 86] Auernheimer, B.; Kemmerer, R.A. "Procedural and nonprocedural semantics of the ASLAN formal specification language". *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, 1986.

[Bun 96] Bun, L. "Checking properties of ASTRAL specifications with PVS". *Proceedings of the 2nd Annual Conference of the Advanced School for Computing and Imaging*, Lommel, Belgium, June 1996, p. 102-7.

[Bun 97] Bun, L. "Embedding Astral in PVS". *Proceedings of the 3rd Annual Conference of the Advanced School for Computing and Imaging*, Heijen, The Netherlands, June 1997, p. 130-6.

[CGK 97] Coen-Porisini, A.; Ghezzi, C.; Kemmerer, R.A. "Specification of realtime systems using ASTRAL". *IEEE Transactions on Software Engineering*, Sept. 1997, vol. 23, (no. 9): 572-98.

[CKM 94] Coen-Porisini, A.; Kemmerer, R.A.; Mandrioli, D. "A formal framework for ASTRAL intralevel proof obligations". *IEEE Transactions on Software Engineering*, Aug. 1994, vol. 20, (no. 8): 548-61.

[CKM 95] Coen-Porisini, A.; Kemmerer, R.A.; Mandrioli, D. "A formal framework for ASTRAL inter-level proof obligations". *Proceedings of the 5th European Software Engineering Conference*, Sitges, Spain, 25-28 Sept. 1995. Berlin, Germany: Springer-Verlag, 1995. p. 90-108.

[CM 95] Carreño, V.A.; Miner, P.S. "Specification of the IEEE-854 floating-point standard in HOL and PVS". *8th International Workshop on Higher-Order Logic Theorem Proving and Its Applications*, Aspen Grove, UT, Sept. 1995.

[COR 95] Crow, J.; Owre, S.; Rushby, J.; Shankar, N.; Srivas, M. "A Tutorial Introduction to PVS". *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, Apr. 1995.

[CSK 94] Coen-Porisini, A.; San Pietro, P.; Kemmerer, R.A. "Formal semantics definition for ASTRAL". Unpublished technical report, Department of Computer Science, University of California, Santa Barbara, 1994.

[GK 91a] Ghezzi, C.; Kemmerer, R.A. "ASTRAL: an assertion language for specifying realtime systems". *Proceedings of the 3rd European Software Engineering Conference*, Milan, Italy, 21-24 Oct. 1991. Edited by: van Lamsweerde, A.; Fugetta, A. Berlin, Germany: Springer-Verlag, 1991. p. 122-40.

[GK 91b] Ghezzi, C.; Kemmerer, R.A. "Executing formal specifications: the ASTRAL to TRIO translation approach". *Proceedings of the Symposium on Testing, Analysis, and Verification*, Victoria, B.C., Canada, Oct. 1991.

[GM 93] Gordon, M.J.C.; Melham, T.F.: editors. Introduction to HOL: a theorem proving environment for higher order logic New York: Cambridge University Press, 1993.

[GMM 90] Ghezzi, C.; Mandrioli, D.; Morzenti, A. "TRIO: a logic language for executable specifications of real-time systems". *Journal of Systems and Software*, May 1990, vol. 12, (no. 2): 107-23.

[Gor 95] Gordon, M. "Notes on PVS from a HOL perspective". Available at http://www.cl.cam.ac.uk/users/mjcg/PVS.html, Aug. 1995.

[HCH 93] Hale, R.; Cardell-Oliver, R.; Herbert, J. "An embedding of timed transition systems in HOL". *Formal Methods in System Design*, Aug. 1993, vol.3, (no.1-2):151-74.

[HM 96] Heitmeyer, C.; Mandrioli, D.: editors. Formal methods for real-time computing. New York: John Wiley, 1996.

[KK 97] Kemmerer, R.A.; Kolano, P.Z. "Formally specifying and verifying real-time systems". *Proceedings of the 1st IEEE International Conference on Formal Engineering Methods*, Hiroshima, Japan, 12-14 Nov. 1997.

[KM 96] Kaufmann, M.; Strother Moore, J. "ACL2: an industrial strength version of Nqthm". *Proceedings of the 11th Annual Conference on Computer Assurance*, Gaithersburg, MD, USA, 17-21 June 1996. New York, NY, USA: IEEE, 1996. p. 23-34.

[OSR 93a] Owre, S.; Shankar, N.; Rushby, J.M. The PVS Proof Checker: A Reference Manual. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[OSR 93b] Owre, S.; Shankar, N.; Rushby, J.M. The PVS Specification Language. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[OSR 93c] Owre, S.; Shankar, N.; Rushby, J.M. User Guide for the PVS Specification and Verification System. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[SS 94] Skakkebaek, J.U.; Shankar, N. "Towards a duration calculus proof assistant in PVS". *Formal Techniques in Real-Time and Fault-Tolerant Systems. 3rd International Symposium Proceedings*, Lubeck, Germany, 19-23 Sept. 1994. Edited by: Langmaack, H.; de Roever, W.-P.; Vytopil, J. Berlin, Germany: Springer-Verlag, 1994. p. 660-79.

[You 96] Young, W.D. "Comparing verification systems: interactive consistency in ACL2". *Proceedings of 11th Annual Conference on Computer Assurance*, Gaithersburg, MD, USA, 17-21 June 1996. New York, NY, USA: IEEE, 1996. p. 35-45.

[ZHR 91] Zhou Chaochen; Hoare, C.A.R.; Ravn, A.P. "A calculus of durations". *Information Processing Letters*, 13 Dec. 1991, vol.40, (no.5): 269-76.

Appendix A: PVS Definitions of Entry, Exit, and Enabled

```
astral_trans_ext[transition: NONEMPTY_TYPE, parameter: NONEMPTY_TYPE,
          Duration: [transition -> posreal],
          Base_Trans: [transition -> transition],
          Has_Parms: [{tr1: transition | Base_Trans(tr1) = tr1} -> bool],
          Exported: [{tr1: transition | Base_Trans(tr1) = tr1} -> bool],
          (IMPORTING astral_basic) time: TYPE,
          Entry_No_Parms: [{tr1: transition | NOT Has_Parms(
             Base_Trans(tr1))} -> [astral_basic.time -> bool]],
          Exit_No_Parms: [{tr1: transition | NOT Has_Parms(
             Base_Trans(tr1))} -> [astral_basic.time -> bool]],
          Entry_Parms: [[{tr1: transition | Has_Parms(Base_Trans(tr1))},
             parameter] -> [astral_basic.time -> bool]],
          Exit_Parms: [[{tr1: transition | Has_Parms(Base_Trans(tr1))},
             parameter] -> [astral_basic.time -> bool]]
]: THEORY

BEGIN

   ASSUMING
      base_of_base: ASSUMPTION
         (FORALL (tr1: transition):
            Base_Trans(Base_Trans(tr1)) = Base_Trans(tr1))
   ENDASSUMING

   IMPORTING astral_trans[transition, parameter, Duration, Base_Trans,
                     Has_Parms, Exported]

   tr1: VAR transition
   t1: VAR astral_basic.time

   % definition of Entry
   Entry(tr1, t1): bool =
      IF Has_Parms(Base_Trans(tr1)) THEN
         IF Exported(Base_Trans(tr1)) THEN
            Issued_Call(Base_Trans(tr1), t1) IMPLIES
               Entry_Parms(tr1, Call_Parms(Base_Trans(tr1),
                  Call1(Base_Trans(tr1))(t1)))(t1)
         ELSE
            Fired(tr1, t1) IMPLIES
               Entry_Parms(tr1, Fire_Parms(Base_Trans(tr1), t1))(t1)
         ENDIF
      ELSE Entry_No_Parms(tr1)(t1)
      ENDIF

   % definition of Exit
   Exit(tr1: transition, t1: {t1 | t1 >= Duration(tr1)}): bool =
      IF Has_Parms(Base_Trans(tr1)) THEN
         IF Exported(Base_Trans(tr1)) THEN
            Issued_Call(Base_Trans(tr1), t1 - Duration(tr1)) IMPLIES
               Exit_Parms(tr1, Call_Parms(Base_Trans(tr1),
                  Call1(Base_Trans(tr1))(t1 - Duration(tr1))))(t1)
```

```
        ELSE
          Fired(tr1, t1 - Duration(tr1)) IMPLIES
            Exit_Parms(tr1, Fire_Parms(Base_Trans(tr1),
              t1 - Duration(tr1)))(t1)
        ENDIF
      ELSE Exit_No_Parms(tr1)(t1)
      ENDIF

  % definition of Enabled
  Enabled(tr1, t1): bool =
    (Exported(Base_Trans(tr1)) IMPLIES
      Issued_Call(Base_Trans(tr1), t1)) AND
    IF Has_Parms(Base_Trans(tr1)) THEN
      IF Exported(Base_Trans(tr1)) THEN
        Entry_Parms(tr1, Call_Parms(Base_Trans(tr1),
          Call1(Base_Trans(tr1))(t1)))(t1)
      ELSE
        (EXISTS (p1: parameter):
          Entry_Parms(tr1, p1)(t1))
      ENDIF
    ELSE Entry_No_Parms(tr1)(t1)
    ENDIF

  % definition of Enabled_Set
  Enabled_Set(t1): set[transition] =
    {tr1 | Enabled(tr1, t1)}

END astral_trans_ext
```

Appendix B: ASTRAL Elevator System Specification

```
SPECIFICATION          Elevator_System
   GLOBAL SPECIFICATION   Elevator_System
      PROCESSES
            the_elevator: Elevator,
            the_elevator_buttons: Elevator_Button_Panel,
            the_floor_buttons: array [1..n_floors] of Floor_Button_Panel
      TYPE
            pos_integer: TYPEDEF i: integer (i > 0),
            floor: TYPEDEF i: pos_integer (i ≤ n_floors)
      CONSTANT
            n_floors: pos_integer,
            move_dur, arrive_dur, open_dur, close_dur, door_stop_dur: time,
            request_dur, clear_dur: time,
            t_service_request, t_move, t_stop, t_move_door: time
      AXIOM
            /* t_service_request must be big enough to handle the worst case. One instance of the
            worst case is when the elevator is moving up from floor 1 to 2 and 2 has not been requested
            on the elevator panel nor has any request been made on 2's button panel. Let t_arrive be the
            next time such that End(arrive, t_arrive). up_request and down_request are simultaneously
            called on floor 2 an "instant" after t_arrive-2*request_dur and down_request fires first. In
            addition, every floor in the building (besides 2) has up_requested (except the top floor) and
            down_requested (except the bottom floor). Thus, the up request is not posted in time for
            the elevator to service it and the elevator must stop and open the door at every floor up to
            the top, back down to the bottom, and back up to 2. The maximum time possible to spend
            on any floor when a request is outstanding elsewhere in the building is spent once on floors
            1, 2, and n_floors, and twice on every other floor, hence 2*n_floors-3. Additionally,
            2*request_dur + move_dur + t_move + arrive_dur elapses before the elevator initially
            reaches floor 3 and open_dur + t_move_door + door_stop_dur elapses on floor 2 before the
            door is fully opened when 2 is eventually reached again with the elevator traveling upward.
            */
            (t_service_request ≥
                  2 * request_dur + move_dur + t_move + arrive_dur +
                  (2 * n_floors - 3) * (     open_dur + t_move_door + door_stop_dur +
                                             t_stop + close_dur + t_move_door +
                                             door_stop_dur + request_dur + move_dur +
                                             t_move + arrive_dur) +
                  open_dur + t_move_door + door_stop_dur)
            /* clear_request must be able to fire no matter how many requests are made while the
            elevator door is opening */
      &     (clear_dur + n_floors * request_dur < t_move_door)
            /* must be at least 2 floors in the building */
      &     (n_floors ≥ 2)

      ENVIRONMENT
            /* assume that down arrow on first floor and up arrow on top floor have been physically
            covered and short circuited so that they are not available to the environment */
               ~the_floor_buttons[1].Call(request_down, now)
      &     ~the_floor_buttons[n_floors].Call(request_up, now)
            /* multiple button pushes should have no effect */
      &     (FORALL f: floor
                  (     Change(the_elevator_buttons.floor_requested(f), now)
```

```
                & ~the_elevator_buttons.floor_requested(f)
        →       FORALL t: time
                        (   t ≥ the_elevator_buttons.Start(request_floor(f))
                        →   ~the_elevator_buttons.Call(request_floor(f), t))))
    &   (FORALL f: floor
                (   Change(the_floor_buttons[f].up_requested, now)
                &   ~the_floor_buttons[f].up_requested
        →       FORALL t: time
                        (   t ≥ the_floor_buttons[f].Start(request_up)
                        →   ~the_floor_buttons[f].Call(request_up, t))))
    &   (FORALL f: floor
                (   Change(the_floor_buttons[f].down_requested, now)
                &   ~the_floor_buttons[f].down_requested
        →       FORALL t: time
                        (   t ≥ the_floor_buttons[f].Start(request_down)
                        →   ~the_floor_buttons[f].Call(request_down, t))))
        /* requests cannot be made of the elevator to stop at a floor from when the door starts
        opening on that floor until the door is fully closed */
    &   (   Change(the_elevator.door_open, now)
        &   ~the_elevator.door_open
→       FORALL t: time
                        (   t ≥ Change2(the_elevator.door_moving)
                    →
        ~the_elevator_buttons.Call(request_floor(the_elevator.position), t)
                        &   (   past(the_elevator.going_up, t)
                            &
        ~the_floor_buttons[the_elevator.position].Call(request_up, t)
                            |   past(~the_elevator.going_up, t)
                            &
        ~the_floor_buttons[the_elevator.position].Call(request_down, t))))
SCHEDULE
        /* any request must be serviced within time t_service_request */
        FORALL f: floor
            (   (the_elevator_buttons.Call(request_floor(f), now - t_service_request)
            →   EXISTS t: time
                        (   t > now - t_service_request
                        &   past(the_elevator.position, t) = f
                        &   Change(the_elevator.door_open, t)
                        &   past(the_elevator.door_open, t)))
            &   (the_floor_buttons[f].Call(request_up, now - t_service_request)
            →   EXISTS t: time
                        (   t > now - t_service_request
                        &   past(the_elevator.position, t) = f
                        &   Change(the_elevator.door_open, t)
                        &   past(the_elevator.door_open, t)
                        &   past(the_elevator.going_up, t)))
            &   (the_floor_buttons[f].Call(request_down, now - t_service_request)
            →   EXISTS t: time
                        (   t > now - t_service_request
                        &   past(the_elevator.position, t) = f
                        &   Change(the_elevator.door_open, t)
                        &   past(the_elevator.door_open, t)
                        &   past(~the_elevator.going_up, t))))
```

```
END Elevator_System

PROCESS SPECIFICATION          Elevator
  LEVEL   Top_Level
    IMPORT
        floor, move_dur, arrive_dur, open_dur, close_dur, request_dur, door_stop_dur,
        t_stop, t_move, t_move_door, the_elevator_buttons, the_floor_buttons,
        the_elevator_buttons.floor_requested, the_floor_buttons.up_requested,
        the_floor_buttons.down_requested
    EXPORT
        position, going_up, door_open, moving, door_moving
    VARIABLE
        position: floor,
        going_up, door_open, moving, door_moving: boolean
    DEFINE
        request_above(f0: floor): boolean ==
            EXISTS f: floor
              (     f > f0
              &     (     the_elevator_buttons.floor_requested(f)
                    |     the_floor_buttons[f].up_requested
                    |     the_floor_buttons[f].down_requested)),
        request_below(f0: floor): boolean ==
            EXISTS f: floor
              (     f < f0
              &     (     the_elevator_buttons.floor_requested(f)
                    |     the_floor_buttons[f].up_requested
                    |     the_floor_buttons[f].down_requested))
    INITIAL
        position = 1
      & going_up
      & ~door_open
      & ~moving
      & ~door_moving
    INVARIANT
        /* the elevator door must stay closed while the elevator is moving */
        (moving → ~door_open & ~door_moving)
        /* if the elevator is moving in some direction, then there must be an outstanding request
        in that direction */
      & (moving & going_up → request_above(position))
      & (moving & ~going_up → request_below(position))
    CONSTRAINT
        /* if the elevator changes direction, there cannot be an outstanding request in the old
        direction */
        (going_up & ~going_up′ → ~request_below′(position′))
      & (~going_up & going_up′ → ~request_above′(position′))

    TRANSITION move_up
      ENTRY             [TIME: move_dur]
                ~door_open
        &       ~door_moving
        &       request_above(position)
        &       (       going_up
                |       ~going_up
```

```
                &       ~request_below(position)
                &       ~the_floor_buttons[position].up_requested)
        &       (       End(arrive, now)
                &       ~the_elevator_buttons.floor_requested(position)
                &       ~the_floor_buttons[position].up_requested
                |       FORALL t, t1: time
                                (       Change(moving, t) & Change(door_open, t1)
                        →               t < t1 & now ≥ t1 + request_dur))
    EXIT
                moving
        &       going_up
TRANSITION      move_down
    ENTRY               [TIME: move_dur]
                ~door_open
        &       ~door_moving
        &       request_below(position)
        &       (       ~going_up
                |       going_up
                &       ~request_above(position)
                &       ~the_floor_buttons[position].down_requested)
        &       (       End(arrive, now)
                &       ~the_elevator_buttons.floor_requested(position)
                &       ~the_floor_buttons[position].down_requested
                |       FORALL t, t1: time
                                (       Change(moving, t) & Change(door_open, t1)
                        →               t < t1 & now ≥ t1 + request_dur))
    EXIT
                moving
        &       ~going_up
TRANSITION      arrive
    ENTRY               [TIME: arrive_dur]
                moving
        &       now - t_move ≥ Change(moving)
    EXIT
                going_up′ & position = position′ + 1
        |       ~going_up′ & position = position′ - 1
TRANSITION      open_door
    ENTRY               [TIME: open_dur]
                ~door_open
        &       ~door_moving
        &       (       ~moving
                |       moving
                &       EXISTS t: time
                                (       Change(position, t)
                                &       t > Change(moving)))
        &       (       the_elevator_buttons.floor_requested(position)
                |       going_up
                &       (       the_floor_buttons[position].up_requested
                        |       ~request_above(position)
                        &       the_floor_buttons[position].down_requested)
                |       ~going_up
                &       (       the_floor_buttons[position].down_requested
                        |       ~request_below(position)
```

```
                    &       the_floor_buttons[position].up_requested))
        EXIT
                    ~moving
            &       door_moving
            &       going_up =      (       going_up′
                                            &       (
                    request_above′(position′)
                                            |
                    the_floor_buttons[position′].up_requested′)
                                        |       ~request_below′(position′)
                                        &
                    ~the_floor_buttons[position′].down_requested′)
    TRANSITION close_door
        ENTRY           [TIME: close_dur]
                    door_open
            &       ~door_moving
            &       now - t_stop ≥ Change(door_open)
        EXIT
                    door_moving
    TRANSITION door_stop
        ENTRY           [TIME: door_stop_dur]
                    door_moving
            &       now - t_move_door ≥ Change(door_moving)
        EXIT
                    ~door_moving
            &       door_open = ~door_open′
  END   Top_Level
ENDElevator

PROCESS SPECIFICATION       Elevator_Button_Panel
  LEVEL   Top_Level
    IMPORT
        floor, request_dur, clear_dur, the_elevator, the_elevator.position,
        the_elevator.door_open, the_elevator.door_moving
    EXPORT
        floor_requested, request_floor
    VARIABLE
        floor_requested(floor): boolean
    INITIAL
        FORALL f: floor (~floor_requested(f))
    TRANSITION request_floor(f: floor)
        ENTRY           [TIME: request_dur]
                    ~floor_requested(f)
        EXIT
                    floor_requested(f) Becomes True
    TRANSITION clear_floor_request
        ENTRY           [TIME: clear_dur]
                    floor_requested(the_elevator.position)
            &       ~the_elevator.door_open
            &       the_elevator.door_moving
        EXIT
                    floor_requested(the_elevator.position) Becomes False
  END   Top_Level
```

```
    END Elevator_Button_Panel

    PROCESS SPECIFICATION          Floor_Button_Panel
        LEVEL    Top_Level
            IMPORT
                request_dur, clear_dur, the_floor_buttons, the_elevator, the_elevator.position,
                the_elevator.door_open, the_elevator.going_up, the_elevator.door_moving
            EXPORT
                up_requested, down_requested, request_up, request_down
            VARIABLE
                up_requested, down_requested: boolean
            INITIAL
                ~up_requested
              & ~down_requested
            TRANSITION request_up
              ENTRY             [TIME: request_dur]
                        ~up_requested
              EXIT
                        up_requested
            TRANSITION request_down
              ENTRY             [TIME: request_dur]
                        ~down_requested
              EXIT
                        down_requested
            TRANSITION clear_up_request
              ENTRY             [TIME: clear_dur]
                        up_requested
                &       the_floor_buttons[the_elevator.position] = Self
                &       the_elevator.going_up
                &       ~the_elevator.door_open
                &       the_elevator.door_moving
              EXIT
                        ~up_requested
            TRANSITION clear_down_request
              ENTRY             [TIME: clear_dur]
                        down_requested
                &       the_floor_buttons[the_elevator.position] = Self
                &       ~the_elevator.going_up
                &       ~the_elevator.door_open
                &       the_elevator.door_moving
              EXIT
                        ~down_requested
        END      Top_Level
    END Floor_Button_Panel
END     Elevator_System
```

Appendix C: PVS Elevator System Global Specification

```
global: THEORY

BEGIN

astral_lib: LIBRARY = "/fs/rsl/pkgs/kolano/research/pvs/astral2"

IMPORTING astral_lib@astral_defs

% The following IMPORTING clause must be filled out if set_size is
% used on a global set type.  Check definition of cardinality in
% pvs/lib/cardinality/cardinality.pvs for information on the parameters
% required to instantiate cardinality
% IMPORTING astral_lib@astral_set_card[T, m, f]

process: TYPE = {elevator, elevator_button_panel, floor_button_panel}

id: NONEMPTY_TYPE

i_transition: TYPE = {i_request_floor, i_request_up, i_request_down}

the_elevator: id

the_elevator_buttons: id

pos_integer: TYPE = {i: integer | ((const(i)) > (const(0)))(0)}

n_floors: pos_integer

the_floor_buttons: [{I1: int | I1 >= 1 AND I1 <= n_floors} -> id]

Id_Type(ID1: [time -> id])(T1: time): process =
                        IF ID1(T1) = the_elevator
                                THEN elevator
                        ELSIF ID1(T1) = the_elevator_buttons
                                THEN elevator_button_panel
                        ELSE floor_button_panel
                        ENDIF

floor: TYPE = {i: pos_integer | ((const(i)) <= (const(n_floors)))(0)}

pos_real: TYPE = {r: real | ((const(r)) > (const(0)))(0)}

i_parameter: TYPE = [# i_request_floor__f: floor #]

i_undef_parm: i_parameter

i_position: [id -> [time -> floor]]

i_going_up: [id -> [time -> boolean]]

i_door_open: [id -> [time -> boolean]]
```

```
i_moving: [id -> [time -> boolean]]

i_door_moving: [id -> [time -> boolean]]

i_floor_requested: [id -> [[floor] -> [time -> boolean]]]

i_up_requested: [id -> [time -> boolean]]

i_down_requested: [id -> [time -> boolean]]

IMPORTING astral_lib@astral_i_trans[id, i_transition, i_parameter]

i_Eval_Parms(ID1: id, ITR1: i_transition, N1: nat,
                    P1: [time -> i_parameter], T1: time): RECURSIVE bool =
                    (IF N1 = 0 THEN TRUE
                     ELSE    (EXISTS (T2: time):
                                    T2 <= T1 AND
                                    Call1(const(ID1), ITR1, const(T2))(T1)) AND
                     CASES ITR1 OF
                            i_request_floor:
                                    IF      N1      =       1       THEN       i_request_floor__f(P1(T1))      =
                    i_request_floor__f(i_Call_Parms(ID1, ITR1, Call1(const(ID1), ITR1)(T1)))
                                    ELSE TRUE
                            ENDIF
                            ELSE TRUE
                    ENDCASES AND
                    i_Eval_Parms(ID1, ITR1, N1 - 1, P1, T1)
                    ENDIF)
MEASURE (LAMBDA (ID1: id, ITR1: i_transition, N1: nat,
                    P1: [time -> i_parameter], T1: time): N1)

IMPORTING astral_lib@astral_i_trans_parm[id, i_transition, i_parameter, time, i_Eval_Parms]

move_dur: pos_real

arrive_dur: pos_real

open_dur: pos_real

close_dur: pos_real

door_stop_dur: pos_real

request_dur: pos_real

clear_dur: pos_real

t_service_request: pos_real

t_move: pos_real

t_stop: pos_real
```

```
t_move_door: pos_real

global_axiom: AXIOM
                ((((const(t_service_request)) >= (((((const(2)) * (const(n_floors))) - (const(2))) *
                (((((((const(t_move_door)) + (const(door_stop_dur))) + (const(move_dur))) +
                (const(t_move))) + (const(arrive_dur))) + (const(open_dur))) + (const(t_move_door))) +
                (const(door_stop_dur)))) + ((((const(2)) * (const(n_floors))) - (const(3))) *
                ((const(t_stop)) + (const(close_dur)))))) AND (((const(clear_dur)) + ((const(n_floors)) *
                (const(request_dur)))) < (const(t_move_door)))) AND ((const(n_floors)) >=
                (const(2))))(0)

id_domain: AXIOM
                (FORALL (ID1: id):
                        ID1 = the_elevator OR
                        ID1 = the_elevator_buttons OR
                        (EXISTS (I1: {K1: int | K1 >= 1 AND K1 <= n_floors}):
                                ID1 = the_floor_buttons(I1)))

id_unique: AXIOM
                the_elevator /= the_elevator_buttons AND
                (FORALL (I1: {K1: int | K1 >= 1 AND K1 <= n_floors}):
                        the_elevator /= the_floor_buttons(I1)) AND
                (FORALL (I1: {K1: int | K1 >= 1 AND K1 <= n_floors}):
                        the_elevator_buttons /= the_floor_buttons(I1)) AND
                (FORALL (I1, J1: {K1: int | K1 >= 1 AND K1 <= n_floors}):
                        the_floor_buttons(I1) = the_floor_buttons(J1) IMPLIES
                                I1 = J1) AND
                TRUE

END global
```

Appendix D: PVS Elevator Specification

```
elevator: THEORY

BEGIN

IMPORTING global

% The following IMPORTING clause must be filled out if set_size is
% used on a local set type.  Check definition of cardinality in
% pvs/lib/cardinality/cardinality.pvs for information on the parameters
% required to instantiate cardinality
% IMPORTING astral_lib@astral_set_card[T, m, f]

self: {ID1: id | Id_Type(const(ID1))(0) = elevator}

transition: TYPE = {move_up, move_down, arrive, open_door, close_door, door_stop}

parameter: TYPE = [# DUMMY: int #]

undef_parm: parameter

position: [time -> floor]

going_up: [time -> boolean]

door_open: [time -> boolean]

moving: [time -> boolean]

door_moving: [time -> boolean]

request_above(f0: floor): [time -> boolean] =
(EX! (f: [time -> floor]): ((f) > (const(f0))) AND ((((LAMBDA (T1: time):
                    i_floor_requested((const(the_elevator_buttons))(T1))((f)(T1))(T1))) OR ((LAMBDA
                    (T1: time): i_up_requested(((LAMBDA (T1: time):
                    the_floor_buttons((f)(T1))))(T1))(T1)))) OR ((LAMBDA (T1: time):
                    i_down_requested(((LAMBDA (T1: time): the_floor_buttons((f)(T1))))(T1))(T1)))))

request_below(f0: floor): [time -> boolean] =
(EX! (f: [time -> floor]): ((f) < (const(f0))) AND ((((LAMBDA (T1: time):
                    i_floor_requested((const(the_elevator_buttons))(T1))((f)(T1))(T1))) OR ((LAMBDA
                    (T1: time): i_up_requested(((LAMBDA (T1: time):
                    the_floor_buttons((f)(T1))))(T1))(T1)))) OR ((LAMBDA (T1: time):
                    i_down_requested(((LAMBDA (T1: time): the_floor_buttons((f)(T1))))(T1))(T1)))))

Vars_No_Change(T1: time, T2: time): bool =
                    Var_No_Change(position, T1, T2) AND
                    Var_No_Change(going_up, T1, T2) AND
                    Var_No_Change(door_open, T1, T2) AND
                    Var_No_Change(moving, T1, T2) AND
                    Var_No_Change(door_moving, T1, T2)
```

```
Base_Trans(TR1: transition): transition =
        TR1

Duration(TR1: transition): posreal =
        CASES TR1 OF
                move_up: move_dur,
                move_down: move_dur,
                arrive: arrive_dur,
                open_door: open_dur,
                close_door: close_dur,
                door_stop: door_stop_dur
        ENDCASES

Exported(BTR1: {TR1: transition | Base_Trans(TR1) = TR1}): bool =
        FALSE

Has_Parms(BTR1: {TR1: transition | Base_Trans(TR1) = TR1}): bool =
        FALSE

IMPORTING astral_lib@astral_trans[transition, parameter, Duration, Base_Trans,
        Has_Parms, Exported]

Entry_No_Parms(TR1: {tr: transition | NOT Has_Parms(Base_Trans(tr))}):
                [time -> bool] =
        CASES TR1 OF
                move_up:
((((NOT (door_open)) AND (NOT (door_moving))) AND ((LAMBDA (T1: time):
        request_above((position)(T1))(T1)))) AND ((going_up) OR (((NOT (going_up)) AND
        (NOT ((LAMBDA (T1: time): request_below((position)(T1))(T1))))) AND (NOT
        ((LAMBDA (T1: time): i_up_requested(((LAMBDA (T1: time):
        the_floor_buttons((position)(T1)))(T1))(T1)))))) AND ((((End1(arrive, now)) AND
        (NOT ((LAMBDA (T1: time):
        i_floor_requested((const(the_elevator_buttons))(T1))((position)(T1))(T1))))) AND
        (NOT ((LAMBDA (T1: time): i_up_requested(((LAMBDA (T1: time):
        the_floor_buttons((position)(T1)))(T1))(T1))))) OR ((FA! (t: [time -> time]): FA! (t1:
        [time -> time]): ((Change1(moving, t)) AND (Change1(door_open, t1))) IMPLIES (((t)
        < (t1)) AND ((now) >= ((t1) + (const(request_dur))))))),
                move_down:
((((NOT (door_open)) AND (NOT (door_moving))) AND ((LAMBDA (T1: time):
        request_below((position)(T1))(T1)))) AND ((NOT (going_up)) OR (((going_up) AND
        (NOT ((LAMBDA (T1: time): request_above((position)(T1))(T1))))) AND (NOT
        ((LAMBDA (T1: time): i_down_requested(((LAMBDA (T1: time):
        the_floor_buttons((position)(T1)))(T1))(T1)))))) AND ((((End1(arrive, now)) AND
        (NOT ((LAMBDA (T1: time):
        i_floor_requested((const(the_elevator_buttons))(T1))((position)(T1))(T1))))) AND
        (NOT ((LAMBDA (T1: time): i_down_requested(((LAMBDA (T1: time):
        the_floor_buttons((position)(T1)))(T1))(T1))))) OR ((FA! (t: [time -> time]): FA! (t1:
        [time -> time]): ((Change1(moving, t)) AND (Change1(door_open, t1))) IMPLIES (((t)
        < (t1)) AND ((now) >= ((t1) + (const(request_dur))))))),
                arrive:
(moving) AND (((now) - (const(t_move))) >= (Change1(moving))),
                open_door:
(((NOT (door_open)) AND (NOT (door_moving))) AND ((NOT (moving)) OR ((moving) AND ((FA! (t:
        [time -> time]): FA! (t1: [time -> time]): ((Change1(position, t)) AND
```

```
        (Change1(moving, t1))) IMPLIES ((t) > (t1))))))) AND ((((LAMBDA (T1: time):
        i_floor_requested((const(the_elevator_buttons))(T1))((position)(T1))(T1))) OR
        ((going_up) AND (((LAMBDA (T1: time): i_up_requested(((LAMBDA (T1: time):
        the_floor_buttons((position)(T1))))(T1))(T1))) OR ((NOT ((LAMBDA (T1: time):
        request_above((position)(T1))(T1)))) AND ((LAMBDA (T1: time):
        i_down_requested(((LAMBDA (T1: time):
        the_floor_buttons((position)(T1))))(T1))(T1))))))) OR ((NOT (going_up)) AND
        (((LAMBDA (T1: time): i_down_requested(((LAMBDA (T1: time):
        the_floor_buttons((position)(T1))))(T1))(T1))) OR ((NOT ((LAMBDA (T1: time):
        request_below((position)(T1))(T1)))) AND ((LAMBDA (T1: time):
        i_up_requested(((LAMBDA (T1: time):
        the_floor_buttons((position)(T1))))(T1))(T1))))))),
                close_door:
((door_open) AND (NOT (door_moving))) AND (((now) - (const(t_stop))) >= (Change1(door_open))),
                door_stop:
(door_moving) AND (((now) - (const(t_move_door))) >= (Change1(door_moving)))
        ENDCASES

Entry_Parms(TR1: {tr: transition | Has_Parms(Base_Trans(tr))}, P1: parameter):
                [time -> bool] =
        const(TRUE)

Exit_No_Parms(TR1: {tr: transition | NOT Has_Parms(Base_Trans(tr))})
                (T1: {T1: time | T1 >= Duration(TR1)}): bool =
        (CASES TR1 OF
                move_up:
((moving) AND (going_up)) AND (LAMBDA (T1: time): position(T1) = position(T1 -
        Duration(move_up))) AND (LAMBDA (T1: time): door_open(T1) = door_open(T1 -
        Duration(move_up))) AND (LAMBDA (T1: time): door_moving(T1) =
        door_moving(T1 - Duration(move_up))),
                move_down:
((moving) AND (NOT (going_up))) AND (LAMBDA (T1: time): position(T1) = position(T1 -
        Duration(move_down))) AND (LAMBDA (T1: time): door_open(T1) = door_open(T1 -
        Duration(move_down))) AND (LAMBDA (T1: time): door_moving(T1) =
        door_moving(T1 - Duration(move_down))),
                arrive:
((((LAMBDA (T1: time): going_up(T1 - Duration(arrive)))) AND ((position) = (((LAMBDA (T1: time):
        position(T1 - Duration(arrive)))) + (const(1))))) OR ((NOT ((LAMBDA (T1: time):
        going_up(T1 - Duration(arrive))))) AND ((position) = (((LAMBDA (T1: time):
        position(T1 - Duration(arrive)))) - (const(1)))))) AND (LAMBDA (T1: time):
        going_up(T1) = going_up(T1 - Duration(arrive))) AND (LAMBDA (T1: time):
        door_open(T1) = door_open(T1 - Duration(arrive))) AND (LAMBDA (T1: time):
        moving(T1) = moving(T1 - Duration(arrive))) AND (LAMBDA (T1: time):
        door_moving(T1) = door_moving(T1 - Duration(arrive))),
                open_door:
(((NOT (moving)) AND (door_moving)) AND ((going_up) = ((((LAMBDA (T1: time): going_up(T1 -
        Duration(open_door)))) AND (((LAMBDA (T1: time):
        request_above((position)(T1))(T1 - Duration(open_door)))) OR ((LAMBDA (T1: time):
        i_up_requested(((LAMBDA (T1: time): the_floor_buttons((position)(T1))))(T1))(T1 -
        Duration(open_door)))))) OR ((NOT ((LAMBDA (T1: time):
        request_below((position)(T1))(T1 - Duration(open_door))))) AND (NOT ((LAMBDA
        (T1: time): i_down_requested(((LAMBDA (T1: time):
        the_floor_buttons((position)(T1))))(T1))(T1 - Duration(open_door))))))))) AND
        (LAMBDA (T1: time): door_open(T1) = door_open(T1 - Duration(open_door))),
```

```
                close_door:
(door_moving) AND (LAMBDA (T1: time): position(T1) = position(T1 - Duration(close_door))) AND
                (LAMBDA (T1: time): going_up(T1) = going_up(T1 - Duration(close_door))) AND
                (LAMBDA (T1: time): door_open(T1) = door_open(T1 - Duration(close_door))) AND
                (LAMBDA (T1: time): moving(T1) = moving(T1 - Duration(close_door))),
                door_stop:
((NOT (door_moving)) AND ((door_open) = (NOT ((LAMBDA (T1: time): door_open(T1 -
                Duration(door_stop)))))) AND (LAMBDA (T1: time): position(T1) = position(T1 -
                Duration(door_stop))) AND (LAMBDA (T1: time): going_up(T1) = going_up(T1 -
                Duration(door_stop))) AND (LAMBDA (T1: time): moving(T1) = moving(T1 -
                Duration(door_stop)))
                ENDCASES)(T1)

Exit_Parms(TR1: {tr: transition | Has_Parms(Base_Trans(tr))}, P1: parameter)
                (T1: {T1: time | T1 >= Duration(TR1)}): bool =
                TRUE

IMPORTING astral_lib@astral_trans_ext[transition, parameter, Duration,
                Base_Trans, Has_Parms, Exported, time, Entry_No_Parms, Exit_No_Parms,
                Entry_Parms, Exit_Parms]

Initial: [time -> bool] =
                (((((position) = (const(1))) AND (going_up)) AND (NOT (door_open))) AND (NOT
                (moving))) AND (NOT (door_moving))

Environment: [time -> bool] =
                const(TRUE)

Imported_Variable: [time -> bool] =
                const(TRUE)

Invariant: [time -> bool] =
                const(TRUE)

Schedule: [time -> bool] =
                const(TRUE)

Constraint(T1: time): bool =
                (FORALL (TR1: transition):
                        End1(TR1, const(T1))(T1) IMPLIES
                                ((((going_up) AND (NOT ((LAMBDA (T1: time): going_up(T1 -
                Duration(TR1)))))) IMPLIES (NOT ((LAMBDA (T1: time): request_below(((LAMBDA
                (T1: time): position(T1 - Duration(TR1))))(T1))(T1 - Duration(TR1)))))) AND (((NOT
                (going_up)) AND ((LAMBDA (T1: time): going_up(T1 - Duration(TR1))))) IMPLIES
                (NOT ((LAMBDA (T1: time): request_above(((LAMBDA (T1: time): position(T1 -
                Duration(TR1))))(T1))(T1 - Duration(TR1)))))))(T1))

Further_Environment_1: [time -> bool] =
                const(TRUE)

Constant_Refinement_1: [time -> bool] =
                const(TRUE)

Elgible_Set_1(T1: time): set[transition] =
```

```
                Enabled_Set(T1)

Transition_Selection_1(T1: time): bool =
                (FORALL (TR1: transition):
                        Fired(TR1, T1) IMPLIES member(TR1, Elgible_Set_1(T1)))

local_axiom: AXIOM
                (const(TRUE))(0)

self_imports: AXIOM
                i_position(self) = position AND
                i_going_up(self) = going_up AND
                i_door_open(self) = door_open AND
                i_moving(self) = moving AND
                i_door_moving(self) = door_moving

END elevator
```

Appendix E: PVS Elevator Local Proof Obligations

```
elevator_obl: THEORY

BEGIN

IMPORTING elevator
IMPORTING astral_lib@astral_axioms[transition, time, Base_Trans,
                        Duration, Exported, Issued_Call, Entry, Exit, Enabled, Fired,
                        Initial, Vars_No_Change]

t0: time
t1, t2: VAR time

% local invariant base case
elevator_inv_base: THEOREM
        Invariant(0)

% local invariant induction case
elevator_inv_ind: THEOREM
        (FORALL (t1): t1 <= t0 IMPLIES Invariant(t1)) IMPLIES
                (FORALL (t2): t2 > t0 IMPLIES Invariant(t2))

% constraint
elevator_con: THEOREM
        (FORALL (t1): Constraint(t1))

% local schedule base case
elevator_sch_base_1: THEOREM
        Invariant(0) AND
        Environment(0) AND
        Further_Environment_1(0) AND
        Constant_Refinement_1(0) AND
        Transition_Selection_1(0) IMPLIES
                Schedule(0)

% local schedule induction case
elevator_sch_ind_1: THEOREM
        (FORALL (t1):
                Invariant(t1) AND
                Environment(t1) AND
                Further_Environment_1(t1) AND
                Constant_Refinement_1(t1) AND
                Transition_Selection_1(t1)) AND
        (FORALL (t1): t1 <= t0 IMPLIES Schedule(t1)) IMPLIES
                (FORALL (t2): t2 > t0 IMPLIES Schedule(t2))

END elevator_obl
```