Approximation Techniques for Using the ASTRAL Symbolic Model Checker as a Specification Debugger *

Zhe Dang
The School of Electrical Engineering and
Computer Science
Washington State University
Pullman, WA 99164 USA
+ 1 509 335-7238
zdang@eecs.wsu.edu

and

Richard A. Kemmerer
Reliable Software Group
Computer Science Department
University of California
Santa Barbara, CA 93106 USA
+ 1 805 893-4232
kemm@cs.ucsb.edu

*Part of the material in this paper was presented in [27, 28]. The work was done when Zhe Dang was a PhD student in the Reliable Software Group of the Computer Science Department at University of California, Santa Barbara.

Abstract

ASTRAL is a high-level formal specification language for real-time systems. This paper presents a symbolic model checker that translates an ASTRAL process instance to a labeled transition system with each transition representable by a Presburger formula. The labeled transition system is unfolded into the execution tree of an ASTRAL process and the Omega library is used to carry out the image computations. Different levels of approximation of the environment behaviors of the instance are considered, as well as symbolic search strategies including depth-first search, breadth-first search, and depth-breadth search. Three approximation techniques to speed up the model checking process for use in debugging a specification are also presented. They are random walk, partial image and dynamic environment generation. Ten mutation tests on a railroad crossing benchmark are used to compare the performance of the techniques applied separately and in combination. The test results are presented and analyzed.

1 Introduction

A real-time system can be defined as a system that performs its functions and responds to external events within a specified amount of time. Telephone switches, robot controllers and air-traffic controllers are examples of real-time systems. Verifying that real-time systems, in particular for safety-critical applications, satisfy both functional correctness requirements and timeliness requirements is extremely important; errors in real-time system designs may cause catastrophic consequences such as loss of human lives.

Model-checking [17] is an automatic procedure that exhaustively explores the state space of a finite-state system in checking whether a temporal property is satisfied. In recent years, symbolic model checkers like SMV [49] and explicit state model checkers like SPIN [42] have been successful in automatically verifying large finite-state systems describing protocols, hardware devices and reactive systems. These successes have naturally raised the question of whether model-checking techniques are applicable for some infinite state systems, such as real-time systems. Timed automata [2] are a popular model for real-time systems. A timed automaton is basically a finite automaton augmented with a number of real-valued (or integer-valued) clocks. Clocks and the difference of two clocks can be compared to a finite number of integer constants. These comparisons are called (simple) clock constraints. Recent results show that a real-time system modeled as a timed automaton can be model-checked against (a part of) a number of real-time logics [1, 3, 4, 5, 40, 52, 55].

However, not every real-time system can be modeled as a timed automaton. A complex (not necessarily large) real-time system might contain complex clock constraints like $\mathtt{Start(T_1)} - \mathtt{Start(T_2)} > \mathtt{d}$ (i.e., the difference between $\mathtt{Start(T_1)}$ and $\mathtt{Start(T_2)}$ is greater than a parameterized or an unspecified constant $\mathtt{d}$), or like $\mathtt{Start(T_1)} - \mathtt{Start(T_2)} > \mathtt{Start(T_3)} - \mathtt{Start(T_4)}$ (i.e., the difference of the (last) start times of transitions $\mathtt{T_1}$ and $\mathtt{T_2}$ is greater than that of transitions $\mathtt{T_3}$ and $\mathtt{T_4}$). It is known [2] that these complex real-time systems have Turing computing power and thus eliminate the possibility of decidable verification even for nontrivial safety properties.

On the other hand, real-world applications need a specification language that can specify complex real-time systems with a number of clocks, a number of untimed (infinite state) variables, complex timing requirements and/or clock constraints, as well as complex interactions between individual components. ASTRAL [21] is one of these high-level formal specification languages for complex real-time systems. It is provided with structuring mechanisms that allow one to build modularized specifications of complex systems with layering. In ASTRAL, a real-time system is modeled by a collection of process specifications and a single global specification. Each process specification consists of a sequence of levels; each level is an abstract view of the process being specified. ASTRAL has been successfully used to specify a number of interesting real-time systems, including a CCITT system [22], a complex wide-area phone system composed of several ASTRAL specifications [21], a hardware description language [14], a robot control system [9], cryptographic protocols [25], and Mobile IP [26].

The ASTRAL Software Development Environment (SDE) [47] is an integrated set of design and analysis tools, which includes, among others, a symbolic model-checker and a mechanical theorem-prover. Since complex real-time systems specified by ASTRAL can achieve Turing computing power, ASTRAL no longer has an automatic verification procedure. Therefore, unlike other model-checkers, the ASTRAL model-checker is primarily intended for use as a specification debugger or a specification testing tool, before a specification is processed by the (semi-automatic) theorem-prover. Once complete verification is out of the question, a number of aggressive approximation techniques can be used in order to speed-up the debugging process of the model-checker while remaining effective in finding violations. This is the central issue of this paper.

Our previous work. The ASTRAL explicit-state model checker [26, 47] generates customized C++ code for each specification and enumerates all the branches of execution of this implementation up to a system time bound set by the user. Strictly speaking, the model checker only tests the specification under a set of given constant values. The main drawback of the model checker is that, when it is used as a specification debugger, the constants must be carefully chosen to assure that something "interesting" actually occurs within the given system time bound. This led us to build a symbolic model checker [27] that tests specifications at the process level and requires only limited input to set up constant values. By *symbolic*, we mean that a set of reachable states is represented as a Presburger formula[1]. Using the Omega library [51], this set is further represented by a union of convex regions. The union is called an *image* with the number of the convex regions as its *size*. Thus, a set of reachable states can be regarded as an image. The symbolic model checker performs image computations on the execution tree (with a pre-assigned depth) of an ASTRAL process that is trimmed by the execution graph of the process. Each node in the tree represents a transition, i.e., a subset of state pairs representable as a Presburger formula. The model checker traverses the tree in a depth-first manner and calculates the postimage for each node on the current path. The symbolic model checker was used to test a railroad crossing benchmark [27]. In those experiments the model checker aborted before completion for two of the test cases due to the large size of the specification instances. Because the model checkers in the ASTRAL SDE are only intended to be used for debugging purposes and the properties specified using ASTRAL are essentially safety properties, we usually do not need to compute the entire set of reachable states to uncover a flaw in a specification. Therefore, a natural question would be: what are the practical approximation techniques that allow the search procedure to complete while still being effective in debugging?

[1] A Presburger formula, which will be defined later, is an integer arithmetic formula with quantification. Since we are using Presburger formulas, time is considered discrete in this paper.

Contributions of the paper. Since, in this paper, Presburger formulas are used to represent images, we consider a subset of ASTRAL, called Small-ASTRAL, such that each Small-ASTRAL specification can be translated into a labeled transition system with each transition representable by a Presburger formula. The translation extends the previous one [27] in the following way. During the execution of an ASTRAL transition[2] in an ASTRAL process instance, different levels of approximation of the environment behaviors of the instance are considered. The approximations differ in the accuracy of modeling the environment and the cost of image computations. The labeled transition system is unfolded into the execution tree of an ASTRAL process and the Omega library is used to carry out the image computations. The strategies used in the image computations on the tree are depth-first search, breadth-first search, and depth-breadth search. The latter two were not presented in our previous work [28]. Due to the fact that the number of nodes in the tree is exponential in the search depth, depth-first search is not ideal for a simple specification. On the other hand, for a complex specification, the image size grows dramatically at each iteration when breadth-first search is used, such that it is possible to complete the search only for a small number of iterations. The depth-breadth search combines the two strategies and automatically switches between the two according to the size of the image at a node.

Besides different levels of approximation on the environment of a process instance, approximation techniques are also desirable for the image computations on the execution tree of the instance so that violations can be found in a shorter time. In this paper, three techniques to meet this need are introduced. They are random walk, partial image and dynamic environment generation. The random walk technique is used to allow the model checker to randomly skip a number of branches when traversing the execution tree. The partial image technique picks a subset of the preimage and uses this subset to calculate the postimage at each node. The dynamic environment generation technique generates a different sequence of imported variable values for different execution paths. The three techniques were briefly presented in our previous paper [28]. In this paper, we demonstrate the effectiveness of the three aggressive approximations in debugging a specification in a larger set of experiments.

Related work. The underlying computation model for Small-ASTRAL and the model checker is based upon labeled transition systems. The model was widely used in various contexts, from concurrent programs [53] to timed automata [2]. The experience gained from our work, therefore, will not be limited to the ASTRAL environment. It should be applicable to debugging specifications written in many other expressive languages as long as safety properties are concerned.

The symbolic model checker in this paper carries out symbolic search up to a preassigned depth. This type of *bounded model checking* approach was used in [6] for a form of parameterized timed automata. But the image computations in [6] are essentially breadth-

[2]In ASTRAL each transition must have positive duration.

first search. That is, the symbolic representation of all the reachable states is calculated at each iteration. Our experience shows that breadth-first search does not work well for large specifications. The reason is that the image size at each iteration step grows dramatically and can easily get out of control for the model checker. Therefore, our model checker works on the execution tree of a transition system. In addition to breadth-first search, we also propose depth-first search and depth-breadth search strategies, which, as far as we know, are not covered by previous studies in symbolic verification/debugging infinite state systems, though both depth-first search and breadth-first search are adopted in a number of finite state model checkers [42, 19, 30, 49]. The depth-breadth search technique in this paper is closely related to the mixed depth-first/breadth-first approaches for BDD-based [10, 11] finite state systems [35]. But our depth-breadth search technique targets at infinite state systems representable by Presburger formulas instead of finite state systems represented in BDDs. Our method of image computations on an execution tree can be traced back to the earlier symbolic execution techniques for programs [15, 45, 44] as well as for specifications [32]. However, the major difference is that our procedure is fully automatic and applicable to real-time systems.

An environment is tied to an ASTRAL transition at different levels of approximation: from the cheapest single-event assumptions (i.e., assuming, during the transition's execution, there is at most one event occurring, such as a change of an environmental clock.) to the most expensive multiple-event assumptions (i.e., assuming that multiple events could happen). Since the model checker is essentially a debugger, a user need not go for a higher level if using a lower level approximation of the environment successfully detects a violation. This is influenced by the general idea of layered specifications and verification, such as the work of Gerber and Lee [37] and HMS [36], as well as in ASTRAL [21].

The three approximation techniques for image computations (i.e., random walk, partial image and dynamic environment generation) are influenced by advocates for using lightweight formal methods [8]. We prefer to consider the three techniques to be natural rather than original, though we are not not aware of their use in symbolic model checking/debugging of infinite state real-time systems. The name "random walk" is borrowed from the theory of stochastic processes. In particular, the *random simulation* technique has been used in protocol testing for many years [41], by exercising a random path of a protocol. But random simulation is essentially explicit state based, unlike our use of it for symbolic model checking on the execution tree of a specification. The partial image technique was inspired by traditional techniques in program testing, such as sampling and random testing methods [34, 33] as well as domain testing [54]. However, instead of picking a single or several samples from the domain, the partial image technique selects a subset of the preimage and uses this subset to calculate the postimage at each node. Dill and Wong-Toi [31] considered the *under-approximation* technique for reachable sets of timed automata. These sets are pairs of control states and clock regions. Our partial image technique, however, is built on the representation of Presburger formulas via the Omega library. Thus, it is applicable to a wider range of transition systems, including infinite-state real-time systems. In addition, the work of Dill and Wong-Toi does not consider depth-first and depth-breadth search strategies with which our partial image technique can be combined. The dynamic environment generation technique is similar to the idea of Colby, Godefroid and Jagadeesan [19] in that both address the problem of automatically closing an open system, in which some of the components are not present. Their approach targets concurrent programs, instead of real-time specifications. Unlike doing static analysis of a concurrent program, our technique dynamically selects a reasonable environment according to the imported variable clause of the ASTRAL specification. With concrete values for the environment as well as the history, the cost of later image calculations can be greatly decreased.

Bultan et. al. [13, 12] used the Omega library as a tool to symbolically represent a set of states that is characterized by a Presburger formula. They also investigated partitions and approximations in order to calculate fixed points. As in the work presented in this paper, they worked with infinite state systems. However, the systems they considered are "simple" in the following sense: (1) quantifications are only limited to a very small number, (2) the transition system is a straightforward history-independent transition system; i.e., the current state only depends upon the last state, and other history references are not allowed, (3) the transition system itself is not a real-time system in the sense that no duration is attached to a transition and the start and end times are not allowed to be referenced. Unfortunately, a typical ASTRAL specification, such as the benchmark considered in this paper, is not "simple". For these complex systems, a fixed point (i.e., the set of all reachable states) may not be computable. However, because the ASTRAL symbolic model checker is primarily intended to be used as a debugger instead of a verifier, calculating the fixed point of a transition system is not an important issue. Therefore, the approaches of Bultan et. al., as well as other approximation techniques, such as abstraction [20, 18, 23] and over-approximation [31], are orthogonal to the approaches presented in this paper.

The model checker considered in this paper is modularized; one need only check one process instance for each process type declared, without looking at the transition behaviors of other process instances. The STeP system also uses a modularized approach [7]. However, STeP primarily uses a theorem prover to validate a property while the approach presented here uses a fully automatic model checker. Comparisons between ASTRAL and other specification languages on a wide range of language design issues can be found in the ASTRAL overview paper [21].

Outline of the paper. The remainder of this paper is organized as follows. In Section 2 a brief overview of the ASTRAL specification language is presented, along with an introduction to the ASTRAL modularized proof theory. In Section 3 the translation procedure from a Small-ASTRAL process instance to a labeled transition system is demonstrated. In Section 4 the symbolic search strategies for the model checker are provided, including depth-first search, depth-breadth search, and breadth-first search. In Section 5 three approximation techniques, i.e., random walk, partial image and dynamic environment generation, are introduced. In Section 6 the results of using different search strategies and approximation techniques on ten mutation tests are given and analyzed. Finally, in Section 7 conclusions are drawn from this work, and future areas of research are proposed.

2 ASTRAL Overview

A railroad crossing specification is used as a benchmark example throughout the remainder of this paper to illustrate various features of ASTRAL. The system description is taken from [39]. The system consists of a set of railroad tracks that intersect a street where cars may cross the tracks. A gate is located at the crossing to prevent cars from crossing the tracks when a train is near. A sensor on each track detects the arrival of trains on that track. The critical requirement of the system is that whenever a train is in the crossing the gate must be down, and when no train has been in between the sensors and the crossing for a reasonable amount of time, the gate must be up. The complete ASTRAL specification of the railroad crossing system can be found in the appendix.

An ASTRAL specification includes a global specification and process specifications. The global specification contains declarations of process instances, global constants, nonprimitive types that may be shared by process types, and system level critical requirements. There is a process specification for each process type declared in the global specification.

2.1 Processes, Constants, Variables, and Types

The global specification begins with a process type declaration:

```
PROCESSES
   the_gate: Gate,
   the_sensors: array [ 1..n_tracks ]  of Sensor.
```

This declaration indicates that there are `n_tracks` sensor instances of type `Sensor` in the system, where `n_tracks` is a global constant declared in the constant declaration part:

```
CONSTANT
   n_tracks: pos_integer.
```

In a global specification, the number of process instances must be fixed, but could be a parameterized constant, e.g. `n_tracks`. This feature allows a user to specify a system with a parameterized number of process instances.

ASTRAL is a strongly typed language. Primitive types include `Integer, Real, Boolean, ID` and `Time`. Constructed types can be declared in the type declaration parts of the specification by using the `TYPEDEF` construct. For instance, `pos_integer` is defined as positive integers:

```
TYPE
   pos_integer: TYPEDEF i: integer ( i > 0 ).
```

The type `ID` is one of the primitive types of ASTRAL. Each process instance has a unique identifier. The ASTRAL specification function `IDTYPE(i)` returns the type of the process with identifier `i`. The `IDTYPE` function is used in the global declaration to define `sensor_id`, which represents all identifiers of process instances of type `Sensor`:

```
TYPE
   sensor_id: TYPEDEF i: id ( IDTYPE ( i ) = Sensor ).
```

In the railroad crossing specification there are two process specifications, `Gate` and `Sensor`. A process specification includes an interface section which specifies the imported variables, types, transitions and constants (from either the global specification, or exported by other processes) used by the process, and the variables and transitions exported by the process. ASTRAL does not have global variables. Therefore, variables must be declared in each process specification. The format used to declare variables is similar to that of the constant declaration above. Also, each process instance can have its own constants. Constants with the same names in distinct process instances do not necessarily share the same value. ASTRAL supports a modularized design principle: every variable is associated with a unique process instance, and changes to the variable can only be caused by the transitions specified inside that process instance. This is discussed further in the next subsection.

2.2 Transitions

The ASTRAL computation model is defined by the execution of state transitions. Transitions are only specified inside process specifications. Therefore, each transition in a process instance can only change the variables specified inside that process instance. The body of an ASTRAL transition includes pairs of entry and exit assertions with a positive duration indicated for each pair. The entry assertion must be satisfied at the time the transition starts, whereas the exit assertion will hold after the time indicated by the duration from when the transition fires. For example, in process `Gate`, the transition,

```
TRANSITION up
   ENTRY          [ TIME : up_dur ]
             position = raising
           & now - End( raise )  >= raise_time
```

```
EXIT
          position = raised,
```

specifies the gate being fully raised, after it has been rising for a reasonable amount of time (`raise_time`). The duration of this transition is indicated by the constant `up_dur`. In ASTRAL, `End(T,t)` is a predicate that is true if and only if the transition `T` ends at time `t` and there is no other time after `t` and before the current time (`now`) when `T` ends. `End(T)` is used to indicate the time `t` such that `End(T,t)` holds. `Start(T)` is defined similarly for the start time of `T`. A transition instance is fired if its entry assertion is satisfied and no other transition in the same process instance is executing. The execution of this transition instance is completed after the duration indicated in the transition specification. A transition can be exported. In this case, its entry assertion alone can not decide whether it is firable. According to the ASTRAL semantics, an exported transition must be called from the external environment in order to fire. `Call(T)` is used to indicate the time when a call to the exported transition `T` is made. ASTRAL broadcasts variable values instantaneously at the time that a transition finishes. Other process instances may refer to these variables as well as the start and end times of transitions under the assumption that these variables and the transitions are exported and the listening process instance properly imports them.

When it is the case that there is more than one transition instance that is enabled inside the same process instance, one of the enabled transitions is nondeterministically chosen to fire, assuming that there is no other transition executing at that moment. Inside a process instance, executions of transitions are non-overlapping and interleaved, while between process instances, maximal parallelism is supported. Thus, the execution of transition instances in different process instances is truly concurrent.

2.3 Assumptions and Critical Requirements

In addition to transitions, requirement descriptions are also included as a part of an ASTRAL specification. They comprise axioms, initial clauses, imported variable clauses, environmental assumptions and critical requirements. An axiom is usually used to specify a property about constants. An initial clause defines the possible system states at startup time. An imported variable clause defines the properties the imported variables should satisfy, for instance, patterns of changes to the values of imported variables and timing information about transitions exported from other processes. ASTRAL uses environmental assumptions to characterize the environment. An environment clause formalizes the assumptions that must hold on the behavior of the environment to guarantee some desired system properties. Typically, it describes the pattern of invocation (`Calls`) of exported transitions. The critical requirements include invariant clauses and schedule clauses. An invariant expresses the properties that must hold for every state of the system that is reachable from the initial state, no matter what the behavior of the external environment is. A schedule expresses the properties that must hold provided the external environment and the other processes in the system behave as assumed (i.e., as specified by the environmental assumptions and the imported variable clauses). Both invariants and schedules are safety properties.

ASTRAL is a rich language and has strong expressive power. For a detailed introduction to ASTRAL and its formal semantics the reader is referred to [21, 22, 47].

2.4 Use of Time

Since ASTRAL is intended to specify complex real-time systems, timing constructs are extensively used in a typical ASTRAL specification. As an example, below is the imported variable clause of process `Gate`.

```
FORALL s: sensor_id
   ( Change( s.train_in_R, now )
     & ~s.train_in_R
  ->   0 <= now -  ( RIImax - response_time )
      & FORALL t: time
         ( now -  ( RIImax - response_time )  <= t
           & t < now
        ->   Past( s.train_in_R, t )  )  )
```

It says that once a sensor reports a train, it will keep reporting a train at least as long as it takes the fastest train to exit the region. In the formula, `now` is the global clock indicating the current time. `Change(s.train_in_R, t)` is used to indicate the variable `s.train_in_R` changes at time `t`. `Past(s.train_in_R, t)` stands for the past value of the variable `s.train_in_R` at time `t`. In addition to these, a number of quantifications can be applied, e.g., `FORALL t:time`. Thus, timings are explicitly used in ASTRAL in the following ways:

- A transition has a positive duration that can be parameterized,
- Clocks, including `now` and `Start`, `End`, `Call` times of a transition, as well as change times of a variable, can be referenced,
- Histories of a variable can be referenced using `Past`,
- A number of quantifications over time can be applied to a formula,
- Clocks can be involved in an arithmetic expression containing parameterized constants. For instance, `now-(RIImax-response_time)` appears in the imported variable clause.

2.5 Modularized Proof Theory

Modularization means the principle that a system specification can be broken into several loosely coupled functional modules. This greatly eases both the verification and the design work. Although most high level specification languages support modularization, each module in the specification is only a syntactical module. That is, those languages provide a way to write a specification as several modules; however, there is no way to verify the correctness of each process without looking at all the behaviors of all the other processes. In verifying a real-time system, it is extremely important to restrict the size of the system. In ASTRAL, a specification is composed of several process specifications. A process instance can be understood as a module, which provides an interface section including a declaration of the imported variables and transitions and an imported variable clause, as was mentioned before. This helps to develop a modular verification theory for real-time systems.

The ultimate goal of modularization is to partition a large system, both conceptually and functionally, into several small modules and to verify each small module instead of verifying the large system as a whole. This can be done in ASTRAL, according to the ASTRAL proof theory [22], which is briefly introduced below. For example, the global invariant of an ASTRAL specification can be verified by using only the invariants for each process instance. It is not necessary to look at the details of each process instance's behavior. The global schedule can be verified by using only the global environment and the schedules for each process instance. In turn, verifying the schedule and the invariant of each process instance uses only the process's local assumptions and behaviors. For instance, verifying the local invariant uses only the behaviors of transitions of the process instance. Verifying the local schedule uses the process's local environment and imported variable clause, plus the behaviors of the process's transitions. The imported variable clause must be a correct assumption and needs to be verified by combining all the invariants from all the other process instances. Based upon this theory, this paper only discusses process-level model-checking for ASTRAL.

3 Building a Labeled Transition System

This section presents a procedure to translate a process instance's local requirements, assumptions and transition system into Presburger formulas, whenever possible. Presburger formulas [48] are arithmetic formulas over integer variables, which are built from logical connectives and quantifiers. The following grammar for generating Presburger formulas is adapted from [13],

$$f ::= \; t \leq t \mid (f) \mid f \wedge f \mid \neg f \mid \exists \mathbf{intvar}(f),$$

$$t ::= \; (t) \mid t + t \mid t - t \mid \mathbf{intvar} \mid \mathbf{intcons},$$

where **intvar** and **intcons** are integer variables and integer constants, respectively. It is easy to see that one can use the grammar to represent a formula containing $<, \neq, \vee, \forall$ as well as multiplication by an integer constant. The complexity of solving Presburger formulas is extremely high ($O(2^{2^{2^n}})$) [50]. The Omega library was developed by Pugh for manipulating integer tuple relations and sets that are characterized by Presburger formulas [51]. Using the Omega library, the solution space of a Presburger formula can be compactly represented by an Omega set or relation that is a union of convex regions. Such a representation is defined as an *image* with its *size* given as the number of unions. Each convex region in an image is represented by a conjunction of linear constraints. In addition, the Omega library provides rich operations on Omega sets and relations, such as join, intersection and projection. For *practical* formulas, especially those with less alternations of quantifications, we found the solving time via the Omega library to be affordable in many cases.

3.1 Modeling

A process instance is modeled by a labeled transition system $\mathcal{T}$:

$$(Q, \mathbf{Init}, \rightarrow_{a\in\Sigma}, \mathbf{Assump}, \mathbf{Prop})$$

that consists of a set Q of (infinitely many) states, a finite set of transitions $\rightarrow_a$ with name a from Σ. Each $\rightarrow_a$ is a relation on Q, i.e.,

$$\rightarrow_a \subseteq Q \times Q.$$

Init $\subseteq Q$ are the initial states. The assumption **Assump** and the property **Prop** of $\mathcal{T}$ are also subsets of states Q. As usual, for a set of states $R \subseteq Q$, we define the preimage, $Pre_a(R)$, of a transition $\rightarrow_a$ as the set of all states from which a state in R can be reached by this transition:

$$Pre_a(R) = \{q : \exists q' \in R \text{ s.t. } q \rightarrow_a q'\}.$$

Similarly, the postimage, $Post_a(R)$, of a transition $\rightarrow_a$ is the set of all states that are reachable from a state in R by this transition:

$$Post_a(R) = \{q : \exists q' \in R \text{ s.t. } q' \rightarrow_a q\}.$$

The semantics of $\mathcal{T}$ is characterized by *runs*

$$q_0 a_1 q_1 a_2 \cdots$$

such that for all i,

$$q_i \rightarrow_{a_{i+1}} q_{i+1} \quad \text{and} \quad q_0 \in \mathbf{Init}.$$

$\mathcal{T}$ is *correct* with respect to its specification, if for any run

$$q_0 a_1 q_1 a_2 \cdots$$

of $\mathcal{T}$, the following condition is satisfied for all k,

$$\{q_0, \cdots, q_k\} \subseteq \textbf{Assump} \text{ implies } q_k \in \textbf{Prop}.$$

Since the Omega library is used to calculate the symbolic representation of a subset of the states, $\mathcal{T}$ is further restricted to have the form in which the components Q, **Init**, $\rightarrow_{a\in\Sigma}$, **Assump** and **Prop** can be expressed as Presburger formulas. As a result, the model-checker presented in this paper handles a subset of ASTRAL, called Small-ASTRAL. Intuitively, Small-ASTRAL is constrained such that each formula in a Small-ASTRAL specification can be represented by a Presburger formula. The detailed grammar and definition of Small-ASTRAL can be found in [24].

Since we use Presburger formulas, the primitive type `Real` is excluded from Small-ASTRAL. What if a specification contains real-valued variables? In most specifications we investigated, real-valued variables can be safely regarded as integer-valued. In particular, when debugging an infinite state real-time system, which is exactly the primary use of the model checker in this paper, an error within the discrete domain usually can be mapped into the corresponding error in the continuous domain. However, doing this is not always safe; i.e., it can cause false negatives. A quick example is the fact that the density of reals, e.g., $\forall x, y \exists z(x > y \rightarrow x > z \wedge z > y)$, is no longer true for integers. Since we are not aware of a practical solver for first-order arithmetic formulas with mixed domains including real variables and integer variables, we prefer in this paper to only look at discrete transition systems augmented with complex timing constraints and discrete clock values, and leave checking possible false negatives caused by treating real-valued variables as integer-valued ones to the user.

Complex timing constraints make ASTRAL a powerful language for specifying real-time systems, and they are also the major research area for real-time verification. Thus, we decided that all the time-related constructs are included in Small-ASTRAL in order to retain the most important timing features from ASTRAL. They include `Start, End, Call, Change, Past`.

Small-ASTRAL in fact is not really small. For instance, it includes the benchmark specification in this paper and most (discrete) timed automata-based specifications. In short, Small-ASTRAL is capable of specifying discrete timed systems containing the history-dependent construct `Past` and Presburger formulas over clocks and parameterized constants.

In the following subsections, we show how to translate a Small-ASTRAL specification into a labeled transition system $\mathcal{T}$.

3.2 Handling Constants and Local variables

A process instance may use a number of global and local constants, as well as local and imported (from other process instances) variables. In the current implementation, they are translated into integer variables. For example, in the benchmark specification, the local variable `position` of the process `Gate` is an enumerated type

`gate_position :  (raised, raising, lowered, lowering)`.

This variable is represented by an integer variable

$$position$$

with the type assumption

$$position = 1 \vee position = 2 \vee position = 3 \vee position = 4$$

added to the **Assump** of $\mathcal{T}$.

Some constant values should be known in advance in order to carry out the symbolic model-checking procedure. There are two categories of these constants. One is the global constants that decide the number of process instances in the whole system. For example, the global constant `n_tracks` characterizing the number of `Sensor` process instances belongs to this category. Another category is the constants involved in multiplications. For example, for the term

$$\texttt{min\_speed} * \texttt{RIImin},$$

which appears in process `Sensor`'s local axiom clause, one of the two constants needs to be set. Otherwise, this local axiom is not in the proper form for a Presburger formula, since multiplications between two variables are not allowed.

3.3 Handling a Process Instance's Transition System

A process instance's transition system is characterized by a finite number of transitions. Each transition T_i contains an entry assertion, an exit assertion and a duration. If both the entry assertion and the exit assertion can be translated into Presburger formulas, then T_i can be easily translated into an Omega relation. For example, consider the transition

```
TRANSITION enter_R
  ENTRY           [ TIME : enter_dur ]
          ~train_in_R
  EXIT
          train_in_R = TRUE.
```

This transition can be translated as

$$\neg(ptrain\_in\_R = 1) \wedge train\_in\_R = 1 \wedge now = pnow + enter\_dur$$

where $ptrain\_in\_R$ and $pnow$ indicate the values before `enter_R` fires, while $train\_in\_R$ and now indicate those after `enter_R` fires. (Note that the variable $train\_in\_R$ itself has a Boolean type assumption

$$train\_in\_R = 1 \vee train\_in\_R = 0$$

already added into the assumption **Assump** of $\mathcal{T}$.)

3.4 Handling Clocks

Clocks in a Small-ASTRAL process instance include

- the global clock `now`,
- `Start(T)`, `End(T)` and `Call(T)` of a transition instance `T` indicating the start, end, and call times. `T` can be either a local transition or an imported transition,
- `Change(X)` indicating the change time of a variable `X`. `X` can be either a local variable or an imported variable.

These are all translated into integer variables. But for imported transitions and exported transitions, we need more constraints to satisfy the ASTRAL semantics. For instance, if `T` is an exported transition, `T` must be called by the external environment in order to fire. Calls on `T` are totally controlled by the external environment, which is typically restricted by a process's local environment clause. Successive calls are not effective if the called transition has not fired in response to the first call. In order to characterize the behavior of an exported transition, it is necessary to introduce a new variable to indicate the last call time. For example, in process `Sensor`, the exported transition `enter_R`'s last call time would be indicated as $enter\_R\_lcl$. Constraints also need to be added to the translated Omega relation of `enter_R`:

$$penter\_R\_lcl > penter\_R\_lst.$$

The constraint says that before the transition fires, the last call time of `enter_R` must be greater than the last start time of `enter_R`.

3.5 Handling Histories of Variables

A process instance may refer to the history of an imported variable or a local variable through the past operator `past(X,t)`. The history-dependent features make model-checking the process instance more difficult. An approach to handle the history of an imported variable is to encode it as a series of variables to indicate all its times of change and the values at each change. The maximal number of those changes is set to a pre-assigned constant *window-size*. For instance, under window-size 2,

$$\texttt{past(s.train\_in\_R,t)} = \texttt{true}$$

can be translated into

$$(t \geq s\_train\_in\_R\_lc0 \rightarrow 1 = s\_train\_in\_R\_v\_0)$$

$$\wedge(t < s\_train\_in\_R\_lc0 \wedge t \geq s\_train\_in\_R\_lc1 \rightarrow 0 = s\_train\_in\_R\_v\_1)$$

$$\wedge(t < s\_train\_in\_R\_lc1 \rightarrow 1 = s\_train\_in\_R\_v\_2)$$

where variables $s\_train\_in\_R\_lc0$ and $s\_train\_in\_R\_lc1$ are the last two change times (with $s\_train\_in\_R\_lc0$ being the most recent), and the variables $s\_train\_in\_R\_v\_0$, $s\_train\_in\_R\_v\_1$, $s\_train\_in\_R\_v\_2$ are the values before and after each change. Since the type of `s.train_in_R` is Boolean, $s\_train\_in\_R\_v\_1$ and $s\_train\_in\_R\_v\_2$ are redundant. Therefore, the above translation can be further simplified by substituting $s\_train\_in\_R\_v\_1$ and $s\_train\_in\_R\_v\_2$ with $1 - s\_train\_in\_R\_v\_0$ and $s\_train\_in\_R\_v\_0$, respectively.

3.6 Handling Assumptions and Properties

If the goal is to check the local invariant, then ASTRAL proof theory requires that the only assumptions that can be used are the local and global axiom clauses and the local initial clause. Therefore, these clauses are added to **Assump**. The property **Prop** is the invariant clause. If the goal is to check the local schedule, then the local and global axiom clauses, the local initial clause, the local imported variable clause and the local environment clause can all be used as assumptions. Therefore, these clauses are added to **Assump**, and in this case the property **Prop** is the schedule clause. It is important to note that the imported variable clause and the environment clause must hold during a transition's execution (not just at the times when it fires and ends). However, for a transition with duration greater than 1 the process's environment could be changed during the duration. Therefore, we must find a way to ensure that **Assump** holds consistently during a transition's execution. This will be discussed in the following subsections.

3.7 Strengthening a Transition Relation

A transition instance `T` in a Small-ASTRAL process instance is related to a transition T in the labeled transition system $\mathcal{T}$. We have already seen that by simply combining the entry assertion and the exit assertion of `T`, a transition relation T can be built. However, in order to obey the ASTRAL semantics, we need to further strengthen T. What we missed from the above translation of `T` is

- how the environment changes during an execution of `T`,
- how to ensure that **Assump** holds consistently during the execution.

The easiest way to do this is to constrain the environment from any changes during a transition's execution; i.e., the environment could change only at the end points of an execution. But our preliminary tests showed that although such a restriction is cheap, it is not effective in detecting bugs in a specification, since a lot of nontrivial errors can only be demonstrated when, for instance, an imported variable's value changes during the execution of a transition. Therefore, a more sophisticated class of assumptions are needed.

We propose two kinds of assumptions. One is called the *single-event* assumptions, the other is called the *multiple-event* assumptions. *Environment clocks* indicate the call time of a local exported transition and the start, end, or call times of an imported transition. By an event, we mean one of the followings behaviors:

- a change in the value of an imported variable,
- a change of an environment clock.

Remember that imported variables and environment clocks are all translated into integer variables. The single-event assumptions constrain the behaviors of the environment in such a way that during a transition's execution, each above mentioned integer variable changes at most once. In contrast, the multiple-event assumptions allow an unrestricted number of changes. Obviously, allowing multiple events during the execution makes the environment more accurate. However, this may cause the strengthened transition relation to become too complex to be calculated using the Omega Library.

Before we proceed to give the details of the single-event and multiple-event assumptions, some notation is needed. Let T be a transition relation in the labeled transition system $\mathcal{T}$ translated from a Small-ASTRAL process instance $\mathcal{P}$. Recall that $T \subseteq Q \times Q$ consists of pairs of states in Q. T itself is definable by a Presburger formula. Given a pair of states $\langle s, s' \rangle$ in T, we use $s(X)$ and $s'(X)$ to denote an imported variable X's values before and after the execution of the transition. Similarly, $s(C)$ and $s'(C)$ denote the values of an environment clock C. Clearly, $s(X) \neq s'(X)$ (or $s(C) \neq s'(C)$) implies an event changing X's value (or C's value) happens during the execution. T itself has a duration $dur(T)$. Even though in the definition of $\mathcal{T}$, no time was mentioned, this fact is implicitly expressed in the transition relation T as $s'(now) = s(now) + dur(T)$. But, since T is a binary relation, a pair $\langle s, s' \rangle$ in T only indicates that T may transit from s to s' without mentioning what has happened during the time interval of the duration $dur(T)$. In order to strengthen T such that **Assump** consistently holds in the interval, we must find an inexpensive way to recover the sequence of events that occurred. In the following subsections, we illustrate in detail the approach used.

3.8 Single-event Assumptions

The single-event assumptions constrain the environment such that each imported variable and environment clock changes at most once during a transition. There are four levels, which are different in the accuracy of approximating the environment.

The first level bets that each "intermediate" state $\hat{s}$ satisfies **Assump** when $\hat{s}$ is obtained by setting the global clock $s'(\mathit{now})$ in s' to any value between $s(\mathit{now})+1$ and $s'(\mathit{now})$. Notice that $s(\mathit{now}) = s'(\mathit{now}) - \mathit{dur}(T)$. Thus, this requirement is independent of the choice of s. Formally, denote an image

$$A_1 = \{s' : \forall t \ \ 1 + s'(\mathit{now}) - \mathit{dur}(T) \leq t \leq s'(\mathit{now}), s'[\mathit{now}/t] \in \textbf{Assump}\}$$

where $s'[\mathit{now}/t]$ denotes a state obtained from s' by replacing now with t. Thus, for each state s' in A_1, **Assump** consistently holds on each intermediate state when the clock now in s' is rolled back from $s'(\mathit{now})$ down to $1 + s'(\mathit{now}) - \mathit{dur}(T)$. It is easy to see that A_1 is definable by a Presburger formula, since **Assump** is a Presburger formula. In this level, T is strengthened as

$$T_{\text{level\_1}} = \{\langle s, s'\rangle \in T : s' \in A_1\}.$$

$T_{\text{level\_1}}$ is cheap to calculate, since we do not care when an event actually happened during the transition. However, it is not accurate. The reason is that the approach does not distinguish between the intermediate states before and after an event. Thus, although $T_{\text{level\_1}}$ is easy to calculate, this approach may cause false negatives. We will see how the model-checker handles them in the next section.

The second level allows at most one event to happen during the execution and checks **Assump** at the moment when that event occurs. More precisely, assume an environment clock C_0 changes during the execution; i.e., $s'(\mathit{now}) - s'(C_0) \leq \mathit{dur}(T) - 1$. Since this is the only event that occurs, for each environment clock $C \neq C_0$, $s'(\mathit{now}) - s'(C) \geq \mathit{dur}(T)$. $s'[\mathit{now}/s'(C_0)]$ represents the state that is rolled back from a state s' to the time $s'(C_0)$ when the clock C_0 changes. The image $A_2^{C_0}$ is constructed such that $s'[\mathit{now}/s'(C_0)]$ satisfies **Assump**. That is,

$$\begin{aligned} A_2^{C_0} = \{s' : s'(\mathit{now}) - s'(C_0) \leq \mathit{dur}(T) - 1 \\ \wedge \forall C \neq C_0 (s'(\mathit{now}) - s'(C) \geq \mathit{dur}(T)) \\ \wedge s'[\mathit{now}/s'(C_0)] \in \textbf{Assump}\}. \end{aligned}$$

The image A_2 is obtained by nondeterministically choosing which C_0 to change or having none of the environment clocks change. Thus, A_2 is defined by

$$A_2 = (\vee_{C_0} A_2^{C_0}) \vee \{s' : \forall C (s'(\mathit{now}) - s'(C) \geq \mathit{dur}(T))\}.$$

For this level, T is strengthened as

$$T_{\text{level\_2}} = \{\langle s, s'\rangle \in T : s' \in A_2\}.$$

Again, A_2 is independent of the starting state s of the transition. $T_{\text{level\_2}}$ is also cheap to calculate, but it is still not accurate. The reason is that $s'[now/s'(C_0)]$ is not exactly the state when C_0 changes since the values of the local variables and local clocks in $s'[now/s'(C_0)]$ are equal to those in s', i.e., values after the transition T's execution. In fact, according to the ASTRAL semantics, the values of local variables and clocks should have the same values as in the state s when the transition started. Another source of inaccuracy is from the fact that in A_2 we do not check that **Assump** holds before and after the moment when C_0 changes. Thus, as was the case for the approach of level 1, false negatives may occur. In order to faithfully rebuild the intermediate states, we have to take the starting state s into account as follows.

The third level also allows at most one event during the execution, however, it checks **Assump** at the point when the event happens, as well as during the times before and after the event. Given a pair $\langle s, s' \rangle \in T$, the state $s[now/s'(C_0)]$ is exactly the state of $\mathcal{P}$ when C_0 changes, i.e., $s'(C_0) > s(now)$, during the transition. Since the change of C_0 is the only event, we have for each environment clock $C \neq C_0$, $s'(C) = s(C)$. Denote

$$A_3^{C_0} = \{\langle s, s' \rangle : s'(C_0) > s(now)$$

$$\wedge \forall C \neq C_0 (s'(C) = s(C))$$

$$\wedge s[now/s'(C_0)] \in \mathbf{Assump}$$

$$\wedge \forall t \quad s(now) < t < s'(C_0)(s[now/t] \in \mathbf{Assump})$$

$$\wedge \forall t \quad s'(C_0) < t < s'(now)(s[now/t, C_0/s'(C_0)] \in \mathbf{Assump})\}.$$

Intuitively $A_3^{C_0}$ is the set of all the state pairs $\langle s, s' \rangle$ such that

- C_0 changes during the state transition from s to s',
- no other environment clock changes,
- at the change point $s'(now)$, **Assump** is satisfied by the intermediate state $s[now/s'(C_0)]$,
- before the change happens, i.e., for all t with $s(now) < t < s'(C_0)$, **Assump** is satisfied,
- after the change happens, i.e., for all t with $s'(C_0) < t < s'(now)$, **Assump** is satisfied. Notice that, in this case, the state at time t is characterized by $s[now/t, C_0/s'(C_0)]$, obtained by substituting the global clock now with t, and substituting the environment clock C_0 with $s'(C_0)$. It is incorrect if we use $s'[now/t]$ since the local variable values in s' are not the same as those during the execution.

It is obvious that $A_3^{C_0}$ is definable by a Presburger formula. Also notice that $A_3^{C_0}$ does not depend on the choice of a transition T. There is a case when no C_0 changes, but **Assump** consistently holds during a transition's execution. This can be expressed by

$$A_3^{\emptyset} = \{\langle s, s'\rangle : \forall C(s(C) = s'(C))$$

$$\wedge \forall t \quad s(now) < t < s'(now)(s[now/t] \in \mathbf{Assump})\}.$$

Allowing one to choose nondeterministically which clock C_0 to change and also including the case when no C_0 changes, we define

$$A_3 = (\vee_{C_0} A_3^{C_0}) \vee A_3^{\emptyset}.$$

Even though A_3 is independent of any choice of a transition, we can "tie" it to a specific transition T. That is, T can be strengthened by

$$T_{\text{level\_3}} = T \wedge A_3.$$

The resulting transition relation $T_{\text{level\_3}}$ guarantees that **Assump** is satisfied during T's execution.

The fourth level allows each environment clock to change at most once during the execution. To express the fact that **Assump** consistently holds during the execution, we enumerate all the subsets $\mathcal{C}$ of the environment clocks. For each subset C, we enumerate all the permutations $\tau_{\mathcal{C}}$. A permutation gives an ordering of the change times of clocks in $\mathcal{C}$ during the execution. Similar to the level 3 approach, the fact that **Assump** holds during an execution can be expressed as a set of state pairs $A_4^{\tau_{\mathcal{C}}}$ that is definable by a Presburger formula. We omit the technical details here. Finally, combining all the possible choices of the subsets $\mathcal{C}$ and the permutations, we define

$$A_4 = \vee_{\tau_{\mathcal{C}}} A_4^{\tau_{\mathcal{C}}}.$$

Again, A_4 is independent of any choice of a transition. For this level, T can be strengthened by

$$T_{\text{level\_4}} = T \wedge A_4.$$

The resulting transition relation $T_{\text{level\_4}}$ ensures that **Assump** consistently holds during T's execution, when each environment clock changes at most once.

3.9 Multiple-event Assumptions

An environment clock could change multiple times during a transition's execution. But the fact that **Assump** holds during the execution in general is not expressible in a Presburger formula. That is, there is no way to strengthen T as in the last subsection to ensure this fact. In order to allow multiple events, a transition relation T is refined by splitting one big move of duration $dur(T)$ into $dur(T)$ moves of unit duration. To do this, a duration counter i is used to indicate how many unit duration moves have been taken. There are three kinds of *micro-moves*. The first kind T_1^Δ is to start the transition T. Each tuple $\langle i, s, i', s' \rangle$ in T_1^Δ satisfies:

- $i = 0$ and $i' = 1$,
- T_1^Δ has unit duration. That is, $s'(now) = s(now) + 1$,
- s satisfies the enabling condition of T,
- other components of s' are exactly the same as s except that each environment clock C may change (i.e., $s'(C) = s'(now)$). Of course, when C is a change clock of an imported variable, a change of the value of the variable implies a change of C.

The second kind T_2^Δ represents all the intermediate micro-moves before the duration is reached. Each tuple $\langle i, s, i', s' \rangle$ in T_2^Δ satisfies:

- $i' = i + 1 \wedge i' < dur(T)$,
- T_2^Δ has unit duration. That is, $s'(now) = s(now) + 1$,
- other components of s' are exactly the same as s except that each environment clock C may change.

The third kind T_3^Δ leads to the end of the transition T. Each tuple $\langle i, s, i', s' \rangle$ in T_3^Δ satisfies:

- $i = dur(T) - 1 \wedge i' = 0$ (reset i to zero),
- T_3^Δ has unit duration. That is, $s'(now) = s(now) + 1$,
- $\langle s, s' \rangle$ satisfies the exit assertion.
- each environment clock C may change.

T is simply replaced by a nondeterministic composition

$$T_1^\Delta \vee T_2^\Delta \vee T_3^\Delta .$$

From the above construction, after the transition T_1^Δ is taken, T_2^Δ is the only successive transition. T_2^Δ loops for $dur(T) - 1$ times and then triggers T_3^Δ. Thus, the refined sequence of micro-moves of T is unique. We do not claim that T is equivalent to the composition; in fact, it is not. However, we shall see later that when model-checking $\mathcal{P}$ with the multiple-event assumptions, using this composition instead of T is sufficient.

4 Symbolic Search Algorithms

The previous section shows that a Small-ASTRAL process instance $\mathcal{P}$ can be translated into a labeled transition system $\mathcal{T}$. Using either the single-event assumptions or the multiple-event assumptions, a transition T in $\mathcal{T}$ can be strengthened. The model-checker calculates the image of reachable states of $\mathcal{T}$, starting from the initial image **Init**. A naive way to do the model-checking is to calculate the Omega representation of a one_step transition defined by

$$T_{\text{one}-\text{step}} = \bigvee_i T_i$$

of the process instance by combining all the transition relations T_i. Next, calculate the iterations

$$\begin{array}{rcl} R_0 & = & \mathbf{Init} \wedge \mathbf{Assump} \\ R_1 & = & Post_{T_{\text{one}-\text{step}}}(R_0) \wedge \mathbf{Assump} \\ \vdots & & \\ R_n & = & Post_{T_{\text{one}-\text{step}}}(R_{n-1}) \wedge \mathbf{Assump} \\ \vdots & & \end{array}$$

and check for each R_n, whether

$$R_n \subseteq \mathbf{Prop}.$$

Our experiments showed, however, that this intuitive solution does not always work for a nontrivial ASTRAL specification, such as the railroad crossing benchmark. The reason is that $T_{\text{one}-\text{step}}$ is so large that its image can not be built using the Omega library, even when allowed to run for hours. A number of more sophisticated solutions are proposed in the following sections. The central idea is how one can make the image computations feasible when each T_i has a large image.

4.1 Execution Graphs

In order to achieve better efficiency, the model-checker operates on the execution graph G of a process instance (represented as a labeled transition system $\mathcal{T}$). The graph G is a pair

$$\langle V_G, R_G \rangle$$

in which $V_G = \{initial, idle, T_1, \cdots, T_k\}$. *initial* indicates the initial transition, which is defined as an identity transition on the initial states with zero duration. *idle* is a newly introduced transition, which has duration one. *idle* fires if every T_i is not firable. *idle* will not change the values of local variables. $R_G \subseteq V_G \times V_G$ excludes all the pairs of transitions such that the second one is not immediately firable after the first one finishes. G is automatically constructed by analyzing the initial conditions and checking if entry/exit pairs for different transitions are satisfiable using the Omega library[3]. Figure 1 is an example of the execution graph of the `Gate` process. A dashed arrow in the figure means zero or more *idle* transitions are executed to reach the next node.

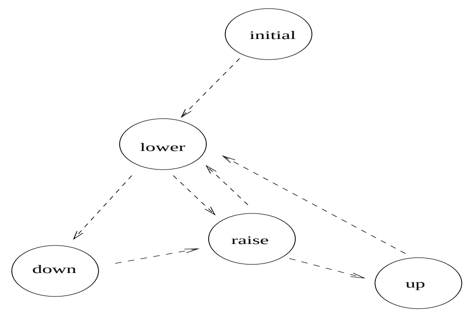

Figure 1: The execution graph of `Gate`

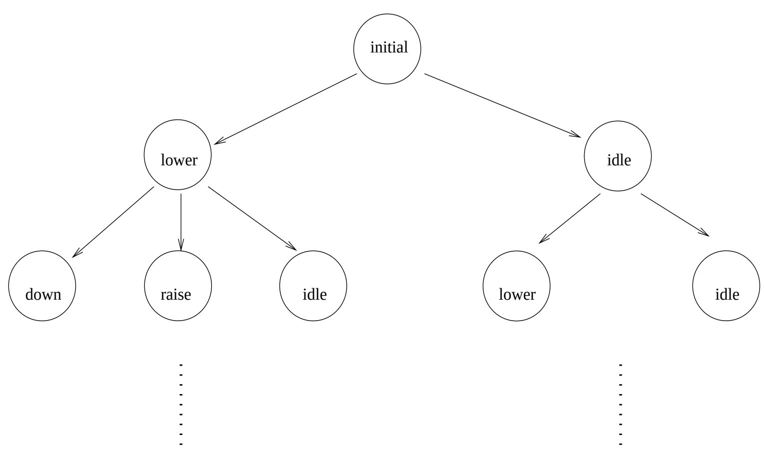

Figure 2: The execution tree of `Gate`

The execution tree of G is the the tree of all possible execution paths trimmed by the execution graph G. Figure 2 is a part of the execution tree of the process `Gate`. Since, as will be discussed later, the use of the model-checker in the ASTRAL SDE is only for debugging purposes rather than full verification, there is no interest in calculating the least

[3]Note that the ASTRAL SDE has a tool called the sequence generator [46], which does this by using a theorem proving approach. The SDE sequence generator is also fully automated.

fixed point for the transition system. Therefore, a user needs to set the depth of the tree indicating the maximal number of iterations of transitions to check. Once this depth is set, the model-checker proceeds with the image computations on G by using the following algorithms.

4.2 Depth-first Search under the Single-event Assumptions

In this subsection, as well as the following two subsections, we assume each transition in $\mathcal{P}$ is strengthened by using the single-event assumptions. The depth-first search algorithm calculates the reachable image along each execution path in a depth-first order.

Figure 3 shows the recursive procedure. In the algorithm, $Post_A$, which was defined earlier, is the post image operator for the transition indicated by node A. Model-checking a node A starts by calculating the preimage and postimage of it. If the postimage is not empty, which means that the transition indicated by A is firable, then the preimage is checked with respect to the property, followed by checking every child node according to the execution graph. The model-checking procedure starts from the initial node *initial*,

$$\text{Check\_dfs}(initial, depth).$$

Boolean Check_dfs(Node A, int *depth*)
{
    if $A.layer = depth$ then return true;
    if $A.layer = 0$ then
        $A.postimage = \mathbf{Init} \wedge \mathbf{Assump}$;
    else
        $A.preimage = (A.parent).postimage$;
        $A.postimage = Post_A(A.preimage) \wedge \mathbf{Assump}$;
    if $A.postimage \neq \emptyset$ then
        if($A.preimage \not\subseteq \mathbf{Prop}$) then return false;
        else foreach B, $\langle A, B\rangle \in R_G$
            if($\neg$Check_dfs($B, depth$)) return false;
    return true;
}

Figure 3: The depth-first search algorithm

4.3 Breadth-first Search under the Single-event Assumptions

Depth-first search may explore too many nodes before an error is detected or before all the nodes are traversed without finding any errors. Theoretically, the number of reachable nodes up to the pre-assigned depth is exponential in depth. Thus, it is natural to search in a breadth-first way. That is, all the nodes in the same layer are combined into one bigger "node". Thus, we only need to do at most *depth* number of iterations.

Figure 4 shows the algorithm. At each layer, the model-checker collects the union of all the postimages for each node A, i.e.,

$$postimage = \vee_A Post_A(preimage).$$

This postimage is used to check against the property, and propagate to the next layer to carry out the breadth-first search. The model-checker starts from Check_bfs($\mathbf{Init} \wedge \mathbf{Assump}, 0$).

Boolean Check_bfs(Image *preimage*,int *layer*)
{
    if $preimage = \emptyset$ then return true;
    if $layer = depth$ then return true;
    $postimage = \vee_A Post_A(preimage) \wedge \mathbf{Assump}$;
    if $postimage \not\subseteq \mathbf{Prop}$ then return false;
    if $\neg$Check_bfs($postimage, layer + 1$) then return false;
    return true;
}

Figure 4: The breadth-first search algorithm

4.4 Depth-Breadth Search under the Single-event Assumptions

The breadth-first search algorithm works well for simple specifications. The reason is that the reachable image for each layer is not large and the image computations are affordable. In contrast, for complex specifications, this algorithm does not work since the size of the image for each layer grows dramatically fast. In that case, the depth-first search would be a better choice, since along an execution path, the reachable image usually is not large. But the load (i.e., the size of the reachable image) is not evenly distributed among paths. Obviously, it is wasteful to adopt depth-first search along a number of paths with extremely small load. Thus, we need a dynamic approach to direct the model-checker to carry out either depth-first search or breadth-first search. The algorithm is called depth-breadth search as shown in Figure 5.

In the algorithm, the procedure Check_dbfs dynamically chooses a search approach. When the size of the reachable image $preimage.Size()$ is greater than a pre-assigned number IMAGE_SIZE, it calls the depth-first search procedure Check_dfs. Otherwise, it calls the breadth-first procedure Check_bfs. Both procedures are adapted from the previous algorithms. It should be noted that a node A could be $NULL$, which means it is a "node"

```
Boolean Check_dbfs(Image preimage,int layer, Node A)
{
    preimage = preimage ∧ Assump;
    if preimage ⊈ Prop then return false;
    if layer = depth then return true;
    if preimage.Size() > IMAGE_SIZE then
        return Check_dfs(preimage,layer,A);
    else
        return Check_bfs(preimage,layer,A);
}
Boolean Check_bfs(Image preimage,int layer, Node A)
{
    Image postimage = ∨_B Post_B(preimage);
    if ¬Check_dbfs(postimage, layer + 1, NULL) then return false;
    return true;
}
Boolean Check_dfs(Image preimage,int layer, Node A)
{
    foreach B, ⟨A, B⟩ ∈ R_G
            Image postimage = Post_B(preimage);
            if ¬Check_dbfs(postimage, layer + 1, B) then return false;
    return true;
}
```

Figure 5: The depth-breadth search algorithm

after a round of breadth-first search. Thus, in Figure 5, when $A = NULL$, $\langle A, B\rangle \in R_G$ is always true, since after a breadth-first search, every branch in the graph could be taken. The depth-breadth search starts from the initial node and layer 0 with the initial image, i.e., Check_dbfs(**Init**, $0, initial$).

4.5 Notes on the Algorithms

Recall that under single-event assumptions, a transition can be strengthened at four levels. For a given transition T, the strengthened transition at level 1 and level 2 can be expressed as

$$T_{\text{level\_1}} = \{\langle s, s'\rangle \in T : s' \in A_1\}$$

and

$$T_{\text{level\_2}} = \{\langle s, s'\rangle \in T : s' \in A_2\},$$

respectively. Each node, excluding the *idle* and *initial* nodes, corresponds to a strengthened transition. In the above algorithms, the image calculations

$$Post_{T_{\text{level\_1}}}(preimage)$$

and

$$Post_{T_{\text{level\_2}}}(preimage)$$

for a node representing $T_{\text{level\_1}}$ or $T_{\text{level\_2}}$ are directly applied on *preimage* and the strengthened transition. In fact, a more efficient way, which is used in the actual implementation, is to calculate the intersections

$$Post_T(preimage) \wedge A_1$$

and

$$Post_T(preimage) \wedge A_2,$$

respectively. The reason is that constructing the transition relation $T_{\text{level\_1}}$ and $T_{\text{level\_2}}$ by restricting the range of T to A_1 and A_2 respectively is very expensive, when A_1 and A_2 are large.

For the strengthened transition at level 3 and level 4, we have

$$T_{\text{level\_3}} = T \wedge A_3$$

and

$$T_{\text{level\_4}} = T \wedge A_4.$$

Instead of calculating the intersection of T and A_3 (or A_4), it would be more efficient if the postimage calculations

$$Post_{T_{\text{level\_3}}}(preimage)$$

and

$$Post_{T_{\text{level\_4}}}(preimage)$$

can be split into

$$Post_T(preimage) \wedge Post_{A_3}(preimage)$$

and

$$Post_T(preimage) \wedge Post_{A_4}(preimage),$$

respectively. However, this is generally not true. Thus, if a user decides to use the approach with level 3 or level 4, the model-checker has to calculate the strengthened transition relation directly by intersecting T with A_3 (or A_4). This calculation is expensive, as we show in section 6, which discusses the experiments.

On finding a violation, the model checker prompts a trace of concrete states that leads to the error. Each state indicates the values of all the clocks and variables in the specification. As we pointed out before, strengthening transitions may cause false negatives. Thus, the model checker checks whether the found error is real or not by recovering each intermediate concrete state along the trace and checking **Assump** against the intermediate states. If the error is not real, the model-checker continues to search the next error.

4.6 Breadth-first Search under the Multiple-event Assumptions

Recall that transitions are split into a number of micro-moves under the multiple-event assumptions. The model-checking procedure proceeds by iterating a big micro-move by combining all the micro-moves. Thus, we only consider the breadth-first search strategy for multiple-event assumptions.

Under multiple-event assumptions, a transition T, excluding the *idle* and *initial* transitions, is refined into a sequence of micro-moves T_1^Δ, T_2^Δ and T_3^Δ as shown above. The labeled transition system $\mathcal{T}$ can be treated as a system with only one "big" transition

$$\bigvee_T T.$$

Iterations of the big transition demonstrate all the behaviors of $\mathcal{T}$. Recall that the construction of the micro-moves T_1^Δ, T_2^Δ and T_3^Δ of T ensures that the sequence of micro-moves uniquely corresponds to an execution of T. Thus, the big transition can be replaced by a big micro-move (i.e., a move with at most unit duration) as follows

$$\bigvee_{T \neq idle, initial} T_1^\Delta \vee T_2^\Delta \vee T_3^\Delta \bigvee initial \vee idle.$$

Thus, iterations of the big micro-move demonstrate refined traces of $\mathcal{T}$ with states at each time point presented. Thus, $\mathcal{T}$ is refined as a new labeled transition system with transitions

$$\cup_{T \neq idle, initial} \{T_1^\Delta, T_2^\Delta, T_3^\Delta\} \cup \{initial, idle\}.$$

The breadth-first search algorithm is exactly the same as the one under single-event assumptions, but for the new labeled transition system. [4]

5 Approximation Techniques

The symbolic model-checker presented in the last section is not feasible for large specifications due to high complexity. From our experiments, which we will discuss further later in this paper, the high complexity of a single process instance can come from two sources: the local and global constants used in the instance and the local and imported variables that constitute the variable portion of the process instance. For example, in the `Gate` process there are 10 global constants and 6 local constants. These constants are used to parameterize the specified system, e.g., to specify a system containing a parameterized number of process instances as well as a system containing parameterized timing requirements. The local variables contribute to the local state and are changed by executing transitions. Though each process is modularized, each process instance does not stand alone. A process instance may also interact with other process instances through imported variables that are exported from the other process instances. Since a process's local properties are proved using only its local assumptions, the process instance must specify assumptions (environment clause and imported variable clause) that are strong enough to correctly characterize the environment and the behaviors of the imported variables. In order to guarantee the local properties, it is not unusual for an assumption to include complex timing requirements on the `call` patterns and the imported variables' change patterns. Thus, the second source of complexity primarily comes from history-dependency.

When it is not practical for the symbolic model-checker to complete the search procedure for a complex process instance, it is desirable to define approximation approaches to speed up the procedure by sacrificing coverage. Based upon the above analysis, four approaches can be used. The first approach is to assign concrete values to some of the constants before using the model checker. In [27], it was shown that doing this will speed up the model checker and that it is still effective in finding bugs in some cases. There are, however, reasons for not using this approach. Most importantly, picking the right set of constant values to cause "interesting" things (especially potential errors) to happen is not trivial. Some choices will miss scenarios in which the specification would fail. In addition, even with a number of the constant values fixed, the model checking process is still expensive in some cases due to the complexity of the behavior of the local and imported variables. Experience shows that this approach, as well as using the explicit state model checker [26], should be used in

[4] We could consider depth-first search and depth-breadth search for the new transition system. But given the fact that the number of micro-moves (i.e., refined transitions) is large (triple the number of transitions in the original process instance), the execution tree of the new transition system is too expensive to explore. Therefore, these two search strategies have not been implemented.

the earlier stages of debugging a specification, when errors are relatively easy to catch. The three remaining approaches, which are discussed in more detail in the following subsections, speed up the model checker by enforcing it to check either fewer nodes or smaller nodes. These approaches free the user from setting up constant values. Three different techniques are proposed. A random walk technique is used to allow the model-checker to randomly skip a number of branches when traversing the execution tree. A partial image technique considers only a subset of the image and uses this subset to calculate the postimage at each node. The dynamic environment generation technique generates different sequences of imported variable values for different execution paths. These three techniques are discussed in more detail in the following subsections.

5.1 Random walk

A path in the execution tree of an ASTRAL process is a sequence of transitions. Each node in the tree contains the image of all reachable states from the initial node along the path. Theoretically, the number of paths is exponential to the user-assigned search depth. Even though the symbolic model-checker itself adopts a number of trimming techniques, the time for a complete search for a large specification is unaffordable. It is our experience that, when a specification has a bug, this bug can usually be demonstrated by many different paths. The reasons are (1) The ordering of some transitions can be switched without affecting the result (though practically it is hard to detect this, due to history-dependency.[5]), (2) Most specifications contain a number of parameterized constants. When a specification has a bug, usually there are numerous scenarios and choices of parameterized constant values that reveal it, so these scenarios can be shown by many different paths.

Random walk is an approximation technique of searching only a portion of the reachable nodes on the execution tree. Though the technique is applicable for all four algorithms proposed in the previous section, without loss of generality, we again illustrate the technique for the depth-breadth search algorithm in Figure 5. Figure 6 shows the recursive procedure which is similar to Figure 5 except that this algorithm includes a random choice (either skip or not) *toss* when the model-checker moves from one node to its children. The statement

$$postimage = \vee_{B, toss(A, layer) = tail} Post_B(preimage)$$

in the procedure Check_bfs in Figure 6 only selects those nodes B with the result of random tests $toss(A, layer)$ being $tail$. For the depth-first search procedure Check_dfs in the same figure, the statement

"foreach B, $\langle A, B\rangle \in R_G$ and $toss(A, layer) = tail$"

[5]This is significantly different from some standard techniques used in finite state model-checking, such as the partial order method [38]

makes the model-checker skip the branches to B when the random test fails. The random Boolean function $toss(A, layer)$ is not symmetric. The probability of result *tail* is chosen as

$$(1 - \frac{layer}{depth}) + \frac{layer}{depth} \cdot \frac{1}{numChildren(A)}$$

where $numChildren(A)$ indicates the number of successors of node A in the execution graph G, i.e., $numChildren(A) = |\{D : \langle A, D\rangle \in R_G\}|$,[6] and *layer* indicates the layer that the model-checker is currently checking. The reason for this choice is to ensure the following:

- A short violation has less chance of being missed. When *layer* is small, the probability of result *tail* is large. When *layer* is large, if $numChildren(A)$ is greater than 1, then the probability is small. Hence a longer path has a higher probability of being skipped.
- When $numChildren(A)$ is 1 [7], the probability of result *tail* is 1. That is, a node with only one successor can not be skipped.

5.2 Partial image

In the Omega library, each image is represented by a union of convex linear constraints. The efficiency of an image calculation depends upon the number of variables and the number of constraints. Experience shows that, when a specification has a bug, there are usually numerous sets of parameterized constant and variable values that lead to the bug. These values usually satisfy many constraints in an image. Thus, considering only a part of the image will usually still let the model-checker find the bug. As reported in [27], fixing a number of parameterized constant values increases the speed of the model-checker, since the number of variables in the image is decreased. This is a special case of the partial image technique. However, finding the right set of constant values leading to a potential bug is not easy for complex specifications; it usually requires a user that thoroughly understands the specification. The partial image technique presented here is used without fixing any constant values, by applying the $PartialImage()$ operator, which returns only half (randomly chosen) of the unions for the image, when the size of the image exceeds the pre-assigned "tolerance" IMAGE_SIZE. Again, the technique is applicable for all four algorithms proposed in the previous section. Without loss of generality, we illustrate the technique for the depth-breadth search algorithm in Figure 5. The algorithm, as shown in Figure 7, is essentially the same as Figure 5 except that the $PartialImage()$ operator is applied on each preimage when the procedure Check_dbfs is invoked. This reduced preimage is then used to calculate the postimage, as stated in the algorithm.

[6] Recall that when $A = NULL$, i.e., a breadth-first search is chosen, $\langle A, D\rangle \in R_G$ is always true.

[7] $numChildren(A)$ is always at least one, since each node has a successor through the *idle* transition.

```
Boolean Check_dbfs(Image preimage,int layer, Node A)
{
    preimage = preimage ∧ Assump;
    if preimage ⊈ Prop then return false;
    if layer = depth then return true;
    if preimage.Size() > IMAGE_SIZE then
        return Check_dfs(preimage,layer,A);
    else
        return Check_bfs(preimage,layer,A);
}
Boolean Check_bfs(Image preimage,int layer, Node A)
{
    Image postimage = ∨_{B,toss(A,layer)=tail} Post_B(preimage);
    if ¬Check_dbfs(postimage, layer + 1, NULL) then return false;
    return true;
}
Boolean Check_dfs(Image preimage,int layer, Node A)
{
    foreach B, ⟨A, B⟩ ∈ R_G and toss(A, layer) = tail
            Image postimage = Post_B(preimage);
            if ¬Check_dbfs(postimage, layer + 1, B) then return false;
    return true;
}
```

Figure 6: The depth-breadth search algorithm with random walk

```
Boolean Check_dbfs(Image preimage,int layer, Node A)
{
    preimage = PartialImage(preimage);
    preimage = preimage ∧ Assump;
    if preimage ⊈ Prop then return false;
    if layer = depth then return true;
    if preimage.Size() > IMAGE_SIZE then
        return Check_dfs(preimage,layer,A);
    else
        return Check_bfs(preimage,layer,A);
}
Boolean Check_bfs(Image preimage,int layer, Node A)
{
    Image postimage = ∨_B Post_B(preimage);
    if ¬Check_dbfs(postimage, layer + 1, NULL) then return false;
    return true;
}
Boolean Check_dfs(Image preimage,int layer, Node A)
{
    foreach B, ⟨A, B⟩ ∈ R_G
            Image postimage = Post_B(preimage);
            if ¬Check_dbfs(postimage, layer + 1, B) then return false;
    return true;
}
```

Figure 7: The depth-breadth search algorithm with partial image

5.3 Dynamic environment generation

This approach reduces the number of variables involved in the image computations. Imported variables characterize the interface between the instance $\mathcal{P}$ and other process instances in the system. The environment of $\mathcal{P}$ is composed of the following elements,

- `Call`s of the transitions exported by $\mathcal{P}$,
- `Start`s, `End`s and `Call`s of the imported transitions,
- the imported variables and their histories.

Except for the imported variables, each element needs only one variable to represent its value. Thus, the imported variables are essentially the bottleneck of the symbolic model-checker. We focus on how to approximate the imported variable part of the environment, and for the remainder of this section *environment* refers to the imported variable part. For other parts of the environment the technique can be used similarly. The approach used is to effectively generate a *reasonable* example of the environment for each execution path. Before presenting the algorithm it is necessary to define some notation. Let *preimage* be an image representing the current reachable states. If all the imported variables and their histories have concrete values in *preimage*, *preimage* is called *having a concrete environment.* By using the `example` operator in the Omega library, a sample, in which each variable has a concrete value, can be picked from *preimage*. The image *preimage* can be made to have a concrete environment by replacing all the imported variables and their histories with the concrete values in the sample. The resulting image is denoted by $ConcreteEnv(preimage)$. In the same manner as for the partial image technique, the operator will not apply if the size of the preimage is less than the pre-assigned IMAGE_SIZE. Again, the technique is applicable to all four algorithms proposed in the previous section. Without loss of generality, we illustrate the technique for the depth-breadth search algorithm in Figure 5. The algorithm, as shown in Figure 8, is essentially the same as Figure 5 except that the $ConcreteEnv()$ operator is applied on each preimage when the procedure Check_dbfs is invoked. This reduced preimage is then used to calculate the postimage, as stated in the algorithm.

It should be noted that in the algorithm, whether or not to apply the $ConcreteEnv()$ operator depends upon the random test $toss(A, layer)$, which was defined in a previous section. As discussed for random walk, the random tests will ensure a reasonably large coverage of the execution paths, which is needed to detect a nontrivial bug in a specification.

6 Experimental Results

Experience shows that real-time specifications are hard to write and to read, especially when they involve complex timing constraints. A user can mutate a part of the specification where

Boolean Check_dbfs(Image $preimage$,int $layer$, Node A)
{
    if $toss(A, layer) = tail$ then $preimage = ConcreteEnv(preimage)$;
    $preimage = preimage \wedge$ **Assump**;
    if $preimage \not\subseteq$ **Prop** then return false;
    if $layer = depth$ then return true;
    if $preimage.Size() >$ IMAGE_SIZE then
        return Check_dfs($preimage$,$layer$,A);
    else
        return Check_bfs($preimage$,$layer$,A);
}
Boolean Check_bfs(Image $preimage$,int $layer$, Node A)
{
    Image $postimage = \vee_B Post_B(preimage)$;
    if $\neg$Check_dbfs($postimage, layer + 1, NULL$) then return false;
    return true;
}
Boolean Check_dfs(Image $preimage$,int $layer$, Node A)
{
    foreach B, $\langle A, B\rangle \in R_G$
        Image $postimage = Post_B(preimage)$;
        if $\neg$Check_dbfs($postimage, layer + 1, B$) then return false;
    return true;
}

Figure 8: The depth-breadth search algorithm with dynamic environment generation

he or she believes that such a change should affect the behavior of the system. A similar technique, called mutation analysis [29, 43], has been used in program testing for many years. The technique makes a minor change to a program, then shows the effectiveness of test data by checking if an error is revealed. In our research mutation testing is applied to specifications instead of programs. Mutation tests on a specification can help a user understand the specification, and they can test the strength of the specification. If the mutant is killed (i.e., a violation is found), then a specification level violation trace is demonstrated. Reading through the trace helps the user to quickly figure out where and how the syntax change affects the specification. If a mutation is created by weakening an assumption in the specification and the model-checker fails to find any violations, then a potential weakness is demonstrated in the original specification. There are two possibilities in this case. One is that the model-checker is not able to find the bug under this specific run with the specific setup. The other is that the mutation is equivalent to the original (correct) specification.

Since the use of the symbolic model-checker in the ASTRAL SDE is only for debugging purposes, its effectiveness for detecting a potential error in a specification is the major concern. The model-checker has been run on ten mutations of the `Gate` process from the railroad crossing specification. The reason that the `Gate` process specification was used is that it contains imported variables as well as their histories. These imported variables result in a large instance of the `Gate` process for which the symbolic model-checker failed to complete when not using approximation techniques. Each mutation contains a minor change to the original specification[8]. A detailed list of all the mutations is presented in Table 1.

For all of the tests, the constants `min_speed` and `max_speed` were set to 15 and 20, respectively, the constant `n_tracks` was set to 2, and the window size was chosen as 3. There were no other user-assigned constants. This setting demonstrates the effectiveness of the model-checker on a large instance. All tests were performed on a Sun workstation with 4 CPUs and with 256M main memory and 512M swap memory. CPU time is measured in seconds and excludes the preprocessing time (normally 200 to 600 seconds, independent of whether an approximation technique is applied or not). Our patience is set to 2000 CPU seconds. If the model-checker fails to complete the search within this limit, it will abort.

The design of the test cases tries to answer the following questions:

- Without using the approximation techniques, how well does each of the search algorithms perform in debugging the same specification. The algorithms include the three algorithms (depth-first search, breadth-first search and depth-breadth search) under the single-event assumptions with four different levels, and the breadth-first search

[8]The unmutated railroad crossing specification has been proved to be correct using the ASTRAL theorem prover, which is also part of the ASTRAL SDE [47].

M1	delete `raise_time` from the 1st conjunction of the axiom of `GATE`
M2	replace `lower_dur` with `raise_dur` in the 3rd conjunction of the axiom of `GATE`
M3	delete the 3rd conjunction from the axiom of `GATE`
M4	delete `up_dur` from the 3rd conjunction of the axiom of `GATE`
M5	delete `response_time` from the 3rd conjunction of the axiom of `GATE`
M6	replace `response_time` with `lower_time` in the 1st conjunction of the schedule of `GATE`
M7	delete `raise_dur` from the 2nd conjunction of the axiom of `GATE`
M8	delete `now-End(lower)>=lower_time` from the entry assertion of transition `down`
M9	delete `now-End(raise)>=raise_time` from the entry assertion of transition `up`
M10	delete `position=raising` from the entry assertion of transition `raise`

Table 1: Ten mutations of the railroad crossing specification

algorithm under multiple-event assumptions.

- Using the approximation techniques, how much does the performance of each of the search algorithms improve in debugging a specification. The approximations are the three approximation techniques used separately and in combination.

Obviously, given the number of possible combinations of the choice of algorithms as well as the 10 mutations, it is not practical to run through all the tests. We decided to run the following two sets of tests. First, we ran M1 using each of the search algorithms without using the approximation techniques. This set of tests indicates how well the algorithms perform in debugging the same specification. Next, we ran all ten mutations for the depth-first search algorithm by using the approximation techniques and their combinations. This set of tests indicates the effectiveness of the model-checker in debugging a specification when using the approximation techniques and their combinations. The details of the two sets of tests are as follows.

- Without using the approximation techniques, run M1 using each of the search algorithms:
  - df_single_1 (depth-first search under the single-event assumptions with level 1),
  - bf_single_1 (breadth-first search under the single-event assumptions with level 1),
  - db_single_1 (depth-breadth search under the single-event assumptions with level 1),

- bf_multiple (breadth-first search under the multiple-event assumptions).

Due to the randomness in a depth-first search where the model-checker chooses the order of branches to take, the first two algorithms df_single_1 and bf_single_1 are run three times. After this round of tests, we should have a good idea whether choosing a different search approach will greatly affect the performance for the model-checker in debugging the specification. Notice that under the single-event assumptions, level 2 is very similar to level 1 so it is not selected. Also levels 3 and 4 are not selected, since, as mentioned before, the model-checker can not complete the preprocessing in the time-limit, due to the fact that the strengthened transition relations are extremely expensive to compute. When using the approximation techniques, we would also like to know whether they work for different algorithms. We chose to use the partial image technique on M1 for all of the above algorithms.

- Finally, and most importantly, we would like to know the effectiveness of the model-checker in detecting a bug in a specification with and without using the approximation techniques. We thoroughly run through all ten mutations for the depth-first search algorithms under the single-event assumptions, i.e., df_single_1 and df_single_2 (of level 2), with all combinations of the approximation techniques:
  - `plain` (without using the approximation techniques),
  - `r.w.` (random walk)
  - `p.i.` (partial image)
  - `d.e.` (dynamic environment generation)
  - `r.w. + p.i.`
  - `r.w. + d.e.`
  - `p.i. + d.e.`
  - `r.w. + p.i. + d.e.`

  Again, due to the randomness in a depth-first search where the model-checker chooses the order of branches to take, each test is run three times.

6.1 Choosing Among Various Algorithms

The tests in this subsection will clarify whether a choice of the search algorithms, i.e., df_single_1, bf_single_1, db_single_1 and bf_multiple will greatly influence the performance of the model-checker in detecting an error. We decided to choose M1 to run through all the algorithms three times and the results are shown in Table 2. Each test result is written as a pair. For instance, (429, $\times$) means the model-checker found a violation in 429 seconds. The status values are "$\times$" (the model-checker is able to detect a violation), "$\surd$"(the model-checker completes the search within the time-limit and reports no error), and "$\uparrow$"(the model-checker fails to finish within the time-limit). The time-limit, as mentioned before, is set to be 2000 CPU seconds. The search depth is 10 for df_single_1, bf_single_1 and db_single_1. Since bf_multiple uses micro-moves of unit duration, we set the depth to be 30 for it. IMAGE_SIZE, which is used by the model-checker to decide in db_single_1 when to shift from depth-first to breadth-first mode, is set to 10.

algorithms	run 1	run 2	run 3
df_single_1	(429, $\times$)	(2000, $\uparrow$)	(1700, $\times$)
db_single_1	(2000, $\uparrow$)	(1051, $\times$)	(283, $\times$)
bf_single_1	(2000, $\uparrow$)	(2000, $\uparrow$)	(2000, $\uparrow$)
bf_multiple	(2000, $\uparrow$)	(2000, $\uparrow$)	(2000, $\uparrow$)

Table 2: Different algorithms run on M1.

From the table, the depth-breadth search algorithm (db_single_1) performs marginally better than the depth-first search algorithm (df_single_1), although they both have one case out of 3 exceeding the time limit. Breadth-first algorithms, which carry a large reachable image for each layer and propagate this image into the next layer, fail to go deep enough to uncover a bug before the time-limit is reached. This is reflected in that both bf_single_1 and bf_multiple fail to kill M1 within the time-limit.

As we expected, breadth-first approaches are not good for debugging a large specification instance. In fact, the fast-growing sizes of the reachable images at each layer usually make it impossible for the model-checker to go deeper than a few layers (for bf_single_1, the model-checker only completed 5 layers in the time-limit.) Therefore, it is unlikely that a nontrivial bug will be uncovered.

Next we will look at whether an approximation technique, like the partial image technique, would make the model-checker faster and more effective in finding a bug for these algorithms. When an image is of size more than IMAGE_SIZE the partial image technique will randomly cut the image in half. For breadth-first algorithms, we should choose a larger IMAGE_SIZE than for the depth-first or depth-breadth search algorithm, since otherwise for each layer too much information is lost after the cut. Therefore, we run the above tests with the partial image technique turned on and using 10 and 20 for IMAGE_SIZE when bf_single_1 and bf_multiple are applied. Other settings are the same as in Table 2.

The test results are presented in Table 3, where, for instance, "bf_multiple (20)" means the algorithm bf_multiple is used with IMAGE_SIZE=20. Using the partial image approximation technique, both df_single_1 and db_single_1 are much faster in finding a bug. Under IMAGE_SIZE=10, both bf_single_1 and bf_multiple are able to complete the search but they are unable to kill M1. When IMAGE_SIZE is increased to 20, bf_single_1 kills M1 in each of the three runs, but bf_multiple is still unable to complete the search.

algorithms	run 1	run 2	run 3
df_single_1	(49, $\times$)	(70, $\times$)	(143, $\times$)
db_single_1	(254, $\times$)	(321, $\times$)	(225, $\times$)
bf_single_1 (10)	(393, $\surd$)	(395, $\surd$)	(394, $\surd$)
bf_single_1 (20)	(567, $\times$)	(567, $\times$)	(569, $\times$)
bf_multiple (10)	(1099, $\surd$)	(1098, $\surd$)	(1099, $\surd$)
bf_multiple (20)	(2000, $\uparrow$)	(2000, $\uparrow$)	(2000, $\uparrow$)

Table 3: Different algorithms run on M1 using the partial image technique.

Our experience shows that the approximation techniques usually do not work well for breadth-first search algorithms, such as bf_single_1 and bf_multiple. The reason is that given a small IMAGE_SIZE, cutting the image in half loses too much information as compared to using the whole image for breadth-first search. However, for a large IMAGE_SIZE, the search procedure does not speed up much, since the image calculations are still expensive. In contrast, for depth-first as well as depth-breadth search algorithms, the image size along a path is much smaller than the one for breadth-first search. Thus, even a small IMAGE_SIZE will make sure that a reasonably rich information source is preserved, even after an image cut.

6.2 Experiments with the Approximation Techniques

In this group of tests, we thoroughly run the two depth-first algorithms dfs_single_1 and dfs_single_2 with and without using the approximation techniques. The intent is to see whether using the approximation techniques speeds up the search procedures while remaining effective in detecting violations in a mutation. Each of the two algorithms is run three times on a test case under each combination of the approximation techniques, including without using approximations. The search depth, i.e., the bound on the number of transitions in a path, is set to be 10. When the partial image technique is applied, the tolerance of image size, IMAGE_SIZE, is set to be 10. That is, as indicated in the algorithms before, if the current reachable image is larger than 10 and the partial image technique is being used, the current image is (randomly) cut by a half. Each test result is written as a triple. For instance, (23, 1234, $\times$) means the model-checker found a violation after visiting 23 nodes[9]

[9] The number of nodes here should not be confused with (and has no relation to) the number of nodes for a BDD. The latter number usually appears in a scale of millions. As we mentioned, a node in the this paper means a transition instance. Traversing even hundreds of nodes (i.e., exercising hundreds of transitions) is very expensive for a large process instance as shown in the following figures.

cases	plain			r.w.		
	run 1	run 2	run 3	run 1	run 2	run 3
M1	(28,429,$\times$)	(48,2000,$\uparrow$)	(58,1700,$\times$)	(98,389,$\times$)	(40,251,$\times$)	(148,725,$\times$)
M2	(54,2000,$\uparrow$)	(18,962,$\uparrow$)	(61,2000,$\uparrow$)	(95,757,$\surd$)	(135,816,$\surd$)	(148,1875,$\surd$)
M3	(78,1820,$\times$)	(115,2000,$\uparrow$)	(45,2000,$\uparrow$)	(75,2000,$\uparrow$)	(28,457,$\times$)	(160,728,$\surd$)
M4	(95,2000,$\uparrow$)	(74,1784,$\times$)	(50,2000,$\uparrow$)	(25,466,$\times$)	(95,1413,$\surd$)	(80,403,$\surd$)
M5	(100,2000,$\uparrow$)	(90,2000,$\uparrow$)	(65,2000,$\uparrow$)	(130,1327,$\surd$)	(45,2000,$\uparrow$)	(83,1080,$\times$)
M6	(27,502,$\times$)	(60,2000,$\uparrow$)	(10,71,$\times$)	(44,175,$\times$)	(21,131,$\times$)	(40,249,$\times$)
M7	(10,97,$\times$)	(16,151,$\times$)	(16,254,$\times$)	(32,218,$\times$)	(70,2000,$\uparrow$)	(94,633,$\times$)
M8	(35,2000,$\uparrow$)	(37,2000,$\uparrow$)	(45,2000,$\uparrow$)	(95,2000,$\uparrow$)	(45,2000,$\uparrow$)	(65,481,$\surd$)
M9	(95,2000,$\uparrow$)	(70,2000,$\uparrow$)	(82,2000,$\uparrow$)	(150,2000,$\uparrow$)	(40,94,$\surd$)	(120,622,$\surd$)
M10	(115,2000,$\uparrow$)	(80,2000,$\uparrow$)	(90,2000,$\uparrow$)	(87,803,$\times$)	(125,1243,$\surd$)	(129,725,$\times$)
cases	p.i.			d.e.		
	run 1	run 2	run 3	run 1	run 2	run 3
M1	(13,49,$\times$)	(31,70,$\times$)	(143,484,$\times$)	(20,40,$\times$)	(100,275,$\times$)	(49,104,$\times$)
M2	(175,576,$\times$)	(148,423,$\times$)	(317,952,$\times$)	(258,2000,$\uparrow$)	(661,1482,$\times$)	(395,2000,$\uparrow$)
M3	(251,700,$\times$)	(177,492,$\times$)	(38,120,$\times$)	(120,2000,$\uparrow$)	(88,399,$\times$)	(240,2000,$\uparrow$)
M4	(202,558,$\times$)	(190,547,$\times$)	(50,137,$\times$)	(191,1389,$\times$)	(250,2000,$\uparrow$)	(28,145,$\times$)
M5	(18,97,$\times$)	(28,128,$\times$)	(41,181,$\times$)	(95,1509,$\times$)	(73,419,$\times$)	(86,1396,$\times$)
M7	(24,68,$\times$)	(24,51,$\times$)	(77,252,$\times$)	(68,248,$\times$)	(23,92,$\times$)	(51,195,$\times$)
M6	(96,270,$\times$)	(75,250,$\times$)	(38,105,$\times$)	(12,99,$\times$)	(25,121,$\times$)	(61,155,$\times$)
M8	(265,707,$\surd$)	(265,708,$\surd$)	(265,712,$\surd$)	(115,2000,$\uparrow$)	(135,2000,$\uparrow$)	(135,2000,$\uparrow$)
M9	(460,1235, $\surd$)	(460,1234, $\surd$)	(460,1232,$\surd$)	(104,1685,$\uparrow$)	(145,2000,$\uparrow$)	(275,2000,$\uparrow$)
M10	(29,157,$\times$)	(178,548,$\times$)	(80,230,$\times$)	(74,385,$\times$)	(23,130,$\times$)	(31,259,$\times$)

Table 4: Experiments for dfs_single_1: the approximation techniques used separately

cases	r.w. + p.i.			r.w. + d.e.		
	run 1	run 2	run 3	run 1	run 2	run 3
M1	(130,305,$\surd$)	(130,220,$\surd$)	(96,185,$\times$)	(90,1207,$\surd$)	(85,323,$\surd$)	(130,172,$\surd$)
M2	(220,426,$\surd$)	(316,190,$\surd$)	(143,314,$\times$)	(115,500,$\surd$)	(155,505,$\surd$)	(160,316,$\surd$)
M3	(80,132,$\times$)	(45,58,$\surd$)	(74,132,$\times$)	(95,400,$\times$)	(115,397,$\surd$)	(220,793,$\surd$)
M4	(98,205,$\times$)	(95,136,$\surd$)	(130,207,$\surd$)	(140,1174,$\surd$)	(190,822,$\surd$)	(95,284,$\surd$)
M5	(135,259,$\surd$)	(210,447,$\surd$)	(105,171,$\surd$)	(70,134,$\surd$)	(140,445,$\times$)	(40,112,$\surd$)
M6	(18,81,$\times$)	(58,137,$\times$)	(71,138,$\times$)	(79,457,$\times$)	(14,63,$\times$)	(68,213,$\times$)
M7	(75,114,$\surd$)	(60,104,$\times$)	(32,59,$\times$)	(75,231,$\times$)	(170,328,$\surd$)	(58,225,$\times$)
M8	(75,133,$\surd$)	(100,164,$\surd$)	(85,136,$\surd$)	(105,442,$\surd$)	(80,2000,$\uparrow$)	(110,2000,$\uparrow$)
M9	(55,97,$\surd$)	(145,251,$\surd$)	(115,199,$\surd$)	(125,542,$\surd$)	(185,682,$\surd$)	(205,456,$\surd$)
M10	(210,401,$\surd$)	(114,255,$\times$)	(185,390,$\times$)	(120,312,$\surd$)	(150,368,$\surd$)	(175,735,$\surd$)
cases	p.i. + d.e.			r.w. + p.i. + d.e.		
	run 1	run 2	run 3	run 1	run 2	run 3
M1	(155,374,$\times$)	(26,64,$\times$)	(50,128,$\times$)	(130,269,$\times$)	(32,78,$\times$)	(18,62,$\times$)
M2	(275,693,$\surd$)	(385,873,$\surd$)	(310,772,$\surd$)	(105,173,$\surd$)	(85,137,$\surd$)	(90,154,$\surd$)
M3	(137,290,$\times$)	(41,113,$\times$)	(23,52,$\times$)	(105,151,$\surd$)	(85,106,$\surd$)	(105,155,$\surd$)
M4	(116,244,$\times$)	(259,547,$\times$)	(95,208,$\times$)	(115,148,$\surd$)	(65,79,$\surd$)	(150,204,$\surd$)
M5	(48,167,$\times$)	(177,465,$\times$)	(224,531,$\times$)	(115,171,$\surd$)	(62,164,$\times$)	(150,232,$\surd$)
M6	(124,358,$\times$)	(102,288,$\times$)	(18,37,$\times$)	(87,200,$\times$)	(75,102,$\surd$)	(53,108,$\times$)
M7	(56,117,$\times$)	(18,63,$\times$)	(31,68,$\times$)	(17,38,$\times$)	(91,161,$\times$)	(77,157,$\times$)
M8	(200,421,$\surd$)	(240,527,$\surd$)	(190,410,$\surd$)	(110,174,$\surd$)	(50,64,$\surd$)	(70,110,$\surd$)
M9	(370,825,$\surd$)	(310,653,$\surd$)	(425,1006,$\surd$)	(135,192,$\surd$)	(145,278,$\surd$)	(170,303,$\surd$)
M10	(27,113,$\times$)	(240,586,$\times$)	(27,62,$\times$)	(98,216,$\times$)	(85,131,$\times$)	(129,236,$\times$)

Table 5: Experiments for dfs_single_1: the approximation techniques used in combination

cases	plain			r.w.		
	run 1	run 2	run 3	run 1	run 2	run 3
M1	(20,108,$\times$)	(24,695,$\times$)	(54,1250,$\times$)	(31,151,$\times$)	(84,867,$\times$)	(80,329,$\times$)
M2	(53,2000,$\uparrow$)	(110,2000,$\uparrow$)	(132,2000,$\uparrow$)	(85,926,$\surd$)	(105,2000,$\uparrow$)	(152,2000,$\uparrow$)
M3	(29,158,$\times$)	(100,2000,$\uparrow$)	(34,213,$\times$)	(65,2000,$\uparrow$)	(100,831,$\surd$)	(22,149,$\times$)
M4	(40,1636,$\times$)	(210,2000,$\uparrow$)	(200,2000,$\uparrow$)	(80,2000,$\uparrow$)	(85,2000,$\uparrow$)	(50,107,$\surd$)
M5	(43,168,$\times$)	(17,134,$\times$)	(100,2000,$\uparrow$)	(18,151,$\times$)	(31,173,$\times$)	(97,768,$\times$)
M6	(19,76,$\times$)	(44,968,$\times$)	(11,71,$\times$)	(120,1162,$\surd$)	(150,2000,$\uparrow$)	(23,86,$\times$)
M7	(39,506,$\times$)	(23,256,$\times$)	(35,502,$\times$)	(57,258,$\times$)	(22,92,$\times$)	(55,225,$\times$)
M8	(50,2000,$\uparrow$)	(39,2000,$\uparrow$)	(90,2000,$\uparrow$)	(50,2000,$\uparrow$)	(20,27,$\surd$)	(145,1633,$\surd$)
M9	(90,2000,$\uparrow$)	(205,2000,$\uparrow$)	(105,2000,$\uparrow$)	(225,1556,$\surd$)	(125,854,$\surd$)	(20,24,$\surd$)
M10	(184,1860,$\times$)	(20,1610,$\times$)	(19,270,$\times$)	(133,580,$\times$)	(59,603,$\times$)	(95,439,$\surd$)
cases	p.i.			d.e.		
	run 1	run 2	run 3	run 1	run 2	run 3
M1	(68,31,$\times$)	(36,87,$\times$)	(148,360,$\times$)	(133,306,$\times$)	(79,298,$\times$)	(19,103,$\times$)
M2	(448,1163,$\times$)	(439,1129,$\times$)	(406,1032,$\times$)	(314,2000,$\uparrow$)	(267,874,$\times$)	(290,1581,$\surd$)
M3	(102,236,$\times$)	(266,585,$\times$)	(89,231,$\times$)	(34,102,$\times$)	(44,269,$\times$)	(279,1581,$\times$)
M4	(49,131,$\times$)	(198,453,$\times$)	(20,48,$\times$)	(237,759,$\times$)	(125,2000,$\uparrow$)	(191,1063, $\times$)
M5	(241,590,$\times$)	(266,647,$\times$)	(22,75,$\times$)	(182,1171,$\times$)	(243,946,$\times$)	(169,1118,$\times$)
M6	(96,231,$\times$)	(74,196,$\times$)	(49,138,$\times$)	(36,100,$\times$)	(87,279,$\times$)	(25,49,$\times$)
M7	(18,43,$\times$)	(22,54,$\times$)	(25,60,$\times$)	(118,760,$\times$)	(14,70,$\times$)	(16,59,$\times$)
M8	(265,608,$\surd$)	(265,605,$\surd$)	(265,607,$\surd$)	(205,1593,$\surd$)	(200,2000,$\uparrow$)	(135,2000,$\uparrow$)
M9	(465,1043,$\surd$)	(465,1044,$\surd$)	(465,1048,$\surd$)	(250,2000,$\uparrow$)	(274,2000,$\uparrow$)	(205,2000,$\uparrow$)
M10	(26,75,$\times$)	(24,72,$\times$)	(13,54,$\times$)	(238,1194,$\times$)	(220,611,$\times$)	(19,110,$\times$)

Table 6: Experiments for dfs_single_2: the approximation techniques used separately

cases	r.w. + p.i.			r.w. + d.e.		
	run 1	run 2	run 3	run 1	run 2	run 3
M1	(45,85,×)	(70,104,√)	(99,185,×)	(100,295,√)	(150,298,√)	(110,246,√)
M2	(65,100,√)	(115,188,√)	(130,227,√)	(89,1611,×)	(95,243,,√)	(75,650,√)
M3	(110,144,√)	(130,183,√)	(215,307,√)	(95,268,√)	(70,159,√)	(143,366,√)
M4	(26,52,×)	(39,70,×)	(19,44,×)	(145,296,√)	(75,148,√)	(130,565,√)
M5	(30,37,√)	(115,196,×)	(25,34,√)	(45,76,√)	(120,276,√)	(105,233,√)
M6	(88,165,×)	(14,36,×)	(16,34,×)	(69,268,×)	(42,621,×)	(12,38,×)
M7	(103,171,×)	(80,120,×)	(19,38,×)	(21,111,×)	(26,91,×)	(21,123,×)
M8	(100,164,√)	(100,172,√)	(45,63,√)	(90,706,√)	(60,285,√)	(80,431,√)
M9	(215,369,√)	(75,121,√)	(150,213,√)	(85,228,√)	(155,739,√)	(115,283,√)
M10	(150,232,√)	(68,107,×)	(30,65,×)	(70,134,√)	(75,266,√)	(70,102,√)
cases	p.i. + d.e.			r.w. + p.i. + d.e.		
	run 1	run 2	run 3	run 1	run 2	run 3
M1	(173,386,×)	(106,211,×)	(27,86,×)	(16,41,×)	(76,133,×)	(75,147,√)
M2	(410,878,√)	(317,695,×)	(355,706,√)	(175,247,√)	(75,99,√)	(165,233,√)
M3	(21,42,×)	(285,511,×)	(32,63,×)	(60,70,√)	(125,166,√)	(120,155,√)
M4	(315,603,√)	(39,95,×)	(209,401,×)	(148,220,×)	(130,172,√)	(28,66,×)
M5	(39,108,×)	(294,605,×)	(17,50,×)	(120,174,√)	(96,146,×)	(191,319,×)
M6	(52,119,×)	(36,80,×)	(28,56,×)	(74,111,×)	(80,114,√)	(80,108,√)
M7	(15,41,×)	(178,345,×)	(16,45,×)	(73,108,×)	(36,60,×)	(57,89,×)
M8	(100,198,√)	(220,430,√)	(210,387,√)	(65,105,√)	(140,229,√)	(80,111,√)
M9	(350,698,√)	(395,803,√)	(380,746,√)	(120,183,√)	(185,267,√)	(130,169,√)
M10	(86,181,×)	(36,100,×)	(87,196,×)	(100,172,×)	(65,103,×)	(90,133,√)

Table 7: Experiments for dfs_single_2: the approximation techniques used in combination

in the execution tree taking 1234 seconds. The status values are "$\times$", "$\surd$" and "$\uparrow$" [10]. The time-limit, as mentioned before, is set to be 2000 CPU seconds.

Tables 4, 5, 6 and 7 show the test results. Among the ten mutations, M8 and M9 are both correct, though they are not strictly equivalent to the original specification. For instance, M8 deletes the conjunction

$$\texttt{now} - \texttt{End(lower)} >= \texttt{lower\_time}$$

from the entry assertion of the transition `down`. Doing this allows `down` to fire immediately after the transition `lower`, while in the original specification there is a delay of `lower_time` between them. This won't cause any problems for the critical requirements of the `Gate` process, because eliminating the delay is essentially a special case of the original specification where `lower_time=0`. All of the other mutations are flawed. Therefore, the model-checker is expected to find violations.

The test results are evaluated by two criteria:

- On average, how much time is spent in finding an error when using a specific combination of the approximation techniques or without using them. This is called *cost per error*, denoted by CPE.

- The average chance that a run is able to kill a mutant. This is called *error finding ratio*, denoted by EFR.

Based upon the test data, we calculate CPE as follows. Excluding the live mutants (M8 and M9, since both are correct), are the remaining eight mutations able to be killed? For instance, for the algorithm dfs_single_1 without using approximations (i.e., under the column "plain" in Table 4), CPE(dfs_single_1, plain) is the total run time of the eight mutations divided by the total number of runs where an error was found, i.e.,

$$39770/9 = 4418.$$

This indicates the average time to find a violation. All the CPEs are presented in Table 8 under the columns "cost per error".

We would also like to know whether aggressive approximations are effective in detecting violations. This is measured by the EFR. For instance, for the algorithm dfs_single_2 with the random walk technique, the three columns under "r.w." show that 13 runs out of 24 for the eight mutations successfully kill a mutant. Since each of the eight mutations are flawed, the expected number of successful runs is 24. Therefore, EFR(dfs_single_2, r.w.) is

$$13/24.$$

[10] Occasionally, the Omega library core-dumped on an image calculation for extremely large images. If this happens, $\uparrow$ status is possible even when time-limit is not reached.

This indicates the chance that the model-checker is able to kill a mutant using the specific algorithm and the random walk technique. All the EFRs are presented in Table 8 under the columns "error finding ratio".

approximations	dfs_single_1		dfs_single_2	
	cost per error	error finding ratio	cost per error	error finding ratio
plain	4418	9/24	1498	17/24
r.w.	1896	13/24	1530	13/24
p.i.	310	24/24	317	24/24
d.e.	991	19/24	777	22/24
r.w. + p.i.	406	12/24	224	13/24
r.w. + d.e.	1502	7/24	1017	7/24
p.i. + d.e.	344	21/24	288	21/24
r.w. + p.i. + d.e.	302	12/24	260	13/24

Table 8: Average cost of finding an error and the ratio of the number of errors found to the total number of errors for dfs_single_1 and dfs_single_2

A number of observations can be made from the results shown in Table 8.

- The difference of the two algorithms dfs_single_1 and dfs_single_2 is the different level in strengthening a transition, as previously introduced. dfs_single_2 is more accurate than dfs_single_1 in approximating the environment. Thus, it reduces the size of reachable images and makes the model-checker faster. Therefore, dfs_single_2 makes it possible for the model-checker to search more nodes within the time-limit, and it makes it more likely that the model-checker will find a bug. This is reflected in both the CPRs (4418 vs. 1498) and EFRs (9/24 vs. 17/24) for the two algorithms without using the approximation techniques. Thus, the data under the column "dfs_single_1" in Table 8 indicate that when the approximation techniques are not used the model-checker usually runs out of time before a violation is found. The data under the column "dfs_single_2" in Table 8 indicate that even without the approximation techniques the model-checker usually is able to find a bug marginally within the time-limit.

- The partial image technique appears to be the most effective approach. For the algorithm dfs_single_1, it is 14 (4418/310) times faster than without using approximations. For dfs_single_2, it is almost 5 (1498/317) times faster. Notice that, for both algorithms, the EFRs with the partial image technique are 24/24, which means the technique is fast and extremely effective.

- The dynamic environment generation technique has high EFRs (19/24 and 22/24), even when it is used in combination with the partial image technique (21/24 and 21/24). For the two algorithms, it is 4 (4418/991) and 2 (1498/777) times faster than without using approximations, respectively. When used in combination, it is 12 (4418/344) and 5 (1498/288) times faster than without using approximations, respectively, while still keeping extremely high EFRs.

- Surprisingly, the random walk technique does not perform as well as expected. Used alone, it has low EFRs (13/24 and 13/24) and high costs per error (1896 and 1530). The reason is that even though a number of branches are skipped during the search procedure, the image computations along a single path are still expensive, and in some cases the model-checker is unable to hit the violation path before the time-limit is reached. In addition, skipping a branch means the whole subtree that starts from the branch is skipped by the model-checker. This approach sacrifices more coverage than the other two techniques; therefore, this technique has a higher chance of skipping a violation path. This fact is indicated by its low EFRs even when the technique is used in combination with the other two techniques. Thus, compared to the other two techniques, we do not consider random walk to be effective when used alone. But used in combination with the partial image technique (in both r.w.+p.i. and r.w.+p.i.+d.e.) it is acceptable. In this case, the results in the table point out that it has around 12/24 EFRs and on average is 6 times faster than without using the approximations.

- The techniques used in combination could speed up the search procedure, but, obviously since more coverage is sacrificed, this usage does not necessarily imply more effectiveness. For instance, the dynamic environment generation approach has very high EFRs (19/24 and 22/24). But when it is used with the other two techniques, the EFRs drop to 12/24 and 13/24, respectively. This is compensated by almost 3 times lower cost per error. For the partial image technique there is almost no compensation in CPRs at all when it is used with the other techniques.

As mentioned before, the two live mutations M8 and M9 are both correct. Runs on them can be used to analyze node coverage for each approximation technique. Node coverage (NC) is measured by the total number of nodes searched in both M8 and M9 for a specific algorithm. For instance, for dfs_single_1 with the dynamic environment generation technique used alone, the sum of nodes under columns "d.e." in Table 4 for M8 and M9 is 779. This number is NC(dfs_single_1, d.e.). All the NCs are presented in Table 9. The partial image technique and the dynamic environment generation technique, either used separately or in combination, generally have much higher node coverage (NC(dfs_single_1, d.e.)=779 is an exception) than all the others. This explains the fact that these techniques also have extremely high EFRs as shown in Table 8. In contrast, the random walk technique, when used separately or in combination with the other techniques, has significantly low node coverage. This is also consistent with the fact that it has low EFRs as shown in Table 8.

From the above analysis, one can conclude that the partial image technique and the

approximations	dfs_single_1	dfs_single_2
plain	364	579
r.w.	810	585
p.i.	2175	2190
d.e.	779	1269
r.w. + p.i.	575	685
r.w. + d.e.	810	585
p.i. + d.e.	1735	1655
r.w. + p.i. + d.e.	680	720

Table 9: Node coverage (NC) for the M8 and M9

dynamic environment generation technique are both effective, either when used separately or in combination, – they are able to kill all the mutants in almost every run and in a much shorter time. The random walk technique, when used alone, is not considered effective, due to the fact that it is not significantly faster in finding an error and it suffers from a low EFR. The situation is improved somewhat when it is used in combination with the other two techniques.

7 Conclusions and Future Work

This paper describes a prototype symbolic model-checker for a subset of ASTRAL that uses a Presburger representation and the Omega library to carry out image computations. The model-checker is primarily intended for use as a specification debugger. Keeping this in mind, we used the model-checker as a platform to experiment with a number of search strategies, environment approximation techniques, and aggressive image approximation techniques to speed-up the model-checker's debugging procedure while remaining effective in finding errors.

The symbolic model checker performs image computations on the execution tree of a Small-ASTRAL process instance. We proposed three symbolic search strategies on the tree. They are depth-first search, breadth-first search, and depth-breadth search. Different levels of approximation of the environment behaviors are also provided in order to strengthen a transition in a process instance. Three image approximation techniques to speed up the debugging procedure of the model checker were also introduced. They are random walk, partial image and dynamic environment generation. We have demonstrated the effectiveness of the three aggressive approximations by running experiments on a large instance of a railroad crossing benchmark.

The three search strategies, which are not necessary restricted within the ASTRAL environment, are applicable for symbolically debugging other infinite-state/finite-state transition systems, e.g., sequential/concurrent programs. However, the environment approximation techniques for strengthening a transition are somewhat specific to ASTRAL. The reasons are that a transition in ASTRAL has a positive duration and, since a modularized proof theory is used, behavior abstractions of the other process instances (in the form of the environment clause and the imported variable clause) are expected to hold during an execution of the transition. Those expectations make the transition relation of a transition instance extremely expensive (or even impossible) to calculate as a Presburger formula. Thus, the environment approximation techniques provide a way to restrict the environment behaviors such that calculating the transition relation is feasible.

We expect the aggressive image approximation techniques, in particular the partial image technique (which is the best performer demonstrated by the experiments), to be applicable to debugging safety properties for systems specified in different specification languages. For instance, a similar form of the partial image technique can be developed for symbolically debugging sequential/concurrent programs; this is in contrast to the traditional random (domain) testing techniques [34, 33, 54] in program testing where a single or several samples are picked. The efficiency of using the partial image technique depends on the cost of cutting an image in half. For instance, in the context of finite state systems represented by BDDs, splitting an image (i.e., a BDD) may be computationally expensive [16]. In this paper, cutting a (Presburger) image is cheap, thanks to the internal representation of a Presburger formula as a union of convex regions by the Omega library. If we had a practical solver (which we are not aware of) for mixed domains including bounded variables, integer variables, and real variables, the partial image technique might also be useful for debugging systems containing variables in mixed domains. In this paper we only consider systems with bounded variables and integer variables. Bounded variables, such as Boolean variables, are encoded as integer variables. This encoding is expensive, since the efficiency of using the Presburger solver (i.e., the Omega library) is sensitive to the number of variables. Bultan et. al. [12] proposed an approach of using both BDD representations and Presburger representations without encoding bounded variables into integer ones. Their approach is applicable only to a restricted form of mixed bounded/integer formulas, and in particular, does not work on general quantified formulas, which are widely used in ASTRAL specifications. However, for specifications without quantified formulas, their mixed BDD/Presburger image representation approach could be used, and the partial image technique could be used accordingly on BDDs and Presburger representations, respectively.

The environment approximation techniques may cause false negatives in general, though the benchmark specification used in the experiments of this paper does not cause false negatives. Checking whether an error is a real one is expensive, in particular, when the violation trace is long. One problem that needs to be solved in the future is to find an inexpensive way to strengthen a transition such that false negatives can be avoided.

The search strategies (with or without using the image approximation techniques) would be better if a number of heuristics could be developed to guide the model-checker to an error. For finite state systems, we can measure how large a set of erroneous states is; i.e., whether errors are dense or sparse with respect to a subset of states. However, for infinite state systems, we do not know how to define that a class of errors is dense or sparse when there are infinitely many errors in the class (which is the usual case). This is an obstacle to developing a good heuristic for the model-checker presented in this paper such that the heuristic would always search a branch (or pick a denser partial image) with a higher chance of finding an error.

The coverage analysis in this paper is empirical. The factors considered are time and number of nodes. An issue to be investigated is what metrics can be used to systematically measure path and/or node coverage for a specific approximation technique applied on an execution tree. This is a challenging topic, since sometimes the symbolic model checker without using the approximation techniques fails to complete the entire search. Therefore, a theoretical estimation is needed.

The authors would like to thank T. Bultan and P. Kolano for many insightful discussions. The specifications, including the current version and the earlier erroneous versions, were written by P. Kolano in the Reliable Software Group at UC Santa Barbara. Thanks also go to Anurag Acharya for providing the machine for running all the tests presented in this paper.

References

[1] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in dense real time," *Information and Computation,* vol. 104, no. 1, pp. 2-34, 1993.

[2] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science,* vol. 126, no. 2, pp. 183-236, 1994.

[3] R. Alur, T. Feder, and T. A. Henzinger, "The benefits of relaxing punctuality," *Journal of the ACM*, vol. 43, no. 1, pp. 116-146, 1996.

[4] R. Alur, T. A. Henzinger, "Real-time logics: complexity and expressiveness," *Information and Computation*, vol. 104, no. 1, pp. 35-77, 1993.

[5] R. Alur and T. A. Henzinger, "A really temporal logic," *Journal of the ACM,* vol. 41, no. 1, pp. 181-204, 1994.

[6] R. Alur, T. A. Henzinger, and M. Y. Vardi. "Parametric real-time reasoning," *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing,* San Diego, California, 1993, pp. 592-601.

[7] N. S. Bjorner, A. Browne, E. Chang, M. Colon, A. Kapur, Z. Manna, H. Sipma, and T. Uribe, "STeP: deductive-algorithmic verification of reactive and real-time systems," *Proceedings of the 8th International Conference on Computer Aided Verification,* New Brunswick, New Jersey, 1996, pp. 415-418.

[8] J. P. Bowen, R. W. Butler, D. L. Dill, R. L. Glass, D. Gries, A. Hall, M. G. Hinchey, C. M. Holloway, D. Jackson, C. B. Jones, M. J. Lutz, D. L. Parnas, J. M. Rushby, J. M. Wing, and P. Zave, "An invitation to formal methods," *IEEE Computer*, vol. 29, no. 4, pp. 16-30, 1996.

[9] K. Brink, L. Bun, J. van Katwijk and W. J. Toetenel, "Hybrid specification of control systems," *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems,* Ft. Lauderdale, Florida, 1995

[10] R.E. Bryant, "Graph-Based algorithms for Boolean function manipulation," *IEEE Transactions on Computers,* vol. 35, no. 8, pp. 677-691, 1986.

[11] R.E. Bryant, "Symbolic Boolean manipulation with ordered binary decision diagrams," *ACM Computing Surveys,* vol. 24, no. 3, pp. 293-318, 1992.

[12] T. Bultan, R. Gerber, and C. League, "Verifying systems with integer constraints and Boolean predicates: a composite approach," *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis,* Clearwater Beach, Florida, 1998, pp. 113-123.

[13] T. Bultan, R. Gerber, and W. Pugh, "Symbolic model-checking of infinite state systems using Presburger arithmetic," *Proceedings of the 9th International Conference on Computer Aided Verification,* Haifa, Israel, 1997, pp. 400-411.

[14] G. Buonanno, A. Coen-Porisini and W. Fornaciari, "Hardware specification using the assertion language ASTRAL," *Proceedings of the Advanced Research Workshop on Correct Hardware Design Methodologies,* Torino, Italy, 1991.

[15] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering,* vol. 2, no. 3, pp. 215-222, 1976.

[16] G. Cabodi, P. Camurati, and S. Quer, "Improved reachability analysis of large finite state machines," *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design,* San Jose, California, 1996, pp. 354-360.

[17] E. M. Clarke, E. A. Emerson and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 234-263, 1986.

[18] E. M. Clarke, O. Grumberg, and D. E. Long, "Model-checking and abstraction," *Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Albequerque, New Mexico, 1992, pp. 342-354.

[19] C. Colby, P. Godefroid and L. J. Jagadeesan, "Automatically closing open reactive programs," *SIGPLAN Notices*, vol. 33, no. 5, pp. 345-357, 1998.

[20] P. Cousot and R. Cousot, "Refining model checking by abstract interpretation," *Automated Software Engineering,* vol. 6, no. 1, pp. 69-95, 1999.

[21] A. Coen-Porisini, C. Ghezzi and R. A. Kemmerer, "Specification of real-time systems using ASTRAL," *IEEE Transactions on Software Engineering*, vol. 23, no. 9, pp. 572-598, 1997.

[22] A. Coen-Porisini, R. A. Kemmerer and D. Mandrioli, "A formal framework for ASTRAL intralevel proof obligations," *IEEE Transactions on Software Engineering*, vol. 20, no. 8, pp. 548-561, 1994.

[23] D. Dams, R. Gerth and O. Grumberg, *"Abstract interpretation of reactive systems," ACM Transactions on Programming Languages and Systems* , vol. 19, no. 2, pp. 253-291, 1997.

[24] Z. Dang, "Verification and debugging of infinite state real-time systems," PhD. Dissertation, University of California at Santa Barbara, August, 2000.

[25] Z. Dang and R. Kemmerer, "Using the ASTRAL model checker for cryptographic protocol analysis," *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols*, Rutgers University, 1997.

[26] Z. Dang and R. A. Kemmerer, "Using the ASTRAL model checker to analyze Mobile IP," *Proceedings of the Twenty-first International Conference on Software Engineering*, Los Angeles, California, 1999, pp. 132-141.

[27] Z. Dang and R. A. Kemmerer, "A symbolic model-checker for testing ASTRAL real-time specifications," *Proceedings of the Sixth International Conference on Real-time Computing Systems and Applications,* Hong Kong, China, 1999, pp. 131-142.

[28] Z. Dang and R. A. Kemmerer, "Three approximation techniques for ASTRAL symbolic model checking of infinite state real-time systems," *Proceedings of the Twenty-second International Conference on Software Engineering,* Limerick, Ireland, 2000, pp. 345-354.

[29] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer,* vol. 11, no. 4, pp. 34-43, 1978.

[30] D. L. Dill, "The Murφ verification system," *Proceedings of the 8th International Conference on Computer Aided Verification,* New Brunswick, New Jersey, 1996, pp. 390-393.

[31] D. L. Dill and H. Wong-Toi, "Verification of real-time systems by successive over and under approximation," *Proceedings of the 7th International Conference on Computer Aided Verification*, Liege, Belgium, 1995, pp. 409-422.

[32] J. Douglas and R. A. Kemmerer, "Aslantest: a symbolic execution tool for testing Aslan formal specifications," *Proceedings of the International Symposium on Software Testing and Analysis,* Seattle, Washington, 1994, pp. 15-27.

[33] J. W. Duran and S. Ntafos, "A report on random testing," *Proceedings of the 5th International Conference on Software Engineering*, San Diego, California, 1981, pp. 179-183.

[34] J. W. Duran and J. J. Wiorkowski, "Quantifying software validity by sampling," *IEEE Transactions on Reliability*, vol. R-29, pp. 141-144, 1980.

[35] R. Fraer, G. Kamhi, B. Ziv, M. Y. Vardi and L. Fix, "Prioritized traversal: efficient reachability analysis for verification and falsification," *Proceedings of the 12th International Conference on Computer Aided Verification*, Chicago, Illinois, 2000, pp. 389-402.

[36] A. Gabrielian and M. K. Franklin, "Multilevel specification of real time systems," *Communications of the ACM*, vol. 34, no. 5, pp. 50-60, 1991.

[37] R. Gerber and I. Lee, " A layered approach to automating the verification of real-time systems," *IEEE Transactions on Software Engineering,* vol. 18, no. 9, pp. 768-784, 1992.

[38] P. Godefroid and P. Wolper, "A partial approach to model checking," *Information and Computation*, vol. 110, no. 2, 305-326, 1994.

[39] C. Heitmeyer and N. Lynch, *"The generalized railroad crossing: a case study in formal verification of real-time systems,"* Proceedings of the Fifteenth IEEE Real-time Systems Symposium, San Juan, Puerto Rico, 1994, pp. 120-131.

[40] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, *"Symbolic model-checking for real-time systems," Information and Computation*, vol. 111, no. 2, pp. 193-244, 1994.

[41] G. J. Holzmann. *"Design and validation of computer protocols,"* Prentice Hall, 1991

[42] G. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering,* vol. 23, no. 5, pp. 279-295, 1997.

[43] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering,* vol. 8, no. 4, pp. 371-379, 1982.

[44] R. A. Kemmerer and S. T. Eckman, "UNISEX: a UNIX-based Symbolic EXecutor for Pascal," *Software - Practice and Experience,* vol. 15, no. 5, pp. 439-458, 1985.

[45] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385-394, 1976.

[46] P. Z. Kolano, "Proof assistance for real-time systems using an interactive theorem prover," *Proceedings of the 5th International AMAST Workshop*, Bamberg, German, 1999, pp. 315-333.

[47] P. Z. Kolano, Z. Dang and R. Kemmerer, "The design and analysis of real-time systems using the ASTRAL software development environment," *Annals of Software Engineering*, vol. 7, pp. 177-210, 1999.

[48] G. Kreisel and J. L. Krevine, *Elements of mathematical logic,* North-Holland, 1967

[49] K. L. McMillan, "Symbolic model checking - an approach to the state explosion problem," PhD thesis, Carnegie Mellon University, 1992.

[50] D. C. Oppen, "A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic," *Journal of Computer and System Sciences,* vol. 16, no. 3, pp. 323-332, 1978.

[51] W. Pugh, "The Omega test: a fast and practical integer programming algorithm for dependence analysis," *Communications of the ACM*, vol. 35, no. 8, pp. 102-114, 1992.

[52] J. Raskin and P. Schobben, "State clock logic: a decidable real-time logic," *Proceedings of the International Workshop on Hybrid and Real-Time Systems*, Lecture Notes in Computer Science 1201, pp. 33-47, 1997.

[53] A. U. Shankar, "An introduction to assertion reasoning for concurrent systems," *ACM Computing Surveys*, vol. 25, no. 3, pp. 225-262, 1993.

[54] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Transactions on Software Engineering,* vol. 6, no. 3, pp. 247-257, 1980.

[55] T. Wilke, "Specifying timed state sequences in powerful decidable logics and timed automata," *Proceedings of the International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science 863, pp. 694-715, 1994.

Appendix ASTRAL Specification of Railroad Crossing

```
/* This specification is written by P. Kolano. */
SPECIFICATION Railroad_Crossing
  GLOBAL SPECIFICATION Railroad_Crossing
    PROCESSES
              the_gate: Gate,
              the_sensors: array [ 1..n_tracks ]  of Sensor
    TYPE
              pos_integer: TYPEDEF i: integer ( i > 0 ) ,
              pos_real: TYPEDEF i: integer ( i > 0 ) ,
              gate_position:  ( raised, raising, lowered, lowering ) ,
              sensor_id: TYPEDEF i: id ( IDTYPE  ( i )  = Sensor )
    CONSTANT
              n_tracks: pos_integer,
              min_speed, max_speed: pos_real,
              dist_R_to_I, dist_I_to_out: pos_real,
              response_time, wait_time: pos_real,
              RImax: pos_real,
              RIImin: pos_real,
              RIImax: pos_real
    AXIOM
      /*since the Omega library can not handle division,
        we therefore have to introduce RImax to represent
        dist_R_to_I/max_speed, RIImin to represent
        (dist_R_to_I + dist_I_to_out)/min_speed, and
        RIImax to represent (dist_R_to_I + dist_I_to_out)/max_speed*/
              max_speed >= min_speed
            & response_time < RImax
            & max_speed * RImax <= dist_R_to_I
            & max_speed * RImax + max_speed > dist_R_to_I
            & min_speed * RIImin <= dist_R_to_I + dist_I_to_out
            & min_speed * RIImin + min_speed >dist_R_to_I+dist_I_to_out
            & max_speed * RIImax <= dist_R_to_I+ dist_I_to_out
            & max_speed * RIImax + max_speed >dist_R_to_I+dist_I_to_out
    SCHEDULE
      /* gate will be down before fastest train reaches crossing */
               ( EXISTS s: sensor_id
                          ( s.train_in_R
                          & EXISTS t2: time
                                      ( t2 <= now
                                      & s.Call ( enter_R, t2 )
                                      & now - t2 >= RImax )  )
            ->   the_gate.position = lowered )
      /* gate will be up after slowest train exits
         crossing and a reasonable wait time has elapsed */
            &  ( FORALL s: sensor_id
                          ( ~s.train_in_R
                          &  ( EXISTS t: time
                                         ( s.Call ( enter_R, t )  )
                          ->   EXISTS t1: time
                                         ( t1 <= now
                                         & s.Call ( enter_R, t1 )
                                         & now - t1 >= RIImin + wait_time )  )  )
            ->   the_gate.position = raised )
  END Railroad_Crossing
  PROCESS SPECIFICATION Sensor
    LEVEL Top_Level
```

```
    IMPORT
              pos_real, max_speed, min_speed, dist_R_to_I,
              dist_I_to_out, response_time, RImax, RIImax,
              RIImin
    EXPORT
              train_in_R, enter_R
    CONSTANT
              enter_dur, exit_dur: pos_real
    VARIABLE
              train_in_R: boolean
    AXIOM
              response_time >= enter_dur
            & RIImin >= response_time + exit_dur
    ENVIRONMENT
      /* only one train will be in the region at the
         same time on the same track */
              Call ( enter_R, now )
            & EXISTS t: time
                       ( t >= 0
                       & t <= now
                       & Call [ 2 ]  ( enter_R, t )  )
         -> Call(enter_R) - Call [ 2 ] ( enter_R ) > RIImin
    INITIAL
              ~train_in_R
    INVARIANT
      /* once a sensor reports a train, it will keep
         reporting a train at least as long as it takes the
         fastest train to exit the region */
              Change ( train_in_R, now )
            & ~train_in_R
         ->   0 <= now -  ( RIImax - response_time )
            & FORALL t: time
                       ( now -  ( RIImax - response_time ) <= t
                       & t < now
                    ->   past ( train_in_R, t )  )
    SCHEDULE
      /* train will be sensed within enter_dur of call */
               ( now >= response_time
               & Call ( enter_R, now - response_time )
            ->   train_in_R )
      /* sensor will be reset when the slowest
         train is beyond the crossing */
            &  ( now >= RIImin
               & Call ( enter_R, now - RIImin )
            ->   ~train_in_R )
    TRANSITION enter_R
      ENTRY          [ TIME : enter_dur ]
                ~train_in_R
      EXIT
                train_in_R = TRUE
    TRANSITION exit_I
      ENTRY          [ TIME : exit_dur ]
                train_in_R
              & now - Start ( enter_R )  >= RIImin - exit_dur
      EXIT
                train_in_R = FALSE
  END Top_Level
END Sensor
PROCESS SPECIFICATION Gate
  LEVEL Top_Level
    IMPORT
              pos_real, gate_position, max_speed, dist_R_to_I,
```

```
          dist_I_to_out, wait_time, response_time,
          sensor_id, the_sensors.train_in_R,
          the_sensors.enter_R, RImax, RIImax, RIImin
EXPORT
          position
CONSTANT
          lower_dur, raise_dur, up_dur, down_dur: pos_real,
          raise_time, lower_time: pos_real
VARIABLE
          position: gate_position
AXIOM
          wait_time >= raise_dur + raise_time + up_dur
        & RImax >= response_time + lower_dur + lower_time
                                 + down_dur + raise_dur
        & RImax >= response_time + lower_dur + lower_time
                                 + down_dur + up_dur
INITIAL
          position = raised
SCHEDULE
  /* gate will be down before fastest train reaches crossing */
           ( EXISTS s: sensor_id
               ( s.train_in_R
                & now - Change(s.train_in_R ) >= RImax - response_time)
        ->   position = lowered )
  /* gate will be up after slowest train exits
     crossing and enough time has elapsed for gate to be
     raised */
        &  ( FORALL s: sensor_id
               ( FORALL t: time
                      ( now - wait_time <= t
                        & t <= now
                     ->   ~past ( s.train_in_R, t )  )  )
        ->   position = raised )
IMPORTED VARIABLE
  /* once a sensor reports a train, it will keep
     reporting a train at least as long as it takes the
     fastest train to exit the region */
          FORALL s: sensor_id
                   ( Change ( s.train_in_R, now )
                   & ~s.train_in_R
                ->   0 <= now -  ( RIImax - response_time )
                   & FORALL t: time
                              ( now -  ( RIImax - response_time )  <= t
                              & t < now
                           ->   past ( s.train_in_R, t )  )  )
TRANSITION lower
  ENTRY          [ TIME : lower_dur ]
            ~ ( position = lowering
             |  position = lowered )
          & EXISTS s: sensor_id
                     ( s.train_in_R )
  EXIT
            position = lowering
TRANSITION down
  ENTRY          [ TIME : down_dur ]
            position = lowering
          & now - End ( lower )  >= lower_time
  EXIT
            position = lowered
TRANSITION raise
  ENTRY          [ TIME : raise_dur ]
```

```
                  ~ ( position = raising
                   |  position = raised )
                & FORALL s: sensor_id
                         ( ~s.train_in_R )
        EXIT
                  position = raising
      TRANSITION up
        ENTRY           [ TIME : up_dur ]
                  position = raising
                & now - End ( raise )  >= raise_time
        EXIT
                  position = raised
    END Top_Level
  END Gate
END Railroad_Crossing
```