Three Approximation Techniques for ASTRAL Symbolic Model Checking of Infinite State Real-time Systems

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

ABSTRACT
ASTRAL is a high-level formal specification language for real-time systems. It has structuring mechanisms that allow one to build modularized specifications of complex real-time systems with layering. Based upon the ASTRAL symbolic model checker reported in [13], three approximation techniques to speed-up the model checking process for use in debugging a specification are presented. The techniques 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.

Keywords
Formal methods, Formal specification and verification, Real-time systems, Timing requirements, Model checking, State machines, ASTRAL.

1 Introduction

ASTRAL is a high-level formal specification language for real-time systems. It includes structuring mechanisms that allow one to build modularized specifications of complex systems with layering [9]. It has been successfully used to specify a number of interesting real-time systems [1, 2, 9, 10, 11, 12]. The ASTRAL Software Development Environment (SDE) [20, 22] is an integrated set of design and analysis tools, which includes, among others, an explicit-state model checker, a symbolic model checker and a mechanical theorem prover. The explicit-state model checker [12, 22] 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. The symbolic model checker [13] tests specifications at the process level and requires only limited input to set up constant values. Its model checking procedure uses the Omega library [23] to perform image computations on the execution tree 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. The model checker traverses the tree in a depth-first manner and calculates the preimage and postimage for each node on the current path. In [13], the symbolic model checker was used to test a railroad crossing benchmark. In those experiments the model checker aborted before completion for two of the test cases due to the extremely large size of the specification instances. Because the model checkers in the ASTRAL SDE are only intended to be used for debugging purposes, it is reasonable to use lower approximation techniques that allow the search procedure to complete, while still remaining effective in finding violations. Although the lower approximation techniques calculate only a subset of the reachable states, these techniques will not cause false negativities. This is because the properties specified using ASTRAL are essentially safety properties.

In this paper, three techniques to meet this need are introduced. They are random walk, partial image and dynamic environment generation. The idea of random walk techniques and partial image techniques is not new, although we are not aware of their use in symbolic model checking. The name "random walk" is borrowed from the theory of stochastic processes. This technique is used to allow the model checker to randomly skip a number of branches when traversing the execution tree. The partial image technique is inspired by sampling and random testing methods [16, 15] in software testing. However, instead of picking a single sample from the domain, the partial image technique selects a subset of the preimage and uses this subset to calculate the postimage at each node. The dynamic environment generation technique [14] generates a different sequence of imported variable values for different execution paths. It is similar to the idea of Colby, Godefroid and Jagadeesan [8] 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 (written in C) and is based upon static analysis of a program to translate it into

a self-executable closed form. By considering real-time specifications, the approach presented in this paper dynamically selects a reasonable environment according to the imported variable clause and the previous environment. The reason for considering the previous environment is that, as stated later, ASTRAL is a history-dependent specification language. As a case study, ten mutation tests [21] of a railroad crossing benchmark are used to show their effectiveness in finding bugs, and the performance of the model checker is compared when the three approximation techniques are used separately and in combination.

In [4, 5] Bultan used the Omega library as a tool to symbolically represent a set of states that is characterized by a Presburger formula. He also investigated partitions and approximations in order to calculate fixed points. As in the work reported in this paper, Bultan worked with infinite state systems. However, the systems Bultan 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, Bultan's approaches can be considered to be 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, 6]. However, STeP primarily uses a theorem prover to validate a property while the approach presented here uses a fully automatic model checker.

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 ASTRAL symbolic model checker and three approximation techniques are presented. Section 4 gives the results of using the techniques separately and in combination on ten mutation tests, and it analyzes the results. Finally, in section 5, conclusions are drawn from this work, and future areas of research are proposed.

2 Overview of ASTRAL

A railroad crossing specification is used as a benchmark example throughout the remainder of this paper. The system description, which is taken from [19], 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 requirements of the system are 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 at `http://www.cs.ucsb.edu/~dang`.

An ASTRAL system 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. Each process specification consists of a sequence of levels, with the highest level being an abstract view of the process being specified. ASTRAL semantics are formally defined for both dense time and discrete time. The current version of the model checker, however, only supports a discrete time ASTRAL computation model.

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 is one process instance of type `Gate` and `n_tracks` process instances of type `Sensor` in the system, where `n_tracks` is a global constant of type `pos_integer`. In ASTRAL, primitive types include `Integer, Real, Boolean, ID` and `Time`. Additional types can be declared by using the `TYPEDEF` construct. For instance, `pos_integer` is defined as follows:

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

Each process instance has a unique identifier with type `ID`. The ASTRAL specification function `IDTYPE(i)` represents the type of the process with the identifier `i`. For instance, the global declaration

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

represents all identifiers of process instances of type `Sensor`. In the railroad crossing specification there are two process specifications `Gate` and `Sensor`, which cor-

respond to the two process types declared in the global specification. 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, as well as local constants, must be declared in each process specification. 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 in that process instance. This is discussed further in the next subsection.

Transitions

The ASTRAL computation model is defined by the execution of state transitions, which are specified inside process specifications. Each transition in a process instance can only change the variables specified in that instance. The body of an ASTRAL transition includes pairs of entry and exit assertions with a nonzero duration associated with 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`). `Start(T)` and `End(T)` specify the last start and end time of a transition `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, for instance `up_dur` above. 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 the execution finishes. Other process instances may refer to these variables as well as to the start and end times of transitions under the assumption that these variables and transitions are explicitly exported and the process instances properly import them. If it is the case that there is more than one transition instance that is enabled inside the same process instance and no other transition is executing, then one of the enabled transitions is nondeterministically chosen to fire. Inside a process instance, executions of transitions are non-overlapping interleaved, while between process instances, maximal parallelism is supported. Thus, the execution of transition instances in different process instances is truly concurrent.

Assumptions and Critical Requirements

Besides 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. Axioms are used to specify properties about constants. An initial clause defines the system state 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. 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 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 additional 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 [9, 10, 22].

Modularized Proof Theory

In this paper, modularization means the principle that a system specification can be broken into several loosely independent functional modules. Although most high level specification languages support modularization, each module in the specification is only a syntactical module. That is, these languages provide a way to write a specification as several modules, however, there is often no way to verify the correctness of each process without looking at all the behaviors of all the other processes. 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 greatly eases both verification and design work.

In ASTRAL, a process instance is considered as a module. It provides an interface section including an imported variable clause, which is an ASTRAL well-formed formula, that can be regarded as an abstraction of the behaviors of the other processes. This fea-

ture helps to develop a modular verification theory that does not exist in any other infinite state real-time system model checking approach. For example, verifying the schedule and the invariant of each process instance uses only the process's local assumptions and behaviors. Thus, verifying the local invariant uses only the behaviors of transitions of the process instance, and verifying the local schedule uses the process's local environment and imported variable clause, plus the behaviors of the process's transitions. Finally, because the imported variable clause must be a correct assumption, it needs to be verified by combining all the invariants from all the other process instances. At the global level, the global invariant of an ASTRAL specification can be verified by using only the invariants for all process instances, without looking at the details of each process instance's behavior. Similarly, the global schedule can be verified by using only the global environment and the schedules for all process instances. Due to page limitations, the ASTRAL proof theory can not be presented in detail in this paper. The interested reader should see [10].

3 Approximation techniques for the ASTRAL symbolic model checker

In this section, an overview of the ASTRAL symbolic model checker is given along with the motivation for introducing the approximation techniques. Next, each of the techniques is formulated in more detail.

An overview of the ASTRAL symbolic model checker

A prototype implementation of the ASTRAL symbolic model checker for a nontrivial subset of ASTRAL is given in [13]. This prototype uses the Omega library [23] as a tool to symbolically represent a set of states that are characterized by a Presburger formula, which is an arithmetic formula over integer variables that is built from logical connectives and quantifiers. The Omega library provides rich operations on Omega sets and relations, such as join, intersection and projection. These operations are used in the image computations in the symbolic model checker.

The symbolic model checker presented in this paper is implemented as a process level model checker, based upon the modularized ASTRAL proof theory. For each process type that is globally declared, one only needs to check one process instance's critical requirements. Each ASTRAL process declaration $\mathcal{P}$ can be translated into 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. Σ consists of all the transition names declared in $\mathcal{P}$ as well as two special transitions *idle* and *initial*. Each $\rightarrow_a$ is a relation on Q, i.e., $\rightarrow_a \subseteq Q \times Q$. $\mathbf{Init} \subseteq Q$ denotes the initial states. The assumption **Assump** and property **Prop** of $\mathcal{T}$ are also subsets of states Q. $\mathcal{T}$ is further restricted to have a form in which the components Q, **Init**, $\rightarrow_{a\in\Sigma}$, **Assump** and **Prop** are Presburger formulas. As usual, for a set of states $R \subseteq Q$, one can denote 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: $Pre_a(R) = \{q : \exists q' \in R \text{ s.t. } q \rightarrow_a q'\}$. 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, $q_i \rightarrow_{a_{i+1}} q_{i+1}$ and $q_0 \in \mathbf{Init}$. $\mathcal{T}$ 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, $\{q_0, \cdots, q_k\} \subseteq \mathbf{Assump}$ implies $q_k \in \mathbf{Prop}$.

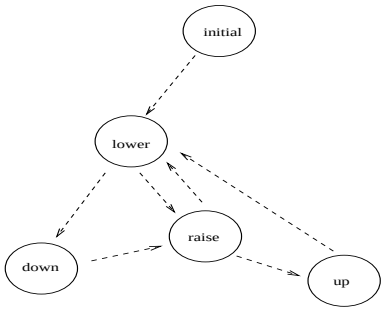

Figure 1: The execution graph of `Gate`

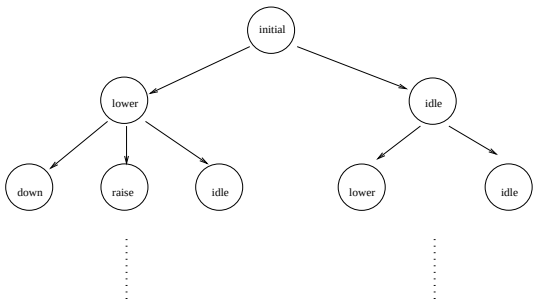

Figure 2: The execution tree of `Gate`

The model checking procedure starts by constructing the execution graph G of $\mathcal{P}$. The graph G is a pair $\langle V_G, R_G \rangle$ in which $V_G = \{initial, idle, T_1, \cdots, T_k\}$, where T_i represents a transition declared in $\mathcal{P}$. *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 and no transition is currently executing. *idle* will not change the values of any local variables. $R_G \subseteq V_G \times V_G$ excludes all the pairs of transitions such that the second transition is not immediately firable after the first one finishes. G is automatically constructed using the Omega library

by analyzing the initial conditions and the entry and exit assertions of each ASTRAL transition in the process. Figure 1 is an example of the execution graph for the `Gate` process in the railroad crossing specification. A dashed arrow in Figure 1 means that zero or more *idle* transitions are executed to reach the next node. The model checking procedure is carried out on the tree of all possible execution paths trimmed by the execution graph G. Figure 2 is part of the execution tree for the `Gate` process. Starting from the initial node *initial* in the execution tree of $\mathcal{P}$, the model checker calculates the image of the reachable states on every node along each path up to a user-assigned search depth. Each image is checked against the assumption **Assump** and the property **Prop** in order to detect potential errors. A number of techniques are also used to dynamically resolve the values of variables according to the path that is being searched. These techniques reduce the number of variables used in the actual image calculation. Whenever an error is found, the model checker generates a concrete specification level trace leading to this error.

Three approximation techniques

In this subsection, three approximation techniques are presented to speed up the ASTRAL symbolic model checker. The techniques are random walk, partial image and dynamic environment generation.

Motivation for introducing the approximation techniques

As mentioned earlier, the ASTRAL symbolic model checker performs process level model checking by checking only one process instance for each process type declared in the global system. The correctness of doing this is ensured by the modularized proof theory of ASTRAL. Without looking at the global behaviors of the entire system, the performance of the model checker can be greatly increased, since dealing with a single process instance is much easier. However, this does not mean that a single process instance is necessarily simple. In [13], the model checker was used to test the railroad crossing benchmark. In those experiments the model checker failed to complete two of the test cases due to the extremely large size of the instances. 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. The 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 in ASTRAL is modularized, each process instance does not stand alone. An environment assumption is typically used to characterize the pattern of invocations (`calls`) of exported transitions from the outside environment. A process instance may also interact with other process instances through imported variables that are exported from other process instances. Since a process's local properties are proved using only its local assumptions, the process instance must specify strong enough assumptions (environment clause and imported variable clause) 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 the history-dependency of ASTRAL, which expresses that a system's current state depends upon its past states.

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 [13], 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 checker 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 [12], should be used in 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. 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.

Random walk

A path in the execution tree of an ASTRAL process is a sequence of transitions. Each node in the tree containing the image of all reachable states from the initial node along the path. Theoretically, the number of paths

is exponential in the user-assigned search depth. Even though the symbolic model checker itself adopts a number of trimming techniques [13], 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, since ASTRAL is history dependent.[1]), (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 to demonstrate 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. Figure 3 shows the recursive procedure, which is based upon depth-first search. The algorithm is similar to the procedure proposed in [13] except that this algorithm includes a random choice when the model checker moves from one node to its children. In the algorithm, *depth* indicates the maximal number of iterations of transitions to check. $Post_A$, which was defined earlier, is the postimage 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 is firable, then the preimage is checked with respect to the property, followed by checking every child node according to the execution graph [2] and the result of the random boolean function $toss(A)$. The function *toss* is not symmetric. The probability of result *tail* is chosen as

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

where $numChildren(A)$ indicates the number of successors of node A in the execution graph G, i.e., $numChildren(A) = |\{D : \langle A, D\rangle \in R_G\}|$, and $A.layer$ indicates the layer where node A is located in the execution tree. The reason for this choice is to ensure the following: (1) A short violation has less chance of being missed. When $A.layer$ is small, the probability of result *tail* is large. When $A.layer$ 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. (2) When $numChildren(A)$ is 1 (it is always at least one, since each node has a successor through the *idle* transition.), the probability of result *tail* is 1. That is, a node with only one successor can not be skipped.

The model checking procedure starts from the initial node *initial*, Check(*initial*, *depth*).

```
Boolean Check(Node A, int depth)
{
    if A.layer = depth then return true;
    if A.layer = 0 then
        A.postimage = Init ∧ Assump;
    else
        A.preimage = A.parent → postimage;
        A.postimage = Post_A(A.preimage) ∧ Assump;
    if A.postimage ≠ ∅ then
        if(A.preimage ⊈ Prop) then return false;
        else for each B, ⟨A, B⟩ ∈ R_G and toss(A) = tail
            if(¬Check(B, depth)) return false;
    return true;
}
```

Figure 3: The model checking algorithm with random walk

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 [13], 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 in this paper is used without fixing any constant values, by applying the $PartialImage()$ operator, which returns only half of the unions for the image. The algorithm presented in Figure 4 is essentially the same as the one in [13] except that during the depth first search the $PartialImage()$ operator is applied on each preimage on node A. This reduced preimage represents the set of reachable states at the node. The approximated image is then used to calculate the postimage of the node, as stated in the algorithm. In previous experiments [13] the two test cases where the symbolic model checker failed to complete the search procedure were due to the extremely large size of the instance. The large number of constants and variables used in the test cases resulted in an extremely long time (hours as observed in [13]) for a single image computation. When represented in the Omega library, these computations usually

[1] This is significantly different from some standard techniques used in finite state model checking, such as the partial order method [18].

[2] In the actual implementation, when A is *idle*, if the nearest non-idle ancestor node of A is A', then a non-idle child node B of A with $\langle A', B\rangle \notin R_G$ is not checked. That is, only a non-idle child which is reachable from the closest non-idle ancestor of A in the execution graph is checked.

involve images containing hundreds or even thousands of unions of convex regions. Thus, it is natural to cut the image size of the reachable states at each node.

```
Boolean Check(Node A, int depth)
{
    if A.layer = depth then return true;
    if A.layer = 0 then
        A.postimage = Init ∧ Assump;
    else
        A.preimage = A.parent → postimage;
        A.preimage = PartialImage(A.preimage);
        A.postimage = Post_A(A.preimage) ∧ Assump;
    if A.postimage ≠ ∅ then
        if(A.preimage ⊈ Prop) then return false;
        else for each B, ⟨A, B⟩ ∈ R_G
            if(¬Check(B, depth)) return false;
    return true;
}
```

Figure 4: The model checking algorithm with partial image

Dynamic environment generation

The dynamic environment generation technique used in this paper is proposed in detail in [14]. ASTRAL is a history dependent specification language. A technique is needed to encode the history of an imported variable when its past values are referenced, since as pointed out in [13], it is too costly to encode the entire history. Therefore, a limited window size technique is proposed in that paper to approximate the entire history by only a part of it. For example, a window size of two means that the process instance can only remember an imported variable's last two change times, and the values before and after the changes. As observed in [14], the imported variables and their history encodings are the main bottleneck of the symbolic model checker, due to the extra variables introduced for each imported variable. Thus, an environment is characterized by all the imported variables and their histories inside a given window. If every variable in the environment has a concrete value, then the environment is called concrete. The dynamic environment generation technique effectively generates a reasonable sequence of concrete environments for each execution path. The sequence is selected according to the imported variable clause and the previous environment. The reader is referred to [14] for the details of the algorithm.

The techniques used in combination

The three techniques mentioned above can also be used in combination in a straightforward manner. For example, random walk and partial image can be combined in such a way that the model checker propagates only part of the reachable states to the children nodes while it randomly skips a number of branches. Random walk and dynamic environment generation can also work together such that along each randomly chosen execution path a sequence of concrete environments are generated. Similarly, partial image and dynamic environment generation can be applied when a part of the image of reachable states is used to calculate the postimage and they are also used to generate the concrete environments. In the following section, the results of running ten mutation tests of the `Gate` process using the model checker with each of the techniques and their combinations are presented.

4 Performance comparisons: a case study

All three approximation techniques are integrated into the ASTRAL symbolic model checker. 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. To demonstrate the effectiveness of the approximation techniques proposed in the last section, the model checker was run on ten mutations [21] 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 previously failed to complete [13], when not using approximation techniques. Each mutation contains a minor change to the original specification, which has been proved to be correct using the ASTRAL theorem prover, which is also part of the ASTRAL SDE [22]. A detailed list of all the mutations can be found in Table 1. As pointed out in [21], the mutation techniques can be used in two ways for formal specifications: they can help a user understand the specification, and they can test the strength of a specification. 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 he or she believes that such a change should affect the behavior of the system. 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.

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 history window size

was chosen as 2.[3] There were no other user-assigned constants. The maximal search depth was 10. The reason that `n_tracks` was chosen to be 2 is that this setting demonstrates a large instance with 23 variables for the model checker.

M1	delete the 1st conjunction from the axiom of `GATE`
M2	delete the 2nd conjunction from the axiom of `GATE`
M3	delete the 3rd conjunction from the axiom of `GATE`
M4	delete the term `raise_dur` from the 2nd conjunction of the axiom of `GATE`
M5	delete the term `up_dur` from the 3rd conjunction of the axiom of `GATE`
M6	delete `now-Change(s.train_in_R)>=RImax-response_time` from the 1st conjunction of the schedule of `GATE`
M7	delete the imported variable clause 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 | position=raised)` from the entry assertion of transition `raise`

Table 1. Ten mutations of the railroad crossing specification

Table 2 and Table 3 show the results with the three approximation techniques used separately and used in combination, respectively. In the tables, each result contains the number of nodes visited in the execution tree, the time taken (measured in seconds), and the result status. The status values are "$\times$" (i.e., the model checker is able to detect a violation), "$\surd$" (i.e., the model checker finishes and reports no error), or "$\uparrow$" (i.e., the model checker fails to finish in less than four hours.). A number is also attached to the status value to indicate the actual number of runs of the specific tests that were performed, where one run differs from another due to the randomness in the approximation algorithms. For example, "$\times(2)$" means a violation is found after the second run and within the first run no error was detected. In this case, the number of nodes and the time taken are the sum of the two runs. For each case at most two trials were made. For M3, M5, M8 and M9, the model checker ran only once. The reason is that the model checker will not report any errors for these cases, as discussed below. For comparison, all the mutations were also run using the earlier symbolic model checker that did not use the approximation techniques. The results of these runs are shown under column "plain" in Table 2. All tests were performed on a Sun Ultra 1 with 64M main memory and 124M swap memory.

As observed in [14], among the ten mutations, M8 and M9 are both correct. Hence the model checker should not report any error for these cases. M3 and M5 are two cases that demonstrate a limitation of the symbolic model checker when it fails to detect an error although the mutations should be killed. In [14], the explicit state model checker [12] was used to successively find the violations under a set of constant values provided by the specifier. Test results on these live mutants are still meaningful in that they can be used to show the node coverage of each approximation technique when the model checker completes the search procedure. The remaining six mutations are the ones that the model checker is able to kill. They are used to demonstrate the effectiveness of using the model checker to debug a specification.

A number of observations can be made from the results shown in Table 2 and Table 3. During the first run, all six mutations were killed by using partial image and dynamic environment generation separately. After the second run, partial image combined with dynamic environment generation was also able to kill all six mutations. Random walk, either applied separately or combined with the other two techniques, was not able to kill the six mutations. M6, M7 and M10 show relatively short violation traces, while M1, M2 and M4 show long violation traces. Therefore, we are more interested in the latter three cases. In the first run, random walk was able to kill three mutations on average. After the second run, however, it could not kill M1 for each of its three uses. The time used in finding an error is 2 to 4 times faster on average than the time without using approximation techniques. Even when the error was detected after the second run, the total time used in the two runs is still 2 times faster on average. One can also notice that using the techniques in combination could speed up the procedure further, though the ratio is not especially high, and on occasion it is worse. The reason is that using the techniques in combination sacrifices more coverage. Therefore, it has less chance of detecting an error in the first run. Consider the results on the live mutants M3, M5, M8 and M9 mentioned above. On these mutations, random walk has the least node coverage especially when applied in combination with the other two techniques. In contrast, both partial image and dynamic environment generation have much higher node coverage, even when applied in combination. However, it is unknown what the total number of reachable nodes is, since the symbolic model checker failed to complete the searching procedure for all four of these mutations. As shown in Table 3, when combining all the three techniques, the model checker can only kill M6 and M10 in two runs. The reason is that such approximation is too aggressive to be able to kill other mutations with longer violation traces.

From the above analysis, one can conclude that all of the approximation approaches are effective. They are able to kill at least half of the mutations in a much shorter time during the first run, while they can finish the procedure in two hours or less for live mutants. For this specific set of tests, partial image and dynamic en-

[3] As pointed out in [13], a window size of 2 is sufficient for this specification; i.e., for a specification instance, the current state never depends on the history for more than the last two changes of each local or imported variable.

cases	partial image (p.i.)			dynamic env (d.e.)			random walk (r.w.)			plain		
	nodes	time	result	nodes	time	result	nodes	time	result	nodes	time	result
M1	23	1,099	×(1)	23	1,610	×(1)	91	4,196	√(2)	23	6,351	×(1)
M2	23	1,276	×(1)	23	1,617	×(1)	121	4,549	×(2)	23	7,012	×(1)
M3	196	2,938	√(1)	256	4,610	√(1)	111	4,389	√(1)	140	24,322	↑(1)
M4	28	1,694	×(1)	23	1,726	×(1)	104	5,069	√(2)	23	10,168	×(1)
M5	201	3,185	√(1)	375	7,221	√(1)	69	3,150	√(1)	147	28,987	↑(1)
M6	7	1,053	×(1)	7	1,045	×(1)	7	835	×(1)	7	1,693	×(1)
M7	10	939	×(1)	10	962	×(1)	16	1,293	×(1)	10	941	×(1)
M8	121	2,403	√(1)	137	3,111	√(1)	91	3,173	√(1)	62	13,008	↑(1)
M9	421	4,892	√(1)	326	6,054	√(1)	143	3,290	√(1)	79	14,945	↑(1)
M10	5	566	×(1)	5	132	×(1)	5	645	×(1)	5	1,066	×(1)

Table 2. Experiments: the approximation techniques used separately

cases	r.w. + d.e.			r.w. + p.i.			p.i. + d.e.			r.w. + p.i. + d.e.		
	nodes	time	result	nodes	time	result	nodes	time	result	nodes	time	result
M1	124	4,450	√(2)	30	1,679	√(2)	23	1,376	×(1)	30	1,596	√(2)
M2	130	3,231	√(2)	47	1,902	×(2)	149	2,829	×(2)	103	2,314	√(2)
M3	54	2,274	√(1)	23	1,372	√(1)	146	2,549	√(1)	27	1,790	√(1)
M4	69	2,428	×(2)	74	2,249	√(2)	23	1,709	×(1)	39	1,836	√(2)
M5	121	2,964	√(1)	67	2,072	√(1)	186	2,735	√(1)	36	1,540	√(1)
M6	7	1,057	×(1)	7	1,107	×(1)	22	1,716	×(1)	7	949	×(1)
M7	53	3,411	×(2)	10	1,001	×(1)	131	1,339	×(2)	52	1,118	√(2)
M8	59	2,151	√(1)	30	1,450	√(1)	172	2,730	√(1)	37	1,417	√(1)
M9	111	2,942	√(1)	50	1,625	√(1)	118	2,025	√(1)	59	2,024	√(1)
M10	10	626	×(1)	5	598	×(1)	5	129	×(1)	44	1,300	×(2)

Table 3. Experiments: the approximation techniques used in combination

vironment generation are the most effective. They are fast to detect an error, and when no error is reported, they attain a high node coverage. Random walk performs slightly worse than the other two techniques. The reason is that once a branch is skipped the whole subtree rooted at the branch is trimmed from the execution tree. As shown by the results on the live mutations, the node coverage for random walk is much lower than for the other two techniques. Along each execution path, the image calculations are still very expensive since random walk propagates a full reachable image. However, compared to the model checker without approximation techniques, the performance in detecting a violation is still much faster.

5 Conclusions and Future Work

In this paper, three approximation techniques for using the ASTRAL symbolic model checker as a specification debugger were introduced. The techniques 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 considers only a subset of the image and uses this subset to calculate the postimage at each node. The dynamic environment generation technique [14] generates different sequences of imported variable values for different execution paths. The three techniques were applied separately and in combination to ten mutation tests on the `Gate` process in the railroad crossing benchmark specification. All the techniques are effective in finding bugs.

Besides the techniques discussed in this paper, we believe that there are many other applicable approximation approaches. Unlike other model checkers, the ASTRAL model checker is primarily intended for use as a specification debugger. Once the fixed point computation is out of the question, numerous approximation techniques that already exist in the testing area can also be investigated. We believe the techniques proposed in this paper will also be useful in model checkers using different specification languages as long as only safety properties are considered. For debugging a general temporal property formulated in a temporal logic, it is still unknown how well these approximation approaches will work. This is an area for further research. The coverage analysis in this paper is empirical. The factors considered are time and number of nodes. Another 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 urgently needed.

As suggested by one of the referees, we reran a number of the experiments on a faster machine with 4 CPUs and 256M memory. Even though the experiments resulted in a 3 times speed up, the ratio between using and not using the approximation techniques was roughly the same. The authors would like to thank T. Bultan and P. Kolano for many insightful discussions. The ten mutations tested in this paper were created as a result of earlier discussions among T. Bultan, P. Kolano and the authors. The specification was written by P. Kolano.

REFERENCES

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

[2] 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, June 1991.

[3] T. Bultan, R. Gerber, and C. League, *"Verifying Systems with Integer Constraints and Boolean Predicates: A Composite Approach."* Proceedings of the 1998 ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98), 1998, pp. 113-123.

[4] T. Bultan, R. Gerber, and W. Pugh, *"Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetic,"* CAV'97, 1997, pp. 400-411.

[5] T. Bultan, R. Gerber, and W. Pugh, *"Model Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations and Experimental Results."* To appear in ACM Transactions on Programming Languages and Systems.

[6] N. S. Bjorner, Z. Manna, H. B. Sipma, and T. E. Uribe, *"Deductive Verification of Real-time Systems using STeP,"* 4th International AMAST Workshop on Real-time Systems, LNCS vol 1231, 1997, pp. 22-43.

[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."* CAV 96, LNCS vol. 1102, 1996, pp. 415-418.

[8] C. Colby, P. Godefroid and L. J. Jagadeesan, *"Automatically closing open reactive programs,"* Proceedings of 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices, vol. 33, No.5, 1998, pp. 345-357.

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

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

[11] 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.

[12] Z. Dang and R. A. Kemmerer, *"Using the ASTRAL model checker to analyze Mobile IP,"* Proc. of ICSE'99, 1999, pp. 132-141.

[13] Z. Dang and R. A. Kemmerer, *"A Symbolic Model Checker for Testing ASTRAL Real-time specifications,"* Proc. of RTCSA'99.

[14] Z. Dang and R. A. Kemmerer, *"Dynamic Environment Generations for an ASTRAL Process,"* Technical Report, Department of Computer Science, University of California, Santa Barbara, 2000.

[15] J. W. Duran and S. Ntafos, *"A report on random testing,"* Proc. of ICSE'81, 1981, pp. 179-183.

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

[17] P. Godefroid, *"Model checking for programming languages using VeriSoft,"* POPL 97, Paris, 1997.

[18] P. Godefroid and P. Wolper, *" A partial approach to model checking. "*, Information and Computation, 110(2) 1994, pp. 305-326.

[19] C. Heitmeyer and N. Lynch, *"The generalized railroad crossing: a case study in formal verification of real-time systems,"* Proc. of 15th Real-time Systems Symposium, 1994, pp. 120-131.

[20] P. Z. Kolano, *"Proof Assistance for Real-Time Systems Using an Interactive Theorem Prover,"* 5th International AMAST Workshop on Real-Time and Probabilistic Systems, LNCS Vol. 1601, pp. 315-333.

[21] R. Kemmerer, T. Bultan, P. Z. Kolano and Z. Dang, *" Mutation tests for ASTRAL real-time specifications,"* in preparation.

[22] 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.

[23] W. Pugh, *"The Omega test: a fast and practical integer programming algorithm for dependence analysis,"* Communications of the ACM, Vol. 8, 1992, pp. 102-104.