Dynamic Environment Generations for an ASTRAL Process

Zhe Dang and Richard A. Kemmerer

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

Abstract. 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 real-time systems with layering. Based upon the ASTRAL symbolic model checker reported in [DK99b], an approximation technique to speed-up the ASTRAL symbolic model checker for debugging a specification is presented. The technique, called dynamic environment generation, randomly generates a sequence of concrete environments for an ASTRAL process instance along each execution path in the execution tree of the ASTRAL process. Doing this greatly reduces the time for finding an error in a specification, as demonstrated by a number of mutation tests, while still ensuring reasonable coverage of the search procedure. The results of the tests show that the techniques presented in the paper are effective.

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[CGK97]. It has been successfully used to specify a number of interesting real-time systems [CKM94] [CGK97] [BCF91] [BBKT95] [DK97] [DK99a]. The ASTRAL Software Development Environment (SDE)[KDK99, K99] 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 [DK99a, KDK99] 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[DK99b] tests specifications at the process level and requires only limited input to set up constant values. Its model checking procedure uses the Omega library [P92] to do image computations on the execution tree of an ASTRAL process that is trimmed by the execution graph of the process. In [DK99b], 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 intended to be used only for debugging purposes, it would be desirable to have an approximation technique that could complete the search procedure while remaining effective in finding violations. In this paper, a dynamic environment generation technique coupled with partial image approximations is proposed to meet this need. Ten mutation tests[KBKD99] are used to show its effectiveness. The model checking procedure is similar to that in [DK99b] except that along each path in the execution tree a sequence of concrete environments is randomly generated. This greatly reduces the image size of the reachable node.

The approach proposed in this paper is similar to the idea of Colby, Godefroid and Jagadeesan[CGJ98] in that both address the problem of *automatically* closing an open system, in which some of the components are presented. However, their approach is only applicable to a trivial subset of ASTRAL specifications. The reasons are as follows. Systems considered in [CGJ98] are not realtime systems. In an ASTRAL process, however, history dependency (i.e., the current state depends upon the past states) can also present complex timing constraints for the behaviors of the process. Thus, nondeterministically selecting an environment can not ensure that the selected environment is reasonable when the history of the system is taken into account. The work of Bultan's [BGP97, BGP99] is most similar to ours in that he also uses the Omega library as a tool to symbolically represent a set of states that is characterized by a Presburger formula, which is an integer arithmetic formula with addition and quantification. Bultan investigates a number of partition and approximation techniques when the transition system is large and a fixed point calculation is not feasible. Because our use of the ASTRAL symbolic model checker is primarily as a debugger instead of a verifier, calculating the fixed point of a transition system is not an important issue. Therefore, we consider Bultan's approaches to be orthogonal to ours. The model checker considered in this paper is modularized [DK99b]; one need only check one process instance for each process type declared, without looking at the transition behaviors of other process instances. A modularized approach is also developed in Manna's STEP[CAV96, BMSU97]. However, STEP primarily uses a theorem prover to validate a property while the approach presented here uses a fully automatic model checker.

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 motivation and the algorithm for the dynamic environment generation are discussed. Section 4 gives some results of using the approach on the mutation tests performed on the railroad crossing specification. 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 is taken from [HL94]. 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 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; each level is an abstract view of the process being specified.

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.
```

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, and each process instance has a unique identifier. The ASTRAL specification function `IDTYPE(i)` returns the type of the process with the identifier `i`. The global declaration `sensor_id` 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 should 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 necessary share the same value. ASTRAL supports a modularized design principle: any variable is associated with a unique process instance, and changes to the variable can only be caused by the transitions specified inside the 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, and 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 nonzero 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. And `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. 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 the transitions are explicitly exported and the process instances properly import 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 the moment. 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.

2.3 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 usually 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. In ASTRAL environmental assumptions are used 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 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).

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 [CKM94, CGK97, KDK99].

2.4 Modularized Proof Theory

In this paper, modularization means the principle that a system specification can be broken into several loosely independent functional modules. This greatly eases both verification and design work. 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. The use of an imported variable clause is a unique feature in ASTRAL. This helps to develop a modular verification theory for real-time systems. 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. 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[CKM94]. For example, the global invariant of an ASTRAL specification can be verified by using only the invariants for each process instance, without looking at the details of each process instance's behavior. The global schedule can be verified by using only the global environment and the schedules in 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, the symbolic model checker is implemented as a process level model checker. That is, the model checker checks only a process instance's critical requirements.

3 Dynamic Environment Generation for an ASTRAL Process

In this section, an overview of the ASTRAL symbolic model checker[DK99b] is given along with the motivation for introducing the dynamic environment generation approach. Next the algorithm used to generate environments for an ASTRAL process is formulated. This is followed by discussions on a number of implementation issues.

3.1 An overview of the ASTRAL symbolic model checker

A prototype implementation of an ASTRAL symbolic model checker for a nontrivial subset of ASTRAL is given in [DK99b]. This prototype uses the Omega library[P92] as a tool to symbolically represent a set of states that are characterized by a Presburger formula [KK67], which is an arithmetic formula over integer variables that is built from logical connectives and quantifiers. The following is the grammar for generating Presburger formulas, which is adapted from [BGP97],

$$\begin{aligned} f &::= \; t \leq t \mid (f) \mid f \wedge f \mid \neg f \mid \exists \mathbf{intvar}(f), \\ t &::= \; (t) \mid t + t \mid t - t \mid \mathbf{intvar} \mid \mathbf{intcons}, \end{aligned}$$

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.

Even though the complexity of solving Presburger formulas is extremely high ($O(2^{2^{2^n}})$)[O78], for *practical* formulas, especially those with less alternations of quantifications, the solving time via the Omega library is affordable in many cases. By using the Omega library, the solution space of a Presburger formula can be compactly represented by a Omega set or relation that is a union of convex regions, where each region 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. 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[CKM94]. That is, for each process type that is globally declared, one only needs to check one process instance's critical requirements. Each ASTRAL process $\mathcal{P}$ declared can be effectively 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 [DK99b]. 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 the 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 which 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 $q_k \in \mathbf{Prop}$.

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 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 of 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. Whenever an error is found, the model checker generates a concrete trace leading to this error at the specification level.

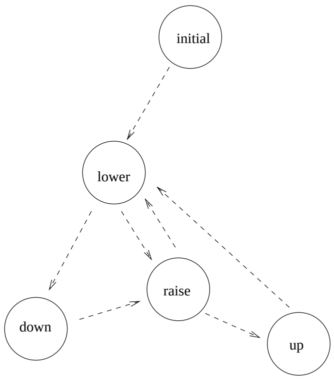

Fig. 1. The execution graph of `Gate`

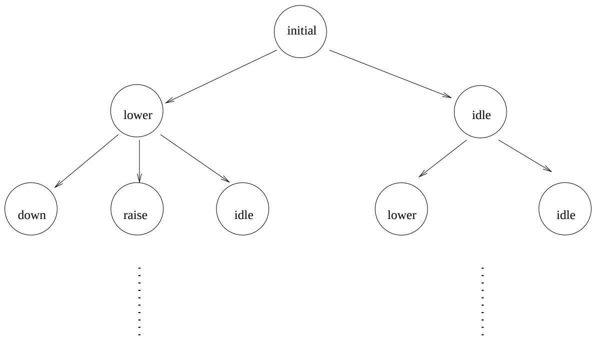

Fig. 2. The execution tree of `Gate`

3.2 Motivation for introducing the dynamic environment generation approach

The ASTRAL model checkers, including the explicit state exploration model checker[DK97, DK99a] and the symbolic model checker[DK99b], are primarily used as specification debugging tools. The explicit state model checker performs global model checking by exploring all possible behaviors of the specified system to a user-assigned time bound. It checks a concrete instance of the specification by setting up the values of all the constants defined in the specification. In contrast, the symbolic model checker performs process level model checking by symbolically exploring all possible behaviors of one process instance to a user-assigned depth. It requires little effort in setting up constant values. Earlier experiments[DK99b] have shown that the explicit state model checker is much faster than the symbolic one in finding a bug. However, it is inappropriate to conclude that the former is also more effective than the latter. The reason is that the explicit state model checker can only check a concrete instance of a specification. Therefore, it is necessary to set up all the constant values in the system, and picking the right set of constant values to cause an error to happen is not trivial. Thus, a natural next step would be to define an approximation approach to speed up the symbolic model checker in finding a bug while still freeing the user from fixing constant values. In earlier experiments, it was noticed that the performance of using the Omega Library is very sensitive to the number of variables and constants contained in a Presburger formula. Thus, a reasonable approximation should focus on how to reduce the number of variables and constants involved in the image computations. A naive idea would be to have the user assign concrete values to some of the constants before using the symbolic model checker. Indeed, as reported in [DK99b], doing this will speed up the model checker, and it is still effective in finding bugs in some cases. However, it is not reasonable for a user to do this except when the model checker fails to complete the searching procedure within a reasonable amount of time due to the extremely large size of the process instance being checked. The main reason for not having the user set the constants is that some constant value choices will miss a number of scenarios in which the specification would fail.

The approach proposed in this paper is to reduce the number of variables involved in the image computations. The variables in an ASTRAL process $\mathcal{P}$ consist of the local variables and the imported variables. The local variables can only be changed by firing a transition in $\mathcal{P}$. $\mathcal{P}$ is also allowed to reference a local variable's history. It is too expensive, however, to encode the entire history. Instead, the approach used dynamically resolves the symbolic values at each node along the path being searched. In addition to local variables, imported variables characterize the interface between $\mathcal{P}$ and other process instances in the system. The change patterns of the imported variables are typically characterized by the imported variable clause declared in $\mathcal{P}$. $\mathcal{P}$ can reference an imported variable's current value as well as its entire history. However, as was the case for local variables, it is too costly to encode the whole history of an imported variable. Instead, a limited window size technique[DK99b] is proposed to encode the history of an imported variable when the history of the variable is referenced. For example, a window size of two means that $\mathcal{P}$ can only remember an imported variable's last two change times, and the values before and after the changes. Thus, in this case one typically has to introduce 5 variables to encode the history of an imported variable. 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, which are exported by other process instances,
- 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. This paper focuses 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. This is discussed in detail in the next subsection.

3.3 The algorithm

Before presenting the algorithm it is necessary to define some notation. An environment E is characterized by all the imported variables and their histories inside a given window. If every variable in E has a concrete value, then E is called a *concrete environment*, otherwise E is a *variable environment*. Along an execution path of $\mathcal{P}$, each node may have distinct concrete environments reflecting the actual changes of the values of the imported variables with respect to the evolution of time. The sequence of these environments forms a sequence of *successive* concrete environments. Given a Presurger formula f (or a set of states represented by it), $f[E]$ denotes the formula f by replacing the variables in E with their corresponding values in E. Of course, if E is a variable environment, then these substitutions have no effect, i.e., $f[E] = f$. $\mathrm{ExampleEnv}(f)$ represents a concrete environment E such that $f[E]$ is satisfiable. This can be done by using the `example` operator in the Omega library. That is, given a concrete environment E_0, an environment E with respect to f and E_0, $\mathrm{ExampleSuccEnv}(f, E_0)$, is selected such that E is a successive environment of E_0 and $f[E]$ is satisfiable. Again, successiveness can be represented by a linear constraint. Therefore, $\mathrm{ExampleSuccEnv}(f, E_0)$ can be effectively computed. If $\mathrm{ExampleSuccEnv}(f, E_0)$ returns empty (i.e., no such example exists), then the return value is E_0. The notation is further abused by defining $\mathrm{ExampleSuccEnv}(f, E_0)$ for a variable environment E_0 as $\mathrm{ExampleEnv}(f)$. There should be no confusion by looking at the context.

The algorithm dynamically generates the concrete environments for $\mathcal{P}$. The following two features summarize the algorithm:

- different sequences of successive concrete environments are generated for different execution paths. Doing this ensures a wide coverage for the procedure,

– along each path, each node's environment is generated randomly. That is, when a node is entered, a coin is tossed. If the result is tails, then the concrete environment ExampleSuccEnv($\mathbf{Assump}, E_0$) is generated with respect to the last environment E_0 and the assumption **Assump** (which includes the imported variable clause). If the result is heads, the environment keeps E_0 unchanged. Bringing in the previous environment E_0 will produce more reasonable sequences of concrete environments, considering that ASTRAL is a history-dependent specification language. The randomness will get rid of some trivial sequences of environments in which either the imported variables always change too eagerly or too lazily.

Preliminary experiments show that both of these features are necessary to ensure reasonably large coverage of the execution paths, which is needed to detect a nontrivial bug in a specification.

```
Boolean Check(Node A, int depth, Environment E_0)
{
    Environment E_A = E_0;
    if A.layer = depth then return true;
    if A.layer = 0 then
        A.postimage = Init ∧ Assump;
    else
        if toss()=tail
            E_A = ExampleSuccEnv(Assump, E_0);
        A.preimage = A.parent → postimage;
        A.postimage = Post_A(A.preimage)[E_A] ∧ Assump[E_A];
    if A.postimage ≠ ∅ then
        if(A.preimage ⊈ Prop[E_A]) then return false;
        else foreach B, ⟨A, B⟩ ∈ R_G
            if(¬Check(B, depth, E_A)) return false;
    return true;
}
```

Fig. 3. The model checking algorithm with dynamic environment generations

Figure 3 shows the recursive procedure, which is based upon depth-first exploration. This procedure is similar to the procedure proposed in [DK99b] except that this algorithm includes the concrete environment generation in the image calculations. In the algorithm, *depth* indicates the maximal number of iterations of transitions to check. E_0 is used to indicate the previous environment. If the current node A is not the initial node, i.e., $A.layer \neq 0$, then a coin is tossed and a concrete environment E_A is generated randomly. After this point, E_A is either a successive environment of E_0 or the same as E_0. Once E_A is concrete, one can trace back along the current path to resolve the whole history of the environment as a sequence of concrete environments. This sequence is therefore used to calculate the substituted image $\mathbf{Assump}[E_A]$. The model checker then calculates the preimage and postimage of A by replacing the environment variables using the concrete environment E_A. 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 $initial$, Check($initial, depth, E$), in which E is a variable environment.

In the actual implementation, toss() is not symmetric. In order to ensure reasonable coverage, the probability that a tail occurs is chosen to be $A.layer/depth$. That is, in the algorithm, the probability that E_A is selected as a concrete successive environment of E_0 is proportional to the depth of node A. Doing this makes it more likely that short violations will not be missed, while for long violations a concrete environment will be generated and the image computations after that will be inexpensive.

A symbolic model checker using this algorithm has been implemented and integrated into the ASTRAL SDE[KDK99]. In the next subsection, a number of experiments using this approach on the railroad crossing specification are reported, and performance comparisons are made with the previous symbolic model checker, which did not use the dynamic environment generation.

3.4 Experimental results

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 its effectiveness, the model checker with dynamic environment generation was run on 10 mutations[KBKD99] on the `Gate` process in the railroad crossing specification. The reason that the `Gate` process specification was used is that it contains imported variables as well as their histories. Therefore, it can be used to demonstrate the usefulness of the dynamic environment generation technique. Each mutation contains a minor change to the original specification[1]. A detailed list of all the mutations can be found in Table 1. As pointed out in [KBKD99], the mutation techniques for formal specifications can be used in two ways: they can help a user understand the specification and they can test the strength of a specification. Experience shows that realtime 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 violation trace is demonstrated at the specification level. Reading through the trace helps the user quickly figure out where and how the 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

[1] The railroad crossing specification, which can be found at `http://www.cs.ucsb.edu/~dang`, has been proved to be correct using the ASTRAL theorem prover, which is also part of the ASTRAL SDE[KDK99].

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 comparison, all of the mutated specifications were also run using the earlier symbolic model checker that did not use dynamic environment generation. The results presented in Table 2 show that using the techniques proposed in this paper the model checker is able to find a violation in a much shorter time. For all of the tests, the constants `min_speed` and `max_speed` are set to be 15 and 20, respectively, and the constant `n_track` is set to be 2. The history window size is chosen as 2. There are no other user-assigned constants. The maximal search depth is 10. The reason that both `n_track` and the window size are chosen to be 2 is that this setting demonstrates the effectiveness of the model checker on an extremely large instance of the specification. In [DK99b], the model checker without dynamic environment generation failed to complete the search on large instances.

For each case in Table 2, results are shown both with dynamic environment generation (labeled as "dynamic on") and without it (labeled as "dynamic off"). Each result contains the number of nodes visited in the execution tree, the time taken (measured in seconds), and the result status. The result values are "error found" (i.e., the model checker is able to detect a violation), "no error found" (i.e., the model checker finishes and reports no error), or "abort"(i.e., the model checker fails to finish in a reasonable amount of time). All runs were performed on a Sun Ultra 1 with 64M main memory and 124M swap memory.

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 whole 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

cases	dymanic on			dymanic off		
	nodes	time	result	nodes	time	result
M1	23	1,376	error found	23	6,351	error found
M2	149	2,829	error found	23	7,012	error found
M3	146	2,549	no error found	140	24,322	abort
M4	23	1,709	error found	23	10,168	error found
M5	186	2,735	no error found	147	28,987	abort
M6	22	1,716	error found	7	1,693	error found
M7	131	1,339	error found	10	941	error found
M8	172	2,730	no error found	62	13,008	abort
M9	118	2,025	no error found	79	14,945	abort
M10	5	129	error found	5	1,066	error found

Table 2. Experiments on the benchmark with the ASTRAL symbolic model checker

A number of observations can be made from these preliminary experiments. First, as shown in Table 2 under "dynamic on", violations are found in six out of ten mutations. Among these six mutations, M1, M4, M6 and M10 were killed in the first run. During the first run, the model checker reports no errors for M2 and M7 after searching 126 nodes in 2138 seconds and 121 nodes in 1253 seconds, respectively. While in the second run, M2 and M7 quickly found errors after searching 23 nodes in 691 seconds and 10 nodes in 86 seconds, respectively. Thus, the data shown in the table under "dynamic on" are those of the first run except for M2 and M7. For these two cases the table shows the sum of the first two runs. It is also clear that compared to the model checker without the approximation technique applied, the time spent in finding an error is 2 to 4 times faster in most cases. M6 is an exception, since with the approximation applied, the model checker skips a quick error and has to search more states. Therefore, time spent in detecting an error is almost the same as that without using the approximation.

Unfortunately the model checker is not able to find any violations for M3, M5, M8 and M9. Without using the approximation technique, the model checker fails to complete the search (noted by "abort" in the table) for these cases. With dynamic environment generation, these cases are able to complete, yet no errors are reported. As mentioned before, these live mutants show either some weakness of the specification or the limitations of the model checker. The live mutants were analyzed by the specifier who quickly found the following facts. M3 and M5 are equivalent since the axioms in both of them, which are the only mutated parts in the specification (see Table 1), are equivalent. M8 and M9 are both correct, though they are not strictly equivalent to the original specification. For instance, M8 deletes the conjunction `now` − `End(lower)` >= `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`. By analyzing the specification it was possible to pick a scenario where M3 (hence M5) would fail by providing the right value for all of the constants. The scenario is as follows. During the execution of transition `up`, if the imported variable `s.train_in_R` changes to `true`, then the first conjunction of the schedule in the `Gate` process would be violated. The explicit state model checker[DK99a] was run under this set-up and a violation was quickly detected by searching 1952 states in 0.09 seconds. The violation was exactly the anticipated scenario. So, M3 (as well as the equivalent M5) demonstrates a limitation of this approach for symbolic model checking. As pointed out in [DK99b], approximations are used in several places in the implementation for the model checker. Thus, there is still a chance that a number of bugs are missed. One of these approximations is that a quantified form of **Assump** is used to ensure the assumptions hold constantly during the execution of a transition. The quantification is applied between the start time and the end time of the transition. This would cause false negatives when an imported variable changed between the start and end. But this approximation does not imply that the variable can only change at the end point of a transition. In fact, most traces of the violations found in this paper contain imported variables changes in the middle of a transition. Unfortunately, M3 and M5 fail to show any errors due to the limitation of the model checker.

In summary, the model checker with dynamic environment generation is able to kill six mutants out of eight that should be killed in relatively short time. In fact, four of them were killed during the first run.

4 Conclusions and Future Work

In this paper, an approximation technique to speed-up the ASTRAL symbolic model checker for debugging a specification was presented. The technique is based upon dynamic environment generation that randomly generates a sequence of concrete environments for an ASTRAL process instance along each execution path in the execution tree. Doing this greatly reduced the time for finding an error in a specification for a set of mutation tests, while ensuring reasonable coverage of the search procedure. The results of the tests show that the techniques used in the paper are effective. It was pointed out that there are two equivalent cases for which the model checker fails to find errors, even though it should. These two cases illustrate a limitation of the symbolic model checker.

Unlike most 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 already existing in the testing area are applicable. Thus, performance comparisons of various techniques are necessary in order to conclude the effectiveness of using them separately or in combination. An immediate comparison set would contain random walk[KBKD99], partial image, and dynamic environment generation. Another issue to be investigated is what metrics can be used to measure path and/or node coverage for a specific approximation technique applied on an execution tree. In this paper, the effectiveness is shown by only considering time as the primarily factor. A suitable metric should also give a user a quantized level of confidence when a specification survives the model checking.

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 discussions among T. Bultan, P. Kolano and the authors before this paper was written. The benchmark specification was written by P. Kolano.

References

[BBKT95] 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.

[BCF91] 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.

[BGL98] 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.

[BGP97] T. Bultan, R. Gerber, and W. Pugh, *"Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetic,"* CAV'97, 1997, pp. 400-411.

[BGP99] 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.

[BMSU97] 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.

[CAV96] 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.

[CGJ98] 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.

[CGK97] 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.

[CKM94] 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.

[DK97] 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.

[DK99a] Z. Dang and R. A. Kemmerer, *"Using the ASTRAL model checker to analyze Mobile IP,"* Proc. of ICSE'99, 1999, pp. 132-141.

[DK99b] Z. Dang and R. A. Kemmerer, *"A Symbolic Model Checker for Testing ASTRAL Real-time Specifications,"* RTCSA'99.

[HL94] C. Heitmeyer and N. Lynch, *"The generalized railroad crossing: a case study in formal verification of real-time systems,"* Proceedings of 15th Real-time Systems Symposium, 1994, pp. 120-131.

[K99] 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.

[KBKD99] R. Kemmerer, T. Bultan, P. Z. Kolano and Z. Dang, *" Mutation tests for ASTRAL realtime specifications,"* in preparation.

[KDK99] 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.

[KK67] G. Kreisel and J. L. Krevine, *"Elements of Mathematical Logic,"* North-Holland, 1967.

[O78] D. C. Oppen, *"A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic,"* Journal of Computer and System Sciences, Vol. 16, 1978, pp. 323-332.

[P92] W. Pugh, *"The Omega test: a fast and practical integer programming algorithm for dependence analysis,"* Communications of the ACM, Vol. 8, pp. 102-104, 1992.

This article was processed using the LaTeX macro package with LLNCS style