A Symbolic Model Checker for Testing ASTRAL Real-time Specifications

Zhe Dang Richard A. Kemmerer

Department of Computer Science
University of California
Santa Barbara, CA 93106
{dang,kemm}@cs.ucsb.edu

Abstract

ASTRAL is a high-level formal specification language for real-time (infinite state) systems. It is provided with structuring mechanisms that allow one to build modularized specifications of complex real-time systems with layering. In this paper, the methods and techniques used in the prototype implementation of the ASTRAL symbolic model checker, which is a component of the ASTRAL Software Development Environment(SDE), are presented. The model checking procedure uses the Omega library to represent a subset of states, and model checking is carried out on the execution tree of an ASTRAL process. The tree is further trimmed by the execution graph of the process. The model checker combines both explicit state exploration and symbolic state calculation in order to reduce the number of variables needed by dynamically resolving their values as well as their histories along a path of execution. Based upon the ASTRAL proof theory, the model checker is modularized, in the sense that each time it checks only one process instance of each process type that is globally declared. A limited window size technique is further proposed to encode the history of an imported variable when the history of the variable is referenced. The model checker is run on several earlier versions of the railroad crossing ASTRAL specification, which contained errors, as well as on the final version, which has been proved correct. The results show that it is effective for detecting bugs in an ASTRAL specification, which is extremely important in our use of the model checker as a specification debugger. The model checker is fully automated without manual abstractions.

1. Introduction

ASTRAL is a high-level formal specification language for real-time systems. It is provided with 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 (infinite state) systems [3,4,9,10,11,12]. The ASTRAL Software Development Environment (SDE) [15] is an integrated set of design and analysis tools, which includes, among others, an explicit-state exploration model checker, and a mechanical theorem prover. By generating customized C++ code for each specification, the model checker [12, 15] enumerates all the branches of execution of this implementation up to a system time bound set by the user. Strictly speaking, the model checker only tests the specification under a set of given constant values. This paper presents a fully automated symbolic model checker that tests the specification using the Omega library [19] and that requires only limited input to set up constant values.

Due to space limitations, the reader is assumed to know the basic concepts of ASTRAL and its modularized proof theory [10,11]. This paper is organized as follows. Section 2 discusses related work in this area. In section 3, the methods and techniques that are used in implementing the ASTRAL symbolic model checker are discussed. Section 4 gives some results on the experiments performed on the railroad crossing specification. Finally, in section 5, conclusions are drawn from this work.

2. Related work

The symbolic model checker in this paper handles a non-trivial subset of ASTRAL specifications that specify parameterized and history-dependent infinite state real-time systems. Parameterization and history-dependency, which are common in ASTRAL, have not been well studied in the model checking literature.

Parameterization in this paper has two meanings. One is that a constant is parameterized, e.g., the maximal size of a buffer *b_size*. The other is that the number of process instances in a system is parameterized, e.g., the number of senders in protocol *n_sender*. The former case has been investigated in several models. For instance, Alur and Henzinger [2] allow parameterized timing constraints in a timed automaton [1]. UPPAAL [17] also extends timed automata by introducing data variables. Under these extensions, only incomplete model checking procedures exist [2]. The latter case where the number of components is parameterized has been investigated by Manna et.al. [8]. The systems considered in this paper are parameterized in both senses. Parameterized transition durations are allowed, as well as parameterizing the number of process instances. The timing requirements used are far more complex than those in timed automata. In considering a parameterization of the number of process instances, the approach presented in this paper is similar to Manna's STEP in that it is modularized. However, STEP primarily uses a theorem prover to validate a property while the approach reported here uses a fully automatic model checker.

History-dependency means that a system's current state depends upon its past states. In reality, it seems that most practical real-time systems are history-independent. For example, a system updates its state according to its clock variable and environment stimulus. Because an ASTRAL specification is modularized, a process's local property can be proved using only its local assumptions including the imported variable clause. Thus, the imported variable clause of a process must specify assumptions about the imported variables that are strong enough to guarantee the local properties. It is not unusual for these assumptions to include complex timing requirements on the imported variables that reflect the patterns of their changes. Besides increasing the expressibility, history-dependency also makes model checking a real-time system with a parameterized number of process instances possible. That is, one need only model-check one process instance for each process type declared, without looking at the transition behaviors of other process instances. Modecharts [14], though provided with very limited support for history-dependency, can only deal with an unparameterized instance of the specified system. History-dependency brings new challenges into the model checking area. We are not aware of any model checking tool capable of handling this feature.

The model checker presented in this paper handles infinite state systems. The work of Bultan [5,6,7] is most similar to ours in that he also uses the Omega library [19] 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. The Omega library represents the solutions of a Presburger formula as a union of convex regions of linear constraints. Bultan investigates a number of partition and approximation techniques when the transition system is large and the 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. The examples that Bultan uses are also “simple”. For example, quantifications are only limited to a very small number. Also the examples are not real-time systems and are history-independent. Unfortunately, a typical ASTRAL specification, like the benchmark considered in this paper, is not as “simple” as mentioned above. Therefore, it is interesting to apply the Omega library to a non-trivial subset of ASTRAL to see how much more one can achieve with the tool.

3. Implementation of the Symbolic Model Checker

A railroad crossing specification is used as a benchmark example throughout the remainder of this paper. The system description is taken from [13]. 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`.

Based upon the ASTRAL modularized proof 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, using only the instance's local assumptions. This section presents a procedure to translate a process instance's local requirements, assumptions and transition system into Presburger formulas, whenever possible. Presburger formulas [16] are arithmetic formulas over integer variables, which are built from logical connectives and quantifiers. The following grammar for generating Presburger formulas is adapted from [6],

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

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

where **intvar** and **intcons** are integer variables and integer constants respectively. The complexity of solving Presburger formulas is extremely high ($O(2^{2^{2^n}})$) [18]. The Omega library was developed by Pugh [19] for manipulating integer tuple relations and sets that are characterized by Presburger formulas. The Omega library provides rich operations on Omega sets and relations, such as join, intersection and projection. For *practical* formulas, especially those with less alternations of quantifications, we found the solving time via the Omega library to be affordable in many cases. In this paper, the Omega library is used for dealing with a more complex real-time specification language than [6]. This experience showed that the Omega library usually can comfortably handle 10 or less integer variables. However, for a formula with more than 20 variables and several quantifications, it is not unusual that it either takes hours to solve the formula or 256M memory can be quickly exhausted. The following subsections focus on the methods and techniques used in the implementation of the ASTRAL symbolic model checker to encode a process instance using less variables.

3.1. Modeling

An ASTRAL process instance is modeled by a labeled transition system $\mathcal{T} = (Q, \mathbf{Init}, \rightarrow_{a \in \Sigma}, \mathbf{Assump}, \mathbf{Prop})$ that consists of a set Q of (infinitely many) states, a finite set of transitions $\rightarrow_a$ with name a from Σ. Each $\rightarrow_a \subseteq Q \times Q$ is a relation on Q. $\mathbf{Init} \subseteq Q$ are the initial states. The assumption $\mathbf{Assump}$ and the property $\mathbf{Prop}$ of $\mathcal{T}$ are also subsets of states Q. As usual, for a set of states $R \subseteq Q$, we define the preimage, $Pre_a(R)$, of a transition $\rightarrow_a$ as the set of all states from which a state in R can be reached by this transition. That is, $Pre_a(R) = \{q : \exists q' \in R \text{ s.t. } q \rightarrow_a q'\}$. Similarly, the postimage, $Post_a(R)$ is defined as $Post_a(R) = \{q : \exists q' \in R \text{ s.t. } q' \rightarrow_a q\}$. 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 $q_k \in \mathbf{Prop}$.

In this paper, since the Omega library is used to calculate the symbolic representation of a subset of the states, $\mathcal{T}$ is further restricted to have the form in which the components Q, $\mathbf{Init}$, $\rightarrow_{a \in \Sigma}$, $\mathbf{Assump}$ and $\mathbf{Prop}$ are Presburger formulas. It will be seen that such a restriction still includes a large collection of nontrivial ASTRAL specifications, including the benchmark discussed in this paper. In the following subsections, the translation of an ASTRAL process specification into a labeled transition system $\mathcal{T}$ is discussed.

3.2. Constants and Local variables

An ASTRAL process instance may use a number of global and local constants, as well as local and imported (from other process instances) variables. In the current implementation, they are translated into integer variables. Therefore, currently complex ASTRAL types like `List`, `Structure` or `Set` can not be handled. However, primitive types and `WFF`-types[1] based upon them can still be handled. For example, in the benchmark specification, the local variable `position` of the process `Gate` is an enumerated type `gate_position: (raised, raising, lowered, lowering)`. This variable is represented by an integer variable $position$ with the type assumption $position = 1 \vee position = 2 \vee position = 3 \vee position = 4$ added to $\mathbf{Assump}$ of $\mathcal{T}$. Also, real valued variables in the original specification are changed to integer valued ones. For example, the global constant `wait_time` is a `WFF`-type (already changed to integers) `pos_real : TYPEDEF i : integer` (`i` > 0), which is translated to an integer parameter constant $wait\_time$ with the type assumption $wait\_time > 0$ added to $\mathcal{T}$. Such a translation from real-valued constants to integer-valued ones is not safe in a strict sense. For instance, the property that reals are dense does not hold for integers. Unfortunately, it is theoretically impossible to construct a tool to verify that such a translation will not cause false negatives. This problem is also unavoidable when a real solver is used instead of the Omega library. In the specification considered in this paper, we believe that such a translation will not cause false negatives even though there are chances that some bugs could possibly be missed.

Some constant values should be known in advance in order to carry out the symbolic model checking procedure. There are two categories of these constants. One is the global constants that decide the number of process instances in the whole system, for instance, the number of `Sensor` process instances `n_tracks`. Another category is the constants involved in multiplications. For example, for the term `min_speed` $*$ `RIImin` which appears in process `Sensor`'s local axiom clause, one of the two constants needs to be set since multiplications between two variables are not allowed in a Presburger formula.

3.3. A Process Instance's Transition System

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

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

It can be translated as $\neg(ptrain\_in\_R = 1) \wedge train\_in\_R = 1 \wedge now = pnow + enter\_dur$ where $ptrain\_in\_R$ and $pnow$ indicate the values before `enter_R` fires, while $train\_in\_R$ and now indicate those after `enter_R` fires. A

[1] In ASTRAL, a `WFF`-type is constructed as a subset of a primitive type. This subset is characterized by a well-formed formula(wff) in ASTRAL.

straightforward way to do the model checking is to calculate the Omega representation of a one_step transition defined by $T_{\text{one-step}} = \bigvee_i T_i$. Next, starting from $\mathbf{Init}$, calculate a finite number of iterations R_i of $T_{\text{one-step}}$ and then check the result against $\mathbf{Prop}$ in order to find an error. However, from our experiments, this intuitive solution does not work for a nontrivial ASTRAL specification, like the one in this paper. The reasons are (1) histories of a number of local variables and imported variables must be encoded in $T_{\text{one-step}}$ and therefore, (2) each R_i calculated is so large that it can only be calculated and checked against $\mathbf{Prop}$ for very small i. In this paper, a method is proposed that is based upon the execution graph of the process instance's transition system. This method combines both explicit-state exploration and symbolic model-checking, which eliminates a number of variables and reduces the size of the formulas in each iteration step.

The method starts by defining the execution graph G of a process instance. The graph G is a pair $\langle V_G, R_G \rangle$ in which $V_G = \{initial, idle, T_1, \cdots, T_k\}$. T_i represents a transition defined in the ASTRAL process instance. $initial$ indicates the initial transition which is defined as an identity transition on the initial states with zero duration. $idle$ is a new transition introduced, which has duration one. $idle$ fires if no T_i is firable. $idle$ will not change the values of local variables. $R_G \subseteq V_G \times V_G$ excludes all the pairs of transitions such that the second one is not immediately firable after the first one finishes. G is automatically constructed by using the Omega library to analyze the initial conditions and the entry and exit assertions of each ASTRAL transition in the process. The model checking procedure is carried out on the tree of all possible execution paths trimmed by the execution graph G.

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

Figure 1. The model checking algorithm on the execution tree

Since, as will be discussed later, the use of the model checker in the ASTRAL SDE is only for debugging purposes rather than full verification, there is no interest in calculating the least fixed point for the transition system. Therefore, a user needs to set the depth of the tree indicating the maximal number of iterations of transitions to check. Figure 1 shows the recursive procedure, which is based upon depth-first exploration. In the algorithm, $Post_A$, 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 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. In fact, 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. The model checking procedure starts from the initial node $initial$, Check($initial, depth$). We have not discussed how each transition relation on node A can be translated in order to calculate the postimage $Post_A(A.preimage)$ in the algorithm. In fact, simply combining the translated entry and exit assertions is not enough. It is necessary to place some restrictions on the executions of a process instance's transition system based on the ASTRAL abstract computation model. The following subsections investigate some of these details, along with some benefits resulting from basing this model checking procedure on the execution tree.

3.4. Local, Imported and Exported Transitions

Let `T` be a local ASTRAL transition. On any node in the execution tree, the expression `Start(T,t)` can be dynamically translated as `LastStarT` $= t$ during the model checking, where `LastStarT` is a string representing the symbolic value of the most recent start time of the transition `T`. By looking backwards from the current node to the root (the $initial$ node), one can explicitly resolve the string `LastStarT` since each node along the path has a symbolic value for the duration. `End(T,t)` can be translated similarly. By doing this one does not need to introduce a new variable to record the most recent `Start` (or `End`) times of a local transition when `Start(T,t)` (or `End(T,t)`) is referenced.

The interface section of a process instance may also contain imported transitions (exported from other process instances) and their `Start`, `End` and `Call` times. When a process instance refers to one of these times, a new variable is introduced to indicate the last `Start` time of an imported transition. This variable's value can be changed during the importing transition's execution, and constraints need to be added to the transition relation. For example, one of the constraints is that, at the time when a transition T is completed, the new value of the last Start time of an imported transition can be either the same as the old value or should be greater than T's start time and less than or equal to the current time.

A local transition T can also be exported. Once T is an exported transition, T must be called by the external environment in order to fire. Calls on T are totally controlled by the external environment, which is typically restricted by a process's local environment clause. Successive calls are not effective if the called transition has not fired in response to the first call. In order to characterize the behavior of an exported transition, it is necessary to introduce a new variable to indicate the last call time. For example, in process Sensor, the exported transition enter_R's last call time would be indicated as *enter_R_lcl*. Constraints also need to be added to the translated Omega relation of enter_R: *penter_R_lcl* > plastStartenter_R. The constraint says that before the transition fires, the last call time of enter_R must be greater than the last start time of enter_R (this time is indicated by a symbolic string plastStartenter_R, which can be dynamically resolved). Furthermore, the variable *enter_R_lcl* can be changed during a transition's execution and constraints about these changes are also added to a transition relation.

3.5. Imported and Local Variables

This is one of the hardest parts to implement in the model checker. An ASTRAL process instance interacts with other process instances only through its interface section, which contains a number of imported variables (exported from other process instances). Once a variable is imported, the process instance may refer to the variable's current value as well as to its whole history. For example, in process Gate, the current value of imported variable s.train_in_R (where s indicates some sensor process's id) is referred to in the schedule clause as s.train_in_R, the past value at some past time t is $\mathtt{past}(\mathtt{s.train\_in\_R}, \mathtt{t})$, and the last time it changed is $\mathtt{change}(\mathtt{s.train\_in\_R})$. These history-dependent features greatly expand the expressibility of ASTRAL as a real-time specification language, while it makes model checking the language more difficult. A naive approach to handle the history of an imported variable is to encode it as a series of variables to indicate all its times of change and the values at each change. The maximal number of such changes is bounded by the depth of the execution tree. However, preliminary experiments demonstrated that this method will not even work for a very small depth using the Omega library. Thus, resolving the complex schedule and imported variable clauses was out of the question. Therefore, it was necessary to develop a new way to handle these ASTRAL features. By investigating a number of ASTRAL specifications, we found that in most cases a process instance is only interested in an imported variable's values during the last one or two changes. Therefore, the solution used is to limit the size of the windows of a process instance's view of its imported variables' histories. A user is asked to select the size to be either 1 or 2. Choosing one means that the process instance can only remember an imported variable's last change time, and the values before and after the change. 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. For example, under window size 1, $\mathtt{past}(\mathtt{s.train\_in\_R}, \mathtt{t}) = \mathtt{true}$ can be translated into $(t \geq s\_train\_in\_R\_lc \rightarrow 1 = s\_train\_in\_R\_v\_1) \wedge (t < s\_train\_in\_R\_lc \rightarrow 1 = s\_train\_in\_R\_v\_0)$ where variables $s\_train\_in\_R\_lc$, $s\_train\_in\_R\_v\_0$ and $s\_train\_in\_R\_v\_1$ indicate the last change time and the values before and after the change. Since the type of s.train_in_R is Boolean, one of the two variables $s\_train\_in\_R\_v\_0$ and $s\_train\_in\_R\_v\_1$ is redundant. Therefore, the above translation can be further simplified by substituting $s\_train\_in\_R\_v\_1$ with $1 - s\_train\_in\_R\_v\_0$. Of course, these newly introduced variables can be changed during a transition's execution. Therefore, it is necessary to add some further constraints to a transition relation that indicate how these variables can be changed when a transition is completed. One of the constraints is $s\_train\_in\_R\_lc > pNow \rightarrow (s\_train\_in\_R\_v\_0 = ps\_train\_in\_R\_v\_1 \wedge s\_train\_in\_R\_v\_1 \neq s\_train\_in\_R\_v\_0)$, which says that if the value of s.train_in_R changed during the transition, the new value of $s\_train\_in\_R\_v\_0$ is the old value of $s\_train\_in\_R\_v\_1$ and the new value of $s\_train\_in\_R\_v\_1$ is different from the new value of $s\_train\_in\_R\_v\_0$.

In some ASTRAL specifications, the entire history is referenced, and the limited window method will produce false negatives. Fortunately, an automated tool has been implemented to check whether each violation found is a false negative or not. Therefore, the model checker is sound in that it will not pop up a fake error if the specification is ok. It should also be pointed out that limiting the window size does not necessarily mean that an imported variable is only allowed to change one or two times during the whole model checking process. In fact, the maximal number of changes is still bounded by the depth of the execution tree. The real limit is that a process instance can only remember a very limited number of changes.

A local variable X is *explicitly resolvable*, if after the execution of a transition in which X is contained, the value of X has an explicit symbolic value. Therefore, when a local variable's history is referenced, such as the past value of a local variable X at some past time t, $\mathtt{past}(\mathtt{X}, \mathtt{t})$, it is desirable for this X to be an explicitly resolvable variable. Otherwise, an expensive translation is applied. For example, in the following transition

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

the local variable `position` has value `raised` after transition `up`'s execution. An automatic tool has been implemented to check whether a local variable is explicitly resolvable or not. Once `X` is explicitly resolvable, on any node in the execution tree, the entire history (sometimes excluding the initial value) of `X` is explicitly resolvable by looking backwards from the current node up to the root. Therefore, `past(X,t)` can be translated into a formula containing only one variable t (if the initial value of X is not explicitly resolvable an extra variable is added to indicate the initial value.) On the other hand, if `X` is not explicitly resolvable and `past(X,t)` is referenced, it is necessary to use an expensive method to encode the whole history of the variable `X`; i.e., new variables are added to record every time `X` changes and the values before and after the change. Experience shows that in this case it almost always means that the model checker can not check the specification for a non-trivial depth. This is due to the large number of introduced variables involved in a Presburger formula. Again, by looking at a number of existing ASTRAL specifications it is rare that a local variable is not explicitly resolvable and that its history is referenced.

3.6. Assumptions and Properties

If the goal is to check the local invariant, then ASTRAL proof theory dictates that the only assumptions that can be used are the local and global axiom clauses and the local initial clause. Therefore, these clauses comprise the **Assump** and the property **Prop** is the invariant clause. If the goal, on the other hand, is to check the local schedule, then the local and global axiom clauses, the local initial clause, the local imported variable clause and the local environment clause can all be used as assumptions. Therefore, these clauses comprise the **Assump**, and the property **Prop** is the schedule clause. It is important to note, however, that the imported variable clause and the environment clause must hold during a transition's execution (not just at the times when it fires and ends). Therefore, it is necessary to add the quantified ($\forall$`now` between the start time and the end time of the transition) form of the two clauses into **Assump** and to add that the start and the end time are dynamically resolvable at any node. The current implementation of the model checker only checks the properties in the start and end times of a transition in the label transition system $\mathcal{T}$, including the *idle* transition as introduced before. One can not simply add a quantified form of the properties as was done with the imported variable clause and the environment clause. The reason is that for a transition with duration more than 1 the process's environment could be changed during the duration and the quantified properties can not always reflect these changes.

4. Experiments and Results

The prototype implementation of the ASTRAL symbolic model checker is integrated into the ASTRAL SDE. Within the implementation, a cache is used to store the intermediate results of Omega relations and sets as well as their hashes. The model checker first visits the cache before an Omega calculation takes place. Doing this speeds up the model checking an average of four times, especially when a bug is found and the user makes a minor change to the specification and reruns the model checker. The cache miss rate on the benchmark specification was less than 40% on average. Since the use of the symbolic model checker in the ASTRAL SDE is only for debugging purposes, its effectiveness for detecting an error in a specification is the major concern. The symbolic model checker was run on earlier flawed versions of the benchmark specification. The results shown in Table 1 are encouraging: it successfully detects an error for each of the versions. As a comparison, the table also lists the times and states used to find the first error for the explicit state exploration model checker in the SDE. The symbolic model checker was also run on the benchmark specification with different set-ups, as shown in Table 2. In the "const set" column of Table 2, "none" means that no user-assigned constant values were used other than `min_speed` and `max_speed`, which must be set. Throughout the tests, `min_speed` and `max_speed` are set to be 15 and 20, respectively. In the same column, "duration" means that the constants representing duration of all the transitions in the process are set to 1. This is used to compare the performance of the model checker when a number of constant values are explicitly assigned by the user. The time is measured in seconds. All runs were performed on a Sun Ultra 1 with 64M main memory and 192M swap memory. The benchmark, which is also the final version of the railroad crossing ASTRAL specification survives the symbolic model checker. In fact, it has been proved to be correct using the theorem prover in the ASTRAL SDE [15]. It should be noted that the following experiments are independent. That is, before each run of the symbolic model checker the cache was cleared. Therefore, the performances of different cases are comparable. The following conclusions are drawn from the tables.

All flawed versions are falsified in relatively short time. For example, the error in the fourth flawed version in Table 1 is in fact caused by the second conjunction of the entry assertion of transition exit_I, $\texttt{train\_in\_R}$ $\texttt{and now} - \texttt{Call(enter\_R)} >= \texttt{RIImin} - \texttt{exit\_dur}$. This entry assertion wrongly says that if the sensor currently reports a train and the time since the transition enter_R was called is long enough for the slowest train to pass the sensor's region, this transition exit_I should be fired to report that the train leaves the region. However, the time when the entry assertion is satisfied is a bit earlier than expected, since it could be the case that enter_R was called but fired later. This will produce a scenario where exit_I completes so early that even the fastest train does not have enough time to pass the region. This error is corrected by changing Call(enter_R) to Start(enter_R). In fact, the resulting specification is the final version. Though the explicit state model checker is significantly faster than the symbolic model checker, 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. Picking an appropriate set of constant values to cause an error to happen is not trivial.

version	explicit-state model checker	symbolic model checker
version 1	119 states 0.01 secs	22 nodes 49 secs
version 2	602 states 0.05 secs	5 nodes 4 secs
version 3	4 states 0.01 secs	6 nodes 52 secs
version 4	1,029 states 0.14 secs	497 nodes 782 secs

Table 1. Experimental results on the earlier (flawed) versions

cases	process	property	window	n_track	const set	depth	nodes	time
T1	Sensor	invariant	N/A	N/A	duration	10	136	198↓
T2	Sensor	invariant	N/A	N/A	none	10	904	1,030↓
T3	Sensor	schedule	N/A	N/A	duration	10	136	266↓
T4	Sensor	schedule	N/A	N/A	none	10	613	1,026↓
T5	Gate	schedule	1	1	duration	10	186	286↓
T6	Gate	schedule	1	1	none	10	971	3,417↓
T7	Gate	schedule	1	2	duration	10	186	2,371↓
T8	Gate	schedule	1	2	none	10	774	73,996↑
T9	Gate	schedule	2	1	duration	10	186	323↓
T10	Gate	schedule	2	1	none	10	471	3,545↓
T11	Gate	schedule	2	2	duration	10	186	6,737↓
T12	Gate	schedule	2	2	none	10	377	58,644↑

Table 2. Experiments on the final version

Explicitly setting a number of constants can reduce the number of symbolic constant parameters when using the Omega library, since the efficiency of the Omega library is very sensitive to the number in some cases. For example, in Table 2, with all the constants representing the durations of the transitions set to 1, T5 is almost two times faster than T6 for each node searched. Sometimes, setting several constants make it possible for the symbolic model checker to finish the job. For example, T8 aborts (indicated by ↑ in the table) after checking 774 nodes. But T7, with the duration constants set up, is able to be completed (indicated by ↓ in the table). Similar results are shown for the last two cases T11 and T12.

The window size and the number of process instances in the system also directly affect the efficiency, since the number of variables and the size of the Presburger formulas grow dramatically when they become bigger.

The use of the model checkers in the ASTRAL SDE is only for debugging purposes. The recommended approach suggests that a user first use the explicit state model checker to check a specification by setting up all the global and local constants. That model checker is very fast and cheap, and performs global model checking. After several rounds of trials and revisions, if no further errors are found, it is time to start the symbolic model checker. The symbolic model checker performs process level model checking for each process type declared in the system. A user may start by setting up a number of constants and a window size in order to complete the procedure in a shorter time. Both model checkers are fully automated without manual abstractions. When an error is detected, a specification level trace that leads to the error is shown. When a specification survives the symbolic model checker, it gives a user greater confidence in the specification. The user can then use the ASTRAL theorem prover to formally prove the specification.

5. Conclusions

In this paper, the methods and techniques used in the prototype implementation of the ASTRAL symbolic model checker, which is a component of the ASTRAL SDE, were presented. The model checking procedure is carried out on the execution tree by using the Omega library to represent a subset of states. The tree is further trimmed by the execution graph of a process. The model checker combines both explicit state exploration and symbolic state calculation in order to reduce the number of variables by dynamically resolving their values. The model checker is modularized, based upon the ASTRAL proof theory, in the sense that each time it checks only one process instance of each process type that is globally declared.

A process interacts with other processes through its interface sections, which contain among others, imported variables and transitions. A limited window size technique was proposed to encode the history of an imported variable when the history of the variable is referenced. Doing this will sometimes incur false negatives. However, by using a tool to check whether a violation is a false negative, the model checker only reports real errors. In fact, the window size 2 as chosen in T9 through T12 in Table 2 is complete for the specification tested in this paper. The reason is that the process gate is only interested in the last two changes to the imported variable s.train_in_R. Therefore, a user gets full confidence in the specification with respect to the execution tree searched by the model checker.

The model checker was run on earlier flawed versions of the ASTRAL railroad crossing specification as well as on the latest correct version. The results show that the model checker is effective in detecting bugs in an ASTRAL specification, which is extremely important since its intended use is as a specification debugger. It was also shown that the model checker is not able to complete the procedure on two cases in Table 2. This is due to the extremely large size of the formulas as well as the number of variables.

The authors would like to thank T. Bultan and P. Kolano for many insightful discussions. The specifications, including the current version and the earlier erroneous versions, were written by P. Kolano in the Reliable Software Group at UC Santa Barbara.

References

[1] R. Alur and D. Dill, "Automata for modeling real-time systems," *Theoretical Computer Science*, Vol. 126, No. 2, 1994, pp. 183-236.

[2] R. Alur and T. A. Henzinger, "Parametric real-time reasoning," *STOC'93*, pp. 592-601.

[3] 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.

[4] 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.

[5] T. Bultan, R. Gerber, and C. League, "Verifying Systems with Integer Constraints and Boolean Predicates: A Composite Approach," *ISSTA '98*, pp. 113-123.

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

[7] 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.*

[8] 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*, pp. 415-418.

[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," *ICSE'99*, pp. 132-141.

[13] C. Heitmeyer and N. Lynch, "The generalized railroad crossing: a case study in formal verification of real-time systems," *RTSS'94*, pp. 120-131.

[14] F. Jahanian and A. K. Mok, "Modechart: A Specification Language for Real-Time Systems,"*IEEE Transactions on Software Engineering*, Vol. 20, No. 12, 1994, pp. 933-947.

[15] 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, 1999.

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

[17] K. G. Larsen, P. Petterssen, and W. Yi, "Compositional and symbolic model-checking of real-time systems," *RTSS'95*, pp. 76-87.

[18] 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.

[19] 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.