Real-Time Symbolic Model Checking for Discrete Time Models

Sérgio V. Campos Edmund M. Clarke

May 2, 1994
CMU-CS-94-146

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in T. Rus, C. Rattray, editors, *AMAST Series in Computing: Theories and Experiences for Real-Time System Development.* World Scientific Publishing Company.

This research was sponsored in part by the National Science Foundation under grant no. CCR-8722633, by the Semiconductor Research Corporation under contract 92-DJ-294, and by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of NSF, SRC, or the U.S. government.

Keywords: Symbolic Model Checking, Real-time Systems, Priority Inversion

Abstract

The BDD-based symbolic model checking algorithm given in [4, 10] is extended to handle real-time properties using the *bounded until* operator [9]. We believe that this algorithm, which is based on discrete time, is able to handle many real-time properties that arise in practical problems. One example of such a property is *priority inversion*. This is a serious problem that can make real-time systems unpredictable in subtle ways. Our work discusses this problem and presents one possible solution. The solution is formalized and verified using the modified algorithm. We also propose another extension to the model checking algorithm. *Timed transition graphs* are transition graphs in which events may take non-unit time to occur. The time it takes for a transition in a TTG to happen is determined by a time interval. This allows the construction of smaller and more realistic models. A symbolic model checking algorithm is given for formulas using the bounded until operator in TTG models.

1 Introduction

Temporal logic model checking is a technique for determining the correctness of finite-state systems. A large number of problems in computer science can be modeled using finite-state representations. Real-time systems can often be represented in such a way. Because they are used in many critical applications, being able to depend on them is vital. Model checking [5, 6] can assist in demonstrating the correctness of such systems. The use of this technique can help increase the efficiency of their validation and help generate systems with higher reliability. This work explains how model checking can be applied to the verification of real-time systems.

In model checking, specifications are expressed as formulas of a propositional temporal logic. The system to be verified is modeled as a state-transition graph, and the graph is searched to determine if it satisfies the property. A symbolic model checking algorithm is one in which the transition relation is represented implicitly by boolean formulas, and states are not explicitly enumerated. The SMV symbolic model checking algorithm [4, 10] is the basis of our approach. It is extended to handle real-time properties. The original model checking algorithm represents properties as formulas in the temporal logic CTL (Computation Tree Logic). This logic allows us to state properties such as "event p will happen sometime in the future", but not "event p will happen in at most x units of time". In real-time systems properties of the latter type appear frequently, because we must bound the execution time in order to make the system predictable. We augment CTL so that it is possible to express real-time properties using the *bounded until operator* [9], and show how to check formulas involving operators of this type using BDD-based *symbolic model checking techniques.*

Another extension to the algorithm comes from the fact that all transitions in a SMV model take exactly one step to occur. However, in realistic models this is not always true. Various transitions frequently have different lengths in practice. It is also possible that one transition can take different amounts of time to occur in different executions. Modeling this behavior in SMV can be achieved by expanding a non-unit transition into a sequence of transitions through several intermediate states. The states introduced by this technique may significantly increase the size of the model. We propose an extension called *Timed Transition Graphs* (TTG) to handle this situation. A Timed transition graph is a transition graph that has time intervals associated with transitions. The time intervals specify a lower and an upper bound on the time it takes for a transition to occur. A transition can take a nondeterministic number of steps to occur, within the bounds specified by the TTG. Longer transitions that are also non-deterministic (within specified bounds) allow the modeling of realistic systems without the burden of adding extra states to the model. A symbolic model checking algorithm is presented for bounded CTL formulas using TTGs as models.

As an example of how these techniques can be used, we model the *priority inversion* [8, 11] problem using the extended verifier. Most real-time systems rely on priorities to maintain predictability. The fact that higher priority tasks must be executed before lower priority tasks is essential for the correctness of such systems. However, low priority processes can block high priority processes indefinitely, because of indirect priority constraints. This situation is called *priority inversion*. This behavior makes the system unpredictable. It is described in this paper. Several solutions exist to this problem, and one of those, *priority inheritance*, is

presented and formally verified.

Temporal logic model checking is described in section 2. Section 3 discusses binary decision diagrams, which form the basis for the symbolic algorithms described in this work. The logic used in the model checker is presented in section 4, and in section 5 the symbolic model checking algorithm is explained. The extension that allows real-time properties to be expressed is described in section 6. In section 7 timed transition graphs are presented, and a symbolic model checking algorithm for TTG models is given. An example of how these techniques work, the priority inversion problem, is presented in section 8. The paper ends in section 9 with a discussion of the results.

2 Temporal Logic Model Checking

Extensive simulation is currently the most widely used verification technique. However, simulation does not exhaust all possible behaviors of a computing system. Exhaustive simulation is too expensive, and non-exhaustive simulation can miss important events, specially if the number of states in the system being verified is large. Other approaches for verification include theorem provers, term rewriting systems and proof checkers. These techniques, however, are usually very time consuming, and require user intervention to a large degree. Such characteristics limit the size of the systems they can verify in practice.

Temporal logic model checking [5, 6] is an alternative approach that has achieved significant results recently. Efficient algorithms are able to verify properties of extremely large systems. In this technique, specifications are written as formulas in a propositional temporal logic and computer systems are represented by state-transition graphs. Verification is accomplished by an efficient breadth first search procedure that views the transition system as a model for the logic, and determines if the specifications are satisfied by that model.

There are several advantages to this approach. An important one is that the procedure is completely automatic. The model checker accepts a model description, specifications written as temporal logic formulas and determines if the formulas are true or not for that model. Another advantage is that, if the formula is not true, the model checker will provide a counterexample. The counterexample is an execution trace that shows why the formula is not true. This is an extremely useful feature because it can help locate the source of the error and speed up the debugging process. Another advantage is the ability to verify partially specified systems. Useful information about the correctness of the system can be gathered before all the details have been determined. This allows the verification of a system to proceed concurrently with its design. Consequently verification can provide valuable hints that will help designers eliminate errors earlier and define better systems.

Properties to be verified are described as formulas in a propositional temporal logic. The system for which the properties should hold is given as a state transition graph. It defines a model for the temporal logic since the semantics of the logic are given in terms of state transition graphs. The model checker traverses this graph and verifies if the model satisfies the formula. Checking that a single model satisfies a formula is much simpler than proving that a formula is valid for all possible models. Because of this fact model checkers can be more efficiently implemented than theorem provers. Clarke and Emerson [5] developed the first algorithm. This algorithm used adjacency lists to represent the transition graph and had

a complexity that was polynomial in the size of the model and in the length of the formula. This and other equivalent systems were able to handle graphs with up to 10^5 states.

Around 1987, however, the concept of *symbolic model checking* was introduced [4, 10]. In the new approach the transition relation is represented implicitly by boolean formulas, and implemented by *ordered binary decision diagrams* [1]. This usually results in a much smaller representation for the transition relation, allowing the size of the models being verified to increase up to more than 10^{20} states. The symbolic model checking approach will be explained in more detail later.

3 Binary Decision Diagrams

Ordered binary decision diagrams (BDD) are an efficient way to represent boolean formulas. BDDs often provide a much more concise representation than traditional representations like conjunctive normal form or disjunctive normal form. They can also be manipulated very efficiently [1]. Another advantage offered by BDDs is that they provide a *canonical representation* for boolean formulas. This property means that two boolean formulas are logically equivalent if and only if they have isomorphic representations. It greatly simplifies the execution of operations that are performed frequently like checking equivalence of two formulas or deciding if a given formula is satisfiable or not. Because of these characteristics, BDDs have found application in the implementation of many CAD tools.

Boolean formulas can be represented by binary decision trees. The nodes in the decision tree correspond to the variables of the formula. Descendants of a node are labelled with *true* or *false*. The value of the formula for a given assignment of values to the variables can be found by traversing the tree from root to leaf. At each node the descendant labelled with the value of that variable is chosen. Each leaf corresponds to a particular assignment to the variables, and contain the truth value of the formula for that assignment.

This representation is not particularly compact, because it may store the same information repeatedly in different places. BDDs are derived from binary decision trees but its structure is a directed acyclic graph instead of a tree. Redundant information in the structure is avoided by eliminating common subtrees. As in decision trees, nodes are visited in sequence, from root to leaf. However, BDDs impose a total ordering in which the variables occur in this sequence. For example, the BDD in figure 1 represents the formula $f = (a \wedge b) \vee (c \wedge d)$ using the ordering $a < b < c < d$ for the variables.

Given an assignment for the variables in f we can decide if this assignment satisfies the formula by traversing the BDD from root to leaf. At each node we follow the path that corresponds to the value assigned to the variable in the node. The leaf indicates if the formula is satisfied or not for that particular assignment. Notice that redundancy is eliminated in two ways. Common subtrees are not replicated, as can be seen from the paths when a is false and when b is false. Also, when all the leaves of a subtree lead to the same value, the subtree is eliminated, and a leaf of that value is inserted at its place. Notice in the figure that when a and b are both true a subtree containing the variables c and d is eliminated because all of its leaves would have the value 1.

For any boolean formula and with a fixed variable ordering there exists a unique BDD [1]. The size of the BDD is critically dependent on the variable ordering. It is exponential in

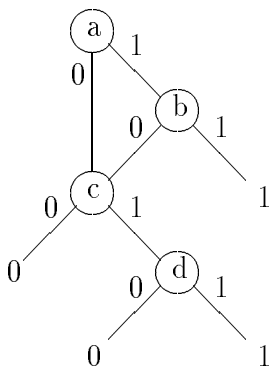

Figure 1: BDD for formula $(a \wedge b) \vee (c \wedge d)$

the number of variables in the worst case. Given a good variable ordering, however, the size is linear in most practical cases. Using a good variable ordering is very important. But finding the optimal order is in itself a NP-complete problem. Nevertheless, there are many heuristics that work quite well in practice.

Efficient algorithms exist to handle boolean formulas represented by BDDs. Given BDD representations for f and g, algorithms for computing $\neg f$ and $f \vee g$ are given in [1]. Algorithms for quantification over boolean variables and substitution of variable names are also required by the model checker. It is simple to compute the restriction of a formula f with a variable v set to 0 or 1. We will denote the restriction of f with v set to 0 by $f|_{v=0}$, and the restriction of f with v set to 1 by $f|_{v=1}$. The formula $\exists v[f]$ is defined as $f|_{v=0} \vee f|_{v=1}$, and $\forall v[f]$ is defined as $\neg\exists v[\neg f]$. Substitution of variable names can be accomplished using the quantification algorithm. $f<v \leftarrow w>$ denotes the substitution of variable w for variable v in formula f. It is computed as $f<v \leftarrow w> = \exists v[(v \Leftrightarrow w) \wedge f]$. These operations are performed very frequently in the model checker, and more efficient algorithms are used in the actual system. Describing these algorithms is out of the scope of this paper, but they can be found in [2].

4 Computation Tree Logic

Computation tree logic, CTL, is the logic used by SMV to express properties that will be verified. *Computation trees* are derived from state transition graphs. The graph structure is unwound into an infinite tree rooted at the initial state, as seen in figure 2. Paths in this tree represent all possible computations of the program being modelled. Formulas in CTL refer to the computation tree derived from the model. CTL is classified as a *branching time* logic, because it has operators that describe the branching structure of this tree.

Formulas in CTL are built from atomic propositions, where each proposition corresponds to a variable in the model, boolean conectives $\neg$ and $\wedge$, and *temporal operators*. Each operator consists of two parts: a path quantifier followed by a temporal operator. Path quantifiers indicate that the property should be true of *all* paths from a given state (**A**), or *some* path from a given state (**E**). The temporal quantifier describe how events should

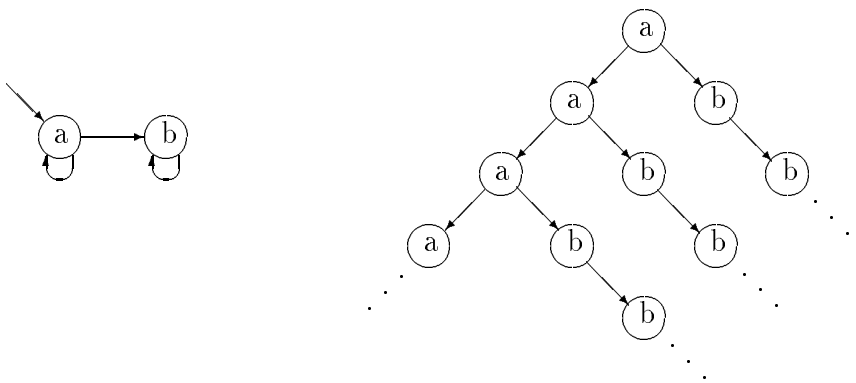

Figure 2: State transition graph and corresponding computation tree

be ordered with respect to time for a path specified by the path quantifier. They have the following informal meanings:

- **F** φ (φ holds sometime in the future) is true of a path if there exists a state in the path that satisfies φ.
- **G** φ (φ holds globally) is true for a path if φ is satisfied by all states in the path.
- **X** φ (φ holds in the next state) means that φ is true in the next state of the path.
- φ **U** ψ (φ holds until ψ holds) is satisfied by a path is ψ is true in some state in the path, and in all preceding states, φ holds.

Formally, the syntax for CTL can be defined by:

- Every atomic proposition p is a CTL formula.
- If f and g are CTL formulas, then so are $\neg f$, $f \wedge g$, **EX** f, **EG** f and **E**$[f$ **U** $g]$.

The semantics of CTL formulas are defined with respect to a labeled state-transition graph, which is a 5-tuple $\mathcal{M} = (P, S, L, N, S_0)$, where P is a set of atomic propositions, S is a finite set of states, L is a function labeling each state with a set of atomic propositions, $N \subseteq S \times S$ is a transition relation, and S_0 is the set of initial states. A path is an infinite sequence of states $s_0 s_1 s_2 ...$, such that $N(s_i, s_{i+1})$ is true for every i.

If f is true in a state s of structure $\mathcal{M}$, we write $\mathcal{M}, s \models f$. We write $\mathcal{M} \models f$ if $\mathcal{M}, s \models f$ for all states s in S_0. The satisfaction relation is defined inductively as follows (Given the model $\mathcal{M}$, we abbreviate $\mathcal{M}, s \models \varphi$ by $s \models \varphi$):

1. If φ is the atomic proposition $v \in P$, then $s \models \varphi$ if and only if $v \in L(s)$.
2. $s \models \neg\varphi$ iff it is not the case that $s \models \varphi$. $s \models \varphi \wedge \psi$ iff $s \models \varphi$ and $s \models \psi$.
3. $s \models$ **EX** φ iff there exists a path $\pi = s_0 s_1 s_2 ...$ starting at $s = s_0$, such that $s_1 \models \varphi$.

4. $s \models \mathbf{EG}\ \varphi$ iff there exists a path π starting at s such that for every state s' on π, $s' \models \varphi$.

5. $s \models \mathbf{E}[\varphi\ \mathbf{U}\ \psi]$ iff there exists a path $\pi = s_0 s_1 s_2 ...$ starting at $s = s_0$ and some $i \geq 0$ such that $s_i \models \psi$ and for all $j < i, s_j \models \varphi$.

The following abbreviatons are used in CTL formulas:

$\mathbf{AX}\ f \equiv \neg\mathbf{EX}\ \neg f.$

$\mathbf{EF}\ f \equiv \mathbf{E}[\text{true}\mathbf{U}f]$

$\mathbf{AF}\ f \equiv \neg\mathbf{EG}\ \neg f$

$\mathbf{AG}\ f \equiv \neg\mathbf{EF}\ \neg f$

$\mathbf{A}[f\mathbf{U}g] \equiv \neg\mathbf{E}[\neg g\mathbf{U}\neg f \wedge \neg g] \wedge \neg\mathbf{EG}\neg g.$

Some examples of CTL formulas are given below to illustrate the expressiveness of the logic.

- $\mathbf{AG}(req \rightarrow \mathbf{AF}\ ack)$: It is always the case that if the signal *req* is high, then eventually *ack* will also be high.
- $\mathbf{EF}(started \wedge \neg ready)$: It is possible to get to a state where *started* holds but *ready* does not hold.
- **AG EF** *restart*: From any state it is possible to get to the *restart* state.
- $\mathbf{AG}(send \rightarrow \mathbf{A}[send\ \mathbf{U}\ recv])$: It is always the case that if *send* occurs, then eventually *recv* is true, and until that time, *send* must remain true.

5 Symbolic Model Checking

Early model checking algorithms represented the transition graph through adjacency lists. All existing states were explicitly enumerated. Since the model checking problem has an exponential behavior in the worst case, this frequently caused state explosion problems. The size of systems that could be verified was severely limited. Symbolic model checking represents states and transitions using boolean formulas. This usually generates smaller representations, because it can automatically eliminate redundancy in the graph. Implementing these boolean formulas as BDDs leads to very efficient algorithms for model checking that are able to verify much larger systems than previous ones. This section will explain the symbolic model checking approach.

Representing the Model

A model of the system in our algorithm is a labeled state-transition graph $\mathcal{M}$, and assertions about the system are expressed as CTL formulas. The key to the efficiency of the algorithm is to use BDDs to represent the labeled state-transition graph and to verify if the formula is true or not. The following method will be used to represent the transition relation as a BDD. Assume that system behavior is determined by the boolean variables $V = \{v_0, ..., v_{n-1}\}$. Let $V' = \{v'_0, ..., v'_{n-1}\}$ be a second copy of these variables. We will use the variables in V to represent the value of the variables in the current state, and the variables in V' to represent the value in the next state. The relationship between values of variables in the current and the next states is written as a boolean formula using V and V'. This will generate the boolean formula N representing the transition relation. This formula will then be converted to a BDD.

$$N(v_0, ..., v_{n-1}, v'_0, ..., v'_{n-1})$$

We represent states by the values of the atomic propositions in those states. In order to guarantee that we can identify states uniquely, we must make the assumption that different states have different labeling of propositions. More formally, we assume that for any two states s_1 and s_2 in S, if $L(s_1) = L(s_2)$ then $s_1 = s_2$. This assumption does not, however, impose any restrictions on the model, since extra atomic propositions can be added in order to make $L(s_1) \neq L(s_2)$ for distinct states s_1 and s_2 [3].

Fixpoint characterization

Consider a labeled transition graph $\mathcal{M}$ with set of states S. We can denote a lattice of predicates over S by $Pred(S)$, where each predicate is identified with the set of states in S that make it true, and use set inclusion as ordering. A functional F that maps $Pred(S)$ to $Pred(S)$ is called a *predicate transformer*. Informally, $Pred(S)$ is a set of states, and F is a function from sets of states to set of states.

As described in [7], if a predicate transformer F is monotonic, it has a least fixpoint **lfp** $Z[F(Z)] = \cup_i F^i(false)$ and a greatest fixpoint **gfp** $Z[F(Z)] = \cap_i F^i(true)$. We can compute both fixpoints by iteration. Starting with $Z^0 = false$ (for **lfp**) or $Z^0 = true$ (for **gfp**), we have $Z^{i+1} = Z^i \cup F(Z^i)$ for **lfp** and $Z^{i+1} = Z^i \cap F(Z^i)$ for **gfp**. The fixpoint is found when $Z^i = Z^{i+1}$. If the number of elements in $Pred(S)$ is finite, termination is guaranteed, because there can be no infinite sequence of Z^is such that $Z^i \neq Z^{i+1}$.

We can identify each CTL formula f with the predicate $\{s \mid \mathcal{M}, s \models f\}$ in $Pred(S)$ (this is the set of states that satisfy f). Then, we can characterize each basic CTL temporal operator as fixpoints of an appropriate predicate transformer. The set of states that satisfy the until operator is given by the least fixpoint $\mathbf{E}[f\mathbf{U}g]$ of $Z = g \vee (f \wedge \mathbf{EX}\ Z)$. Informally $\mathbf{E}[f\mathbf{U}g]$ is true at state s, if either g is true in s, or f is true in s and there exists a successor state where $\mathbf{E}[f\mathbf{U}g]$ is true. The set of states that satisfy the **EG** operator is given by the greatest fixpoint $\mathbf{EG}\ f$ of $Z = f \wedge \mathbf{EX}\ Z$. Informally, this means that $\mathbf{EG}\ f$ holds in a state s if f holds in s and $\mathbf{EG}\ f$ holds in a successor state of s. Proofs that the characterizations above correspond to the expected semantics are given in [7].

The Model Checking Algorithm

Given a CTL formula φ and a model $\mathcal{M}$ represented as described above, we want to verify if φ is satisfied in the initial states of $\mathcal{M}$. The model checking algorithm is defined inductively over the structure of CTL formulas. It accepts the formula as an argument (and $\mathcal{M}$ as an implicit argument), recurses over the structure of φ and returns a BDD that has one boolean variable for every atomic proposition in V. The resulting BDD is true in a state if and only if φ is true in that state. The algorithm is:

- If φ is an atomic proposition p, return the BDD that is true if and only if p is true. This is simply the BDD for p.
- If φ is $\neg f$ or $f \wedge g$, use the standard BDD algorithms for computing boolean connectives.
- If φ is $\mathbf{EX}\ f$, then we must verify if f is true in a successor state of the current state. $\mathbf{EX}\ f$ is true in a state t if and only if there exists a state s such that f is true in state s, and there exists a transition from t to s:

  $$t \models \mathbf{EX}\ f \quad \text{iff} \quad \exists s[f\langle s\rangle \wedge N(t,s)]$$

  where $f\langle s\rangle$ means the value of formula f in state s. To compute this value we substitute the free variables in f by their values in state s using the substitution algorithm. In other words, $f\langle s\rangle$ is true if and only if $s \models f$. The relational product $\exists s[f\langle s\rangle \wedge N(t,s)]$ can be computed using the basic operations on BDDs, as described in [3]. However, this operation occurs frequently, and it is important to compute it in an efficient manner; efficient algorithms for this purpose are discussed in [2].
- If φ is $\mathbf{E}[f\mathbf{U}g]$, the computation of the set of states that satisfy φ can be characterized as a fixpoint computation, as shown previously. The BDD that represents the states where $\mathbf{E}[f\mathbf{U}g]$ is true can be computed by finding the least fixpoint $\mathbf{E}[f\mathbf{U}g]$ of:

  $$\mathbf{E}[f\mathbf{U}g] = g \vee (f \wedge \mathbf{EX}\ \mathbf{E}[f\mathbf{U}g])$$

- If φ is $\mathbf{EG}f$, the algorithm is defined in a similar way. It searches for the greatest fixpoint $\mathbf{EG}f$ instead, and uses the following formula:

  $$\mathbf{EG}f = f \wedge \mathbf{EX}\ \mathbf{EG}f$$

- All other CTL operators are written in terms of the ones presented.

6 Real-Time Logics

The logic CTL can be used to specify many properties of finite state systems. However, there is an important class of properties that cannot be adequately handled using this logic. This class consists of the properties that involve *quantitative* constraints, that is, the class

of properties which place bounds on response time. In CTL it is possible to express the property that some event will happen in the future, but not that some event will happen at most x time units in the future. In this section we will discuss one way of augmenting CTL to permit representation of such properties.

In order to represent bounded properties, we add time intervals to the existing temporal operators, as described in [9]. The basic temporal operator that we use in our real-time logic is the *bounded until* operator which has the form: $\mathbf{U}_{[a,b]}$, where $[a, b]$ defines the time interval in which our property must be true. We say that $f\mathbf{U}_{[a,b]}g$ is true of some path if g holds in some future state s on the path, f is true in all states between the beginning of the path and s, and the distance from this state to s is within the interval $[a, b]$. The bounded **EG** operator can be defined similarly. Other temporal operators are defined in terms of these.

More formally, we extend our CTL semantics to include the *bounded until* by adding the following clauses to the formal semantics given in section 4:

6. $s \models \mathbf{E}[\varphi\mathbf{U}_{[a,b]}\psi]$ iff there exists a path $\pi = s_0 s_1 s_2 ...$ starting at $s = s_0$ and some i such that $a \leq i \leq b$ and $s_i \models \psi$ and for all $j < i, s_j \models \varphi$.

7. $s \models \mathbf{EG}_{[a,b]}\varphi$ iff there exists a path $\pi = s_0 s_1 s_2 ...$ starting at $s = s_0$ and some i such that $a \leq i \leq b$, $s_j \models \varphi$ for all $j \leq i$.

As an example of the use of the *bounded until* consider the property "It is always true that p may be followed by q within 3 time units". this property can be expressed as $\mathbf{AG}(p \rightarrow \mathbf{EF}_{[0,3]}q)$. The bounded **F** operator is derived from the *bounded until* just as in the unbounded case, i.e. $\mathbf{EF}_{[a,b]}f \equiv \mathbf{E}[\text{true}\mathbf{U}_{[a,b]}f]$.

In order to implement this operator, we will use a fixpoint computation that is similar to the one implemented in CTL. It is easy to see that the formula $E[f\mathbf{U}_{[a,b]}g]$ can be expressed in the form:

if $a > 0$ and $b > 0$:	$E[f\mathbf{U}_{[a,b]}g] = f \wedge EXE[f\mathbf{U}_{[a-1,b-1]}g]$
if $b > 0$:	$E[f\mathbf{U}_{[0,b]}g] = g \vee (f \wedge EXE[f\mathbf{U}_{[0,b-1]}g])$
and	$E[f\mathbf{U}_{[0,0]}g] = g$

Other operators are defined similarly.

Consider the first of these cases. We compute the sets of states where f is true for a steps. During this computation, a fixpoint may be reached before a iterations have passed. When this happens, we can skip to the second case. By using this optimization, the number of required iterations may be reduced when the time interval is large, but a fixpoint is reached quickly. The same optimization can also applied in the second case. If a fixpoint is reached before $b - a$ iterations, with b and a being respectively the upper and lower bounds of the operator, we can immediately proceed to the third case.

7 Timed Transition Graphs

The extensions presented above allow the verification of a number of real-time systems. However, transition graphs have another important limitation for modeling time bounded computing systems. All transitions happen in one step. In actual systems events take

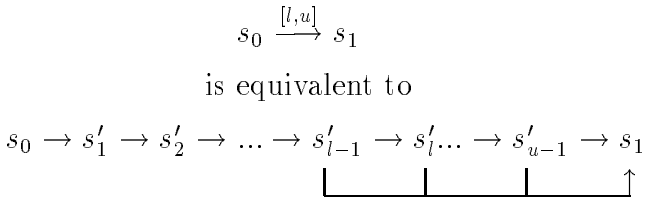

Figure 3: A non-unit transition in a TTG

different amounts of time to occur. Moreover, the time it takes for some event to take place may change in different executions. We call this behavior *bounded stuttering*. A transition can stutter if the time it takes to occur is not fixed, but is instead determined by a time interval.

A transition that takes more than one step and stutters can also be modeled in a transition graph. The longer path can be expanded into a series of one step transitions. Extra states and transitions have to be added to the transition graph. This makes the verification process more complex. The number of states added to the system is proportional to the size of the transitions being expanded. Extra transitions between states have to be added to introduce bounded stuttering. If there are many non-unit transitions, or if the individual transitions are long, this can cause state explosion problems.

We introduce the idea of *Timed Transition Graphs*, TTG, to help alleviate this problem. TTGs remove the unit transition limitation from transition graphs. With each transition in a TTG is associated a time range of the form $[a, b]$, where $a, b \in \mathrm{N}$. A transition labelled with $[a, b]$ will happen in x steps, where $a \leq x \leq b$. This extension allows transitions with length longer than one and also introduces bounded stuttering. A transition takes x steps, but x is chosen nondeterministically, within the bounds defined by a and b.

Formally, a timed transition graph is a 5-tuple $\mathcal{M} = (P, S, L, R, S_0)$, where P is a set of propositional variables, S is a set of states, L is a function labeling each state with a set of propositonal variables that are true in that state, S_0 is a set of initial states and $R \subseteq S \times \mathrm{N} \times \mathrm{N} \times S$ is a transition relation. Informally, $R(s_0, l, u, s_1)$ indicates that the transition between state s_0 and s_1 can take from l to u steps to occur.

The SMV model checking algorithm can be extended to verify properties of TTG models. Procedures for handling unbounded properties and boolean conectives can be used without modification. To verify bounded properties we must first extend the representation of the transition relation to include the bounds for each transition. The algorithm uses a relation $\mathcal{R}$ derived from R to represent the transition relation. $\mathcal{R}(s_0, t, s_1)$ is true iff there exists s_0, l, u, s_1 and t such that $R(s_0, l, u, s_1)$ is a transition of the model, and $l \leq t \leq u$. The algorithm encodes variables and states as vectors of boolean variables. The time variable t is also encoded as a vector of boolean variables. In the discussion below, though, we do not distinguish between the value of a state or t and its encoding.

The model checking algorithm is an extension of the original one. It is computed by an iterative procedure. The algorithm maintains a current set of states that satisfy φ. Each iteration finds states that have a transition to an element in the set computed by the previous iteration and updates the current set. The fixpoint of this iteration process is the

set of states that satisfy φ. For example, to find the set of states that satisfy $\mathbf{E}[f\mathbf{U}_{[a,b]}g]$ we use the method outlined below.

$$s \models \mathbf{E}\,[\,f\,\mathbf{U}_{[a,b]}\,g\,] \text{ iff } \exists\, t.\; s \models [\mathbf{E}\,[\,f\,\mathbf{U}_t\,g\,]\;\wedge a \leq t \leq b\,]$$

We compute the *bounded until* for an interval as an extension of the bounded until for a single time t. Notice that $f\langle s\rangle$ iff $s \models f$.

$$\begin{array}{l}\mathbf{E}\,[\,f\,\mathbf{U}_t\,g\,] = \\ \quad (g\langle s\rangle \wedge t = 0)\vee \\ \quad \exists\, t_1, t_2, s'\,[f\langle s\rangle \wedge (\mathbf{E}\,[\,f\,\mathbf{U}_{t_1}\,g\,])\langle s'\rangle \wedge \mathcal{R}\,(s, t_2, s') \wedge t = t_1 + t_2]\end{array}$$

The formula $g\langle s\rangle \wedge t = 0$ is true if state s satisfies g and the time bound allows the path to have length 0. The formula, $(\mathbf{E}\,[\,f\,\mathbf{U}_{t_1}\,g\,])\langle s'\rangle \wedge \mathcal{R}\,(s, t_2, s')$, is true if s has a transition to a state s' and s' satisfies $\mathbf{E}[f\mathbf{U}_{t_1}g]$. To verify if s satisfies the bounded property we must see if the length of the path from s' added to the length of the path from s to s' is within the bounds. $t = (t_1 + t_2)$ verifies if this requirement is satisfied by s' and some t_1, t_2 that satisfy the transitions on the graph. Equations that compute the set of states that satisfy other operators are similarly defined, and will not be presented here for brevity.

The TTG approach does not suffer from the same problems as the path expansion technique, but it does add to the complexity of the fixpoint calculation. The existential quantification algorithm must be applied to the variables that represent the time of a transition. This is an expensive operation, and can also cause state explosion problems. However, the TTG algorithm is more efficient than unrolling states. The number of boolean variables added to the model to represent the time range is proportional to $\log u$, where u is the largest upper bound of all transitions. The existential quantification is applied to these variables. Also, this approach is independent of the number of long transitions and does not introduce another overhead for stuttering transitions.

8 Examples

As an example of how these techniques can be applied to real-time systems, we'll model the *priority inversion* problem, and a solution to this problem, *priority inheritance*. Our model shows how priority inversion affects the predictability of real-time systems, and how inheritance solves the problem. A description of the problem and the solution is first given.

Priorities are essential in real-time systems. The correct ordering of task execution is a fundamental problem that must be solved if the system is to be predictable. Many scheduling policies have been developed to define what constitutes a correct ordering and to enforce this ordering during the execution of the system. If a scheduling policy requires that higher priority tasks execute before lower priority tasks, it is possible for a low priority process to be executing while a higher priority one is blocked. This situation is called *priority inversion*. Unbounded priority inversions occur when high priority processes are blocked indefinitely

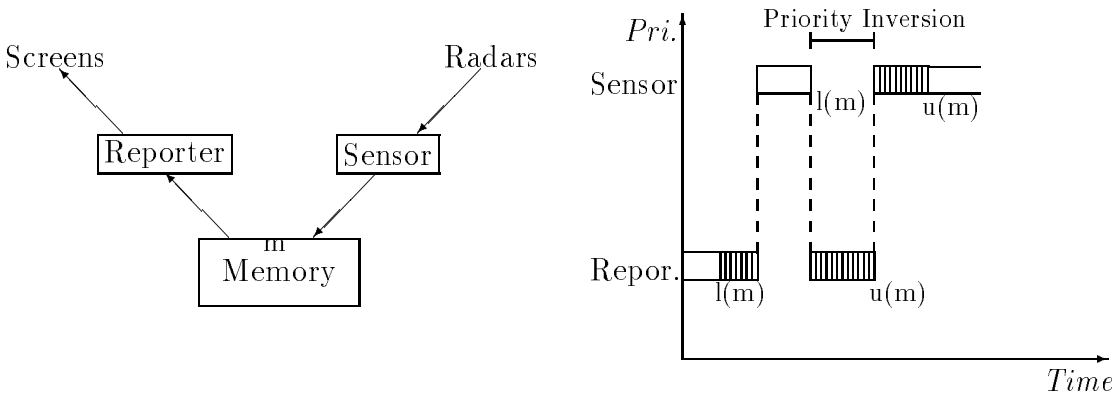

Figure 4: Bounded priority inversion

by low priority processes. When this happens, the system may become unpredictable. The correct ordering of task execution will be compromised, and the system may fail to satisfy its specification.

In order to present the problem in a more concrete framework, we will introduce a hypothetical air-traffic control system. We will concentrate our analysis in two of the processes in the system. The first, called *sensor*, reads airplane position data from radars, sets alarms on catastrophic conditions (conditions that cannot wait for a detailed analysis), and puts the data into shared memory. The other process is the *reporter*, that reads the data collected by the *sensor*, and updates the traffic controller screens. The *sensor* is a high priority process, because it processes urgent events, and must not be blocked by other processes. The *reporter* on the other hand, is a low priority process. Since it doesn't process urgent events, it may be delayed by other more important tasks.

The *sensor* and the *reporter* processes share data. To access this data appropriately, synchronization is necessary. In our system, the synchronization is implemented by a mutex variable which guarantees mutual exclusion among the processes accessing the data. The mutex variable is locked every time shared data is accessed. However, this may result in priority inversion. Suppose *reporter* is inside the critical section, and *sensor* tries to insert new data into the buffer area. The *sensor* can't access the data and blocks, waiting for *reporter* to unlock the mutex. Now a high priority process is waiting for a low priority process, and priority inversion occurs. Figure 4 shows this situation.

This priority inversion scenario is bounded. The *reporter* will delay the *sensor* only while it is inside the critical section. After the *reporter* releases the lock, the *sensor* will start executing, and the priority inversion will disappear. We can calculate the maximum duration of the priority inversion as the time to execute the largest critical section, and incorporate it in our calculations for the execution times. The system will still be predictable, although there may be a little loss in accuracy in execution time predictions. Consequently, if the system is well designed, and the critical sections are small, bounded priority inversions can

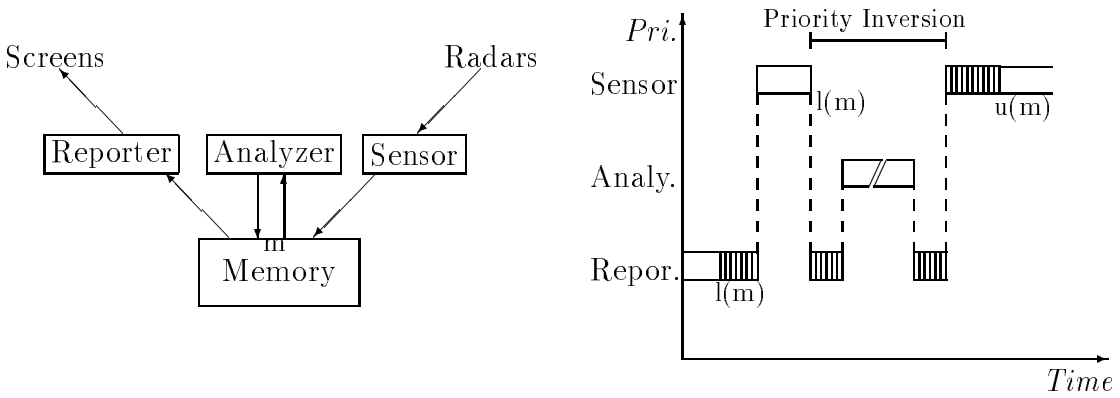

Figure 5: Unbounded priority inversion

be tolerated, without loosing predictability.

In certain cases, it is possible to have unbounded priority inversions that cannot be solved by this simple method. Suppose a third process, called the *analyzer* is added to the system. This process reads data generated by other components of the air-traffic controller and processes it. The *analyzer* is less important than the *sensor* and has a lower priority. But it is more important than the *reporter*, since urgent conditions may arise as the result of the analysis and handling them is more important than updating the screen. Consider now the same scenario as above, with the *reporter* inside the critical section, and the *sensor* waiting on the mutex. At this point, the *analyzer* starts executing. It will block the *reporter*, since it has higher priority. However, the *sensor* is waiting for the *reporter* (and therefore also for the *analyzer*). Since the *analyzer* doesn't know the relation between the *reporter* and the *sensor*, it may execute for an unbounded amount of time and delay the *sensor* indefinitely. If a catastrophic event occurs, it will go unnoticed, because the *sensor* is blocked. As a result, the behavior of the system becomes unpredictable. Figure 5 shows this situation.

Priority inheritance protocols are one way of preventing unbounded priority inversions. A typical protocol might work in the following manner. As soon as a high priority process is blocked by a low priority one, the low priority process is temporarily given the priority of the blocked process. While inside the critical section the *sensor* is trying to access, the *reporter* will execute at high priority. When the *reporter* exits the critical section, it will be restored to its original priority. In this way, the *analyzer* will not be able to interrupt the *reporter*, when the *sensor* is waiting. We will show that this protocol avoids the unbounded priority inversion problem (except possibly for deadlocks in accessing synchronization variables). This allows the designer of the system to predict the maximum priority inversion time, as in the bounded case.

Priority inversion occurred in this example because the *analyzer* preempted the *reporter*. Another cause of priority inversion is queueing. Communication protocols may experience priority inversion for this reason. For example, packets to be sent to the network may have

priorities. Low priority packets may be enqueued ahead of high priority ones in some protocol queue. In a prioritized network a high priority packet may have to wait for a low priority one to be sent. If medium priority packets start arriving in another processor's queue, they may monopolize the network, preventing high priority packets from being sent. Again, we have unbounded priority inversion. This type of priority inversion could also happen in our system, if the different components were distributed over a network. For example, *sensor* packets could be queued after some low priority packets in a queue, while *analyzer* packets were being trasmitted.

The inheritance mechanism that we have described to avoid unbounded inversions is called basic priority inheritance protocol. There are other priority inheritance protocols. Some protocols are designed to avoid deadlocks caused when critical sections are accessed in the wrong order. Other protocols are designed to handle *chained bounded priority inversions.* A chained inversion occurs when a high priority process wants to lock n mutexes that are already locked by low priority processes. In this case, the high priority process has to wait for all low priority processes to finish their critical sections. While this wait is bounded, it may be too expensive to wait for the duration of all critical sections. One possible solution to this problem is to assign priorities to critical sections, based on the priorities of the processes that may access it. A process is allowed to access a critical section only if its priority is higher than the priority of all critical sections currently being accessed. A more complete study of these various algorithms and their characteristics can be found in [8, 11].

Our implementation of the basic priority inheritance protocol is discussed in the full version of the paper. The three processes are implemented as described. We want to determine if the *sensor* can starve:

$$AG(sensor.state = trying \rightarrow AF sensor.state = critical)$$

This property is false without the priority inheritance mechanism. The property becomes true when priority inheritance is activated. Moreover, we can verify that there is an upper limit on the time the *sensor* enters the critical section with the following formula:

$$AG(sensor.state = trying \rightarrow AF_{[0,32]} sensor.state = critical)$$

9 Conclusions

In this work we have shown how temporal logic model checking can be used to verify properties of real-time systems. We extended an existing symbolic model checker to handle properties that are bounded in time. The *bounded until* operator was implemented to allow the expression of such properties.

Timed transition graphs were proposed to extend even further the expressiveness of the tool. In a TTG, transitions have time bounds, and a transition can take a nondeterministic time to occur within these bounds. This allows the representation of more realistic models. A symbolic model checking algorithm was given to verify properties in TTG models.

As an example of the usefulness of bounded operators, we discussed the priority inversion problem in real time systems. We formalized a solution for a particular instance of this

problem and verified that it was correct using temporal logic model checking techniques. This example demonstrates that non-trivial properties of real-time systems can be proven using symbolic model checking techniques.

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[2] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *VLSI 91*, Edinburgh, Scotland, 1990.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27^{th} ACM/IEEE Design Automation Conference*, 1990.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, 1990.

[5] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*. Springer-Verlag, 1981. volume 131 of *Lecture Notes in Computer Science*.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[7] E. M. Clarke, O. Grumberg, and D. E. Long. Verification tools for finite-state concurrent systems. In *REX '93 School/Workshop: A Decade of Concurrency*, Noordwijkerhout, The Netherlands, June 1993. to appear in Springer Lecture Notes in Computer Science.

[8] S. Davari and L. Sha. Sources of unbounded priority inversion in real-time systems and a comparative study of possible solutions. *ACM Operating Systems Review*, pages 110–120, April 1992.

[9] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Lecture Notes in Computer Science*. Springer–Verlag, 1990.

[10] K. L. McMillan. *Symbolic model checking — an approach to the state explosion problem.* PhD thesis, SCS, Carnegie Mellon University, 1992.

[11] R. Rajkumar. *Task synchronization in real-time systems.* PhD thesis, ECE, Carnegie Mellon University, 1989.