1

The Priority Inversion Problem and Real-Time Symbolic Model Checking

Sérgio V. Campos

April 23, 1993
CMU-CS-93-125

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Priority inversion is a serious problem that can make real-time systems unpredictable in subtle ways. This makes it more difficult to implement and debug such systems. Our work discusses this problem and presents one possible solution. The solution is formalized and verified using temporal logic model checking techniques. In order to perform the verification. the BDD-based symbolic model checking algorithm given in [4, 11] was 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 prcblems.

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

The Priority Inversion Problem and Real-Time Symbolic Model Checking

CMU-CS-93-125

Sergio V. Campos

April 1993

Priority inversion is a serious problem that can make real-time systems unpredictable in subtle ways. This makes it more difficult to implement and debug such systems. Our work discusses this problem and presents one possible solution. The solution is formalized and verified using temporal logic model checking techniques. In order to perform the verification, the BDD-based symbolic model checking algorithm given in McMillan's *Symbolic model checking — an approach to the state explosion problem* was extended to handle real-time properties using the *bounded until* operator. We believe that this algorithm, which is based on discrete time, is able to handle many real-time properties that arise in practical problems.

Keywords: SYMBOLIC MODEL CHECKING, REAL-TIME SYSTEMS, PRIORITY INVERSION

(22 pages)

1 Introduction

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*. It is described in detail this paper, and one solution, *priority inheritance*, is presented and formally verified using temporal logic *model checking* techniques.

To make this possible we extend the SMV model checking algorithm [4, 11] to handle real-time properties. The original model checking algorithm represents properties as formulas in the temporal logic CTL. 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.* We argue that this is a useful extension to the model checker and that in spite of the limitation to discrete time, our model checker is powerful enough to handle most properties that occur in practice.

This paper is organized as follows: Section 2 briefly describes the logic used for expressing program properties. In Section 3 we explain the symbolic model checking algorithm used as the basis for this work. The extension to CTL that allows real-time properties to be expressed is described in Section 4. Some simple examples are presented in Section 5 to demonstrate how the new technique works. Section 6 describes the priority inversion problem, and how model checking can be used to verify a solution to this problem. The paper ends in Section 7 with a discussion of the results and some directions for future research.

2 Computation Tree Logic

Our model checking algorithm is based on a propositional, branching-time temporal logic called CTL. This logic is discussed in detail in [6]. The formal syntax for CTL is:

- Every atomic proposition p is a CTL formula.
- If f and g are CTL formulas, then so are $\neg f$, $f \wedge g$, AXf, EXf, $A[f\mathbf{U}g]$ and $E[f\mathbf{U}g]$.

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.

The symbols $\neg$ and $\wedge$ have their usual meanings, X is the *nexttime* operator; the formula AXf (EXf) intuitively means that f holds in every (in some) immediate successor of the current state. $\mathbf{U}$ is the *until* operator; the formula $A[f\mathbf{U}g]$ ($E[f\mathbf{U}g]$) intuitively means that for every computation path (for some computation path) there exists an initial prefix of the path such that g holds at the last state of the prefix and f holds at all other states along the prefix.

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 $\mathcal{M}, s \models \phi$ by $s \models \phi$):

1. If ϕ is the atomic proposition $v \in P$, then $s \models \phi$ if and only if $v \in L(s)$.
2. $s \models \neg\phi$ iff it is not the case that $s \models \phi$. $s \models \phi \wedge \psi$ iff $s \models \phi$ and $s \models \psi$.
3. $s \models AX\phi$ iff for all paths $\pi = s_0 s_1 s_2 ...$ starting at $s = s_0$, $s_1 \models \phi$.
4. $s \models EX\phi$ iff there exists a path $\pi = s_0 s_1 s_2 ...$ starting at $s = s_0$, such that $s_1 \models \phi$.
5. $s \models E[\phi \mathbf{U} \psi]$ 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 $j < i, s_j \models \phi$.
6. $s \models A[\phi \mathbf{U} \psi]$ iff for all paths $\pi = s_0 s_1 s_2 ...$ starting at $s = s_0$ and some $i \geq 0$ such that $s_i \models \psi, s_j \models \phi$ for all $j < i$.

The following abbreviatons are used in CTL formulas:

$AF(f) \equiv A[\text{true}\mathbf{U}f]$ intuitively means that f holds some time in the future along every path from the current state.

$EF(f) \equiv E[\text{true}\mathbf{U}f]$ means that there is some path from the current state to a state at which f holds.

$EG(f) \equiv \neg AF(\neg f)$ means that there is some path from the current state on which f holds at every state.

$AG(f) \equiv \neg EF(\neg f)$ means that f holds at every state on every path from the current state.

Some examples of CTL formulas are:

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

3 The Model Checking Algorithm

The model checking algorithm used here is described in [3, 6, 7]. This algorithm works on pure CTL formulas. We extend the model checker to handle the real-time operators discussed in [9].

BDD's

In this algorithm, boolean formulas are represented by Binary Decision Diagrams (BDD) [2]. BDD's are often substantially more compact than the traditional normal forms for representing boolean formulas, such as conjunctive normal form and disjunctive normal form.

Given BDD representations for formulas ϕ and ψ, there are efficient algorithms for computing the BDD representations of $\neg\phi$ and $\phi \wedge \psi$ [2], quantification over boolean variables and substitution of variable names [3]. These algorithms are the only ones needed to implement the algorithm that follows.

The fact that BDD manipulation is very efficient and it usually takes less space to represent boolean formulas allows the model checker algorithm to be used in a wider range of applications. This is because problems that are intractable using other representations can still be tractable using BDDs.

Representing the Model

The idea of model checking is to represent a reactive system as a model for some temporal logic, and to determine, using the data structure that encodes this model, if some assertion about the system is true or not.

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 BDD's to represent the labeled state-transition graph and to verify if the formula is true or not.

We want to represent the transition relation as a BDD. Assume that system behaviour 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. A transition (s_1, s_2) in N can be expressed as a conjunction of the state variables in V and V' and their negations. For example, if $V = \{p, q\}$ we may have in $s_1 : p, \overline{q}$ and in $s_2 : p, q$. Then the transition $i = (s_1, s_2)$ can be expressed as

$$t_i = p \wedge \overline{q} \wedge p' \wedge q'$$

Notice that we are representing 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, reduce the generality of the process, 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].

By writing formulas t_i that represent each transition i in the model we can represent the transition relation of the system as a disjunction of the form:

$$N(V, V') = \bigvee_i t_i$$

Model Checking

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

The procedure CHECK works in a bottom up fashion, starting from the atomic propositions in the formula, following the rules:

- CHECK(f), where f is an atomic proposition p, is the BDD that is true if and only if p is true.

- CHECK($\neg f$) and CHECK($f \wedge g$) are handled by the standard BDD algorithms for computing boolean connectives.

- CHECK(EXf) = CHECKEX(CHECK(f)).

- CHECK($E[f\mathbf{U}g]$) = CHECKEU(CHECK(f), CHECK(g)).

The procedures CHECKEX and CHECKEU will be explained shortly. Notice that AXf and $A[f\mathbf{U}g]$ can be rewritten using the above operators, so this definition of CHECK covers all CTL formulas.

CHECKEX(f) must verify if the formula f is true in a successor state of the current state. $EX(f)$ 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:

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

here $f\langle s\rangle$ 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 given in [3]. 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, so it is important to compute it in an efficient manner. Algorithms for this purpose are discussed in [5].

The formula $E[f\mathbf{U}g]$ is true at state s, if there exists a path beginning in s such that g is true in a future state t on the path, and f is true in all states between s and t. This means that either g is true, or f is true and there exists a successor state where $E[f\mathbf{U}g]$ is true. Consequently, the BDD that represents the states where $E[f\mathbf{U}g]$ is true can be computed by finding the least fixed point of:

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

The procedure CHECKEU computes the fixed point by iteration. It starts with $z = false$ and computes a new value for z using the formula:

$$z = g \vee (f \wedge \text{checkEX}(z))$$

The algorithm terminates when z does not change from one iteration to the next. Because the number of states is finite, we must eventually reach a fixed point.

4 Real-Time Logics

The logic CTL described previously 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 some bound 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 between a and b. Other temporal operators are defined in terms of the *bounded until.*

More formally, we extend our CTL semantics to include the *bounded until* by adding the following clauses to the formal semantics given in section 2:

7. $s \models E[\phi\mathbf{U}_{[a,b]}\psi]$ iff there exists a path $\pi = s_0s_1s_2...$ starting at $s = s_0$ and some i such that $a \leq i \leq b$ and $s_i \models \psi$ and for all $j < i, s_j \models \phi$.

8. $s \models A[\phi\mathbf{U}_{[a,b]}\psi]$ iff for all paths $\pi = s_0s_1s_2...$ starting at $s = s_0$ and some i such that $s_i \models \psi, a \leq i \leq b$, and $s_j \models \phi$ for all $j < i$.

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 $AG(p \rightarrow EF_{[0,3]}q)$. The bounded F operator is derived from the *bounded until* just as in the unbounded case, i.e. $EF_{[a,b]}f \equiv E[\text{true}\mathbf{U}_{[a,b]}f]$.

In order to implement this operator, we will use a procedure that is similar to the one described in section 3. It is easy to see that the formula $E[f\mathbf{U}_{[A,B]}g]$ can be expressed in the form:

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

Consider the first of these cases. We compute the sets of states where f is true for a steps. During this computation, a fixed point 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 fixed point is reached quickly. The same optimization can also applied in the second case. If a fixed point is reached before $(B - A)$ iterations we can immediately proceed to the third case.

5 Examples

In our implementation of the model checking algorithm we have two different languages. CTL is used to express properties of the model, and another language (the SMV language) is used to define the model. Although, this language was originally developed to represent digital circuits, it proved sufficient for representing real-time programs.

The fact that CTL is a discrete time logic raised some questions concerning the specification of practical real-time problems. We argue that discrete time is adequate to express most of the important properties we are interested in.

5.1 SMV syntax and semantics

This section briefly describes the SMV language in order to make it easier to understand the examples. One simple example expressed in this language is:

```
MODULE main
VAR
  request : boolean;
  state : {ready,busy};
ASSIGN
  init(request) := 0;
  init(state) := ready;
  next(request) := case
                     state = ready: {0, 1};
                     1: 0;
                   esac;
  next(state) := case
                   state = ready & request : busy;
                   1 : {ready,busy};
                 esac;
SPEC
  AG(request -> AF state = busy)
```

The VAR statement declares the boolean variables request and state (state is a scalar type, but is implemented as a boolean). The ASSIGN statement is used to specify the transition relation for the model. We define the initial value for the variables (using init()), and the value for variables after a transition of the system (using next()). The SPEC statement allows the program to be annotated by CTL formulas, and these formulas are checked during the verification procedure.

In this example, when a transition occurs, the state will become busy if it was ready before, and request was true. Otherwise, the value for state will be chosen nondeterministically among the values in the set {ready, busy} . In other words, if state is ready, and request is true then state will be busy. If those conditions are not satisfied, the guard in the last case statement option will be true and state will become either ready or busy nondeterministically.

The SMV compiler constructs a model for the program. The compiler assumes a synchronous semantics for the language, that is, all variables of a module transition at the same time. More than one module can be defined, but all of them transition at the same time by default. Nonsynchronized modules can also be defined using the **process** keyword. Process modules use an interleaving semantics, in which only one process transitions at each step. SMV chooses which process to transition nondeterministically. The FAIRNESS statement can be used to guarantee that the process will transition *infinitely often*. Arbitrary CTL formulas can be used as fairness constraints. A path is *fair* if each fairness constraint holds infinitely often in the path. The model checking algorithm only considers fair paths in determining whether a formula is true or not.

5.2 Bounded Producer Consumer

The first example is a bounded producer—consumer module. The producer produces only one item in any time interval of length x. The consumer takes at most y time units to consume an item from the moment it is produced. If $x \geq y$ the the producer will never overflow the buffer, and the consumer will never try to consume from an empty buffer.

```
-- producer will insert the next produced element position buffer[p],
-- and consumer will read the next element from position buffer[c].

MODULE producer(p, c)
VAR
  pstate: {pwait, prepare1, prepare2, produce};
ASSIGN
  init(pstate) := pwait;
  next(pstate) := case
                    pstate = pwait: {pwait, prepare1};
                    pstate = prepare1: prepare2;
                    pstate = prepare2: produce;
                    pstate = produce: pwait;
                    1: pstate;
                  esac;
  next(p) := case
               (pstate = produce): (p + 1) mod 2;
               1: p;
             esac;

MODULE consumer(p, c)
VAR
  cstate: {cwait, consume};
ASSIGN
  init(cstate) := cwait;
  next(cstate) := case
```

```
                    -- There are items to consume, Proceed.
                    (cstate = cwait) & !(p = c): consume;
                    cstate = consume: cwait;
                    1: cstate;
                  esac;
  next(c) := case
               (cstate = consume): (c + 1) mod 2;
               1: c;
             esac;

MODULE main
VAR
  p: {0, 1};
  c: {0, 1};
  prod: producer(p, c);
  cons: consumer(p, c);

ASSIGN
  init(p) := 0;
  init(c) := 0;

-- Consumer will not try to get an item from an empty buffer.
SPEC
  AG !((p = c) & (cons.cstate = consume))

-- Producer will not try to put an item into a full buffer.
SPEC
  AG !((p = ((c - 1) mod 2)) & (prod.pstate = produce))

-- Anything produced must be consumed in two states.
SPEC
  AG ((prod.pstate = produce) -> (ABG 2..2 (cons.cstate = consume)))

```

The producer can generate an item whenever it is idle; it does not check if the buffer is full. If **pstate = pwait**, the producer is idle. It can then continue waiting, or start producing, nondeterministically. We control the rate at which the producer generates data. We require that it takes three steps to produce an item (the producer goes through states **prepare1** and **prepare2** before **produce**).

The consumer, on the other hand, is fast, and consumes an item as soon as it is produced. Items are consumed two states after they are produced. For this reason, no overrun on the buffer occurs. The variables p and c point to the place in the buffer where a a new item should be inserted, or removed.

Although very simple, this example shows an important restriction on the definition of real-time models in SMV. Different modules in a system must be synchronous. This guarantees that each module advances one step at each transition of the system. If modules are not synchronized, there is no way of specifying when a module will make a transition in SMV, and therefore we cannot bound its execution time. In most SMV applications there is no need to bound the execution time, and starvation is avoided through a special fairness constraint that guarantees that unsynchronized modules are executed "infinitely often". Because of this constraint, our representation was not a realistic one since in "real" systems modules are not synchronized. Later we will show how to achieve the effect we desire using synchronous processes.

5.3 Mutual Exclusion

The second example implements a mutual exclusion module. It illustrates another problem associated with using synchronized modules, and one way to solve it. The problem is that, since all modules step together, they cannot write to the same variable (SMV forbids the possibility of two or more processes writing to the same variable at the same time). However, processes must share information, and therefore shared variables are essential if we want to use the program in real systems.

Notice that this mutex implementation works only for two processes. The extension for more processes is simple, but it will not be explored in this work. The example is:

```
MODULE mutex(state1, state2, owner)
VAR
  last: {1, 2};
ASSIGN
  init(last) := 1;
  init(owner) := 0;
  next(owner) :=
    case
      (state1=unlocked) & (state2=unlocked): 0;
      (state1=try_lock) & (state2=unlocked): 1;
      (state1=unlocked) & (state2=try_lock): 2;
      -- If both are trying to lock, we must wait until owner = 0,
      -- otherwise race conditions may occur, because there is one
      -- state between (owner := i) and (state_i = locked).
      (state1=try_lock) & (state2=try_lock) & (last=1) & (owner=0): 2;
      (state1=try_lock) & (state2=try_lock) & (last=2) & (owner=0): 1;
      1: owner;
    esac;
  next(last) := case
                  (owner = 1): 1;
                  (owner = 2): 2;
                  1: last;
                esac;
```

```

MODULE proc(id, state, owner)
VAR
  lock_time: 0..2;
ASSIGN
  init(state) := unlocked;
  init(lock_time) := 0;
  next(state) :=
             case
               state = unlocked: {unlocked, try_lock};
               (state=try_lock) & !(owner=id): try_lock;
               (state=try_lock) & (owner=id): locked;
               (state=locked)   & !(lock_time=2): {locked, unlocked};
               (state=locked)   & (lock_time=2): unlocked;
             esac;
  next(lock_time) := case
                      !(state = locked): 0;
                      1: (lock_time + 1) mod 3;
                    esac;

MODULE main
VAR
  state1: {unlocked, try_lock, locked};
  state2: {unlocked, try_lock, locked};
  owner: {0, 1, 2};
  p1: proc(1, state1, owner);
  p2: proc(2, state2, owner);
  m: mutex(state1, state2, owner);

-- Mutual exclusion is guaranteed.
SPEC
  !EF ((state1 = locked) & (state2 = locked))

-- There is no unbounded wait for a critical section.
SPEC
  AG (((state1 = try_lock) -> AF (state1 = locked)) &
      ((state2 = try_lock) -> AF (state2 = locked)))

-- There is no deadlock.
SPEC
  AG EF (state1 = locked)
SPEC
  AG EF (state2 = locked)

```

```
-- Critical section doesn't last more than 3 states.
SPEC
  AG ((state1 = locked) -> ABF 0..3 (state1 = unlocked))
```

Processes can be **unlocked**, **try_lock** or **locked** meaning that they don't want to enter the critical section, they do want to enter the critical section and that they are inside the critical section, respectively. The life of a process is simple, when it is unlocked, it can decide to try to lock at any time. When it is trying, it waits until it is allowed to lock, keeps it locked for a finite amount of time and goes back to unlocked.

Again, we have synchronized processes, but now we need to share one variable, which controls the mutual exclusion, the variable **owner**. Both processes change the value of **owner**, but they cannot write directly to the variable. The solution is to create another module, **mutex**, which writes to **owner**, and accepts requests for values.

Each process sends **mutex** its state. **mutex** knows that when a process is trying to lock the variable it wants to write its number on **owner** (this means that this process is allowed to lock). It then receives states from all processes, and decides which value to write on **owner**.

Inside the **mutex** module we model the arbitration that goes on in a real system whenever shared variables are accessed. This means that, although not as simple as just writing to a shared variable, this technique does not introduce more work to be done, it just makes it more explicit.

However, support from the definition language in defining and using shared variables would be very useful. The language could generate the control modules for each variable declared shared, and simplify the exchange of information. This has not yet been done.

6 Priority Inversion

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 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.

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 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 behaviour of the system becomes unpredictable.

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, 12].

The example coded below is slightly different from the one explained above. There is no scheduler choosing which process runs next, so all processes run concurrently. However, the mutex control module chooses which process will lock the variable next, and when there is contention, high priority processes are chosen first. In this case the queueing of processes in the mutex control module cause the priority inversion. In the example we have the same three processes as above, the *sensor*, the *analyzer* and the *reporter*, with high, medium and low priorities, respectively. Therc are also two mutex variables, $m1$ and $m2$, controlling two critical sections. The *sensor* wants the critical section controlled by $m1$, the *analyzer* wants the one controlled by $m2$, and the *reporter* locks both variables, first $m1$ and then $m2$. $m1$ controls access to the area shared by the *sensor* and the *reporter* as explained above. $m2$ controls a shared area where the *analyzer* puts its results. These results will in turn be read by the *reporter* to be printed on the screen. The sequence of events below leads to priority inversion:

1. The *reporter* locks $m1$.

2. The *analyzer* locks $m2$.

3. The *sensor* wants to lock $m1$, but it is already locked, so the *sensor* waits.

4. The *reporter* wants to lock $m2$, but it is locked, so the *reporter* waits.

5. The *analyzer* is continuously generating data, and after unlocking $m2$, it locks the mutex again to insert new data into the buffer. Notice that the *reporter* never locks $m2$, since it has lower priority than the *analyzer.*

6. The *sensor* is waiting for the *reporter*, and the *reporter* is waiting indefinitely for the *analyzer*. Therefore, the *sensor* is blocked by the *analyzer* indefinitely.

The solution works as follows. Since the *sensor* is waiting for the *reporter*, and the *sensor* has high priority, the task being executed by the *reporter* becomes a high priority task. We then make the *reporter* a high priority process temporarily, so it will release the lock the *sensor* wants faster. The *analyzer* eventually notices that the *reporter* has become a high priority process. At this point it will yield $m2$ to the *reporter*. After unlocking $m1$, the *reporter* will have its old priority restored.

The SMV code that implements this example is:

```
MODULE rt_mutex(state1, state2, owner, minherit)
VAR
  stut: 0..10; -- stuttering variable.
ASSIGN
  init(owner) := 0;
  init(stut) := 0;

  next(owner) :=
    case
      (state1=unlocked) & (state2=unlocked): 0;
      (state1=try_lock) & (state2=unlocked) & !(stut=0): {0, 1};
      (state1=try_lock) & (state2=unlocked) &  (stut=0): 1;
      (state1=unlocked) & (state2=try_lock) & !(stut=0): {0, 2};
      (state1=unlocked) & (state2=try_lock) &  (stut=0): 2;
      (state1=try_lock) & (state2=try_lock) & (owner=0) & !minherit: 1;
      (state1=try_lock) & (state2=try_lock) & (owner=0) &  minherit: 2;
      1: owner;
    esac;
  next(stut) := (stut + 1) mod 10;

```

The module rt_mutex is the real-time version of the mutex control module. It implements priorities among the processes. The basic difference between the two modules is that when both processes try to lock at the same time, in rt_mutex process 1 has precedence over process 2. In the mutex module of the previous example resources are granted in a first-come-first-served fashion. rt_mutex implements priority inheritance using a variable minherit. When minherit is false, operation is normal, and process 1 has precedence. However, when minherit is true, process 2 has precedence over process 1. Intuitively, minherit means that process 2 is in a high priority state, and therefore should not be placed in a queue.

The variable stut also plays an important part in this module. Priority inversion depends on a pathological ordering of events in the system. In SMV this ordering is usually avoided by the synchronization of modules, because there are implicit dependences caused by the fact that all modules step at the same time. However, in a real implementation these dependencies would not exist, and we must find a way of representing the correct behaviour of such a system. This is accomplished by using a technique that causes a transition to take an undetermined number of steps to occur. This technique is called *Stuttering*.

The variable **stut** is used to delay the entrance into the critical section for a nondeterministic number of steps, allowing other events to happen in the mean time. When a process wants the critical section, entering it can take from 1 to 10 steps, because of the `!(stut=0)` clause. The **(stut=0)** clause guarantees that the delay is finite.

We now describe the processes that use this mutex control module. As in the previous example, the processes' parameters include the process *id*, the mutex state, and the owner variables. However, some additional information is necessary. The **sensor** writes to **m2inherit**, telling the mutex control module when it is necessary to activate priority inheritance in the mutex **m2**. The decision is made based of information provided by **m1reporterstate**. This variable tells the **sensor** the state of the **reporter** with respect to the mutex variable **m1**. If the low priority process locked **m1** and the high priority process also wants to lock **m1**, then the **sensor** activates the priority inheritance mechanism (by setting **m2inherit**), because the **reporter** is now in a high priority state.

```
--
-- Sensor process.
--
-- The sensor process gathers data from radars while in state 0.
-- Then it locks m1, and puts data into shared buffer, while in
-- state 3. Finally, the sensor unlocks m1 and returns to state 0.
--
MODULE sensor(id1, m1state, owner1, m2inherit, m1reporterstate)
VAR
  state: 0..10;
ASSIGN
  init(state) := 0;
  init(m1state) := unlocked;
  init(m2inherit) := FALSE;

  next(state) :=
    case
      ----> state = 0, m1 unlocked, reading data from radars.
      state = 0 : {0, 1};
      ----> state = 1, try to lock m1.
      state = 1 : 2;
      (state = 2) & (m1state = try_lock): 2;
      (state = 2) & (m1state = locked)  : 3;
      ----> state = 3, m1 locked, put data into shared buffer.
      state = 3 : 4;
      state = 4 : 0;           -- unlock m1.
      1: state;
    esac;

```

Notice that **state** works as a program counter. The current state of the system defines the next task to be executed. For example, state 0 is the starting state of the program, state 1 signals that the process wants to lock **m1**. In state 2 we wait until **m1** is locked, and state 3 corresponds to the code in the critical section. Finally, state 4 means that **m1** should be unlocked.

```
  next(m1state) :=
           case
             (m1state = unlocked) & (state = 1)  : unlocked;
             (m1state = unlocked) & (state = 2)  : try_lock;
             (m1state = try_lock) & !(owner1=id1): try_lock;
             (m1state = try_lock) &  (owner1=id1): locked;
             (m1state = locked)   & (state = 3)  : locked;
             (m1state = locked)   & (state = 4)  : unlocked;
             1: m1state;
           esac;

```

Based on what each state number means, we set **m1state** accordingly. **m1state** and **state** are coordinated so that state changes depend on both states. This separation of process state and mutex state improves the modularity of the system and makes it simpler to write and modify programs. The same coordination between process state and mutex state is also used in the other processes.

```
  -- If the sensor wants to lock m1, but reporter locked it, then make
  -- sure the reporter inherits sensor's priority while accessing m2.
  next(m2inherit) :=
    case
      (m1state = try_lock) & (m1reporterstate = locked): TRUE;
      1: FALSE;
    esac;

--
-- Analyzer process.
--
-- While in state 0, analyzer collects and analyzes data.
-- It proceeds to lock m2, and, in state 3, sends the results to
-- the reporter process. Then the analyzer unlocks m2, and repeats.
--
MODULE analyzer(id2, m2state, owner2, m1inherit)
VAR
  state: 0..10;
ASSIGN
  init(state) := 0;
  init(m2state) := unlocked;
init(m1inherit) := FALSE;

next(state) :=
  case
    ----> state = 0, m2 unlocked, read data and analyze it.
    state = 0: {0, 1};
    ----> state = 1, try to lock m2.
    (state = 1): 2;
    (state = 2) & (m2state = try_lock): 2;
    (state = 2) & (m2state = locked)  : 3;
    ----> state = 3, m2 locked, send data to reporter.
    state = 3: 4;
    state = 4: 0;            -- unlock m2.
    1: state;
  esac;

next(m2state) :=
        case
          (m2state = unlocked) & (state = 1)  : unlocked;
          (m2state = unlocked) & (state = 2)  : try_lock;
          (m2state = try_lock) & !(owner2=id2): try_lock;
          (m2state = try_lock) &  (owner2=id2): locked;
          (m2state = locked)   & (state = 3)  : locked;
          (m2state = locked)   & (state = 4)  : unlocked;
          1: m2state;
        esac;

next(m1inherit) := FALSE; -- We don't  worry about analyzer's
                          -- starvation in this example.

```

The reporter is the most complicated process. We must double the number of process states, because now we want to lock both mutex variables. The order in which these variables are locked is determined by the order in which the state changes. The reporter starts in state 0, and may decide at any time to lock m1 by going to state 1. After m1 is locked, the *reporter* jumps to state 5, and will try to lock m2. When m2 is also locked, the reporter goes to state 8 and thereby unlocks m2. Finally, the *reporter* goes to state 4 and unlocks m1. These state changes simulate the code:

```
...
lock(m1);
lock(m2);
unlock(m2);
unlock(m1);
...
```

```
--
-- Reporter process.
--
-- While in state 0, the reporter updates the screens, and does
-- non critical work. Then it locks m1 to get data from the
-- sensor (state 3), and locks m2 to get data from the analyzer
-- (state 7). Notice that we assume that some property about the
-- data from the sensor and the analyzer must be preserved by
-- getting analyzer data while m1 is locked.
--
MODULE reporter(id1, id2, m1state, m2state, owner1, owner2)
VAR
  state: 0..10;
ASSIGN
  init(state) := 0;
  init(m1state) := unlocked;
  init(m2state) := unlocked;

  -- states: 0: choose when to lock
  --         1: unlocked for m1
  --         2: try_lock for m1
  --         3: locked for m1
  --         4: unlocked m1
  --         5: unlocked for m2
  --         6: try_lock for m2
  --         7: locked for m2
  --         8: unlocked m2
  next(state) := case
                   ----> state 0, m1 unlocked, update screens.
                   state = 0: {0, 1};
                   ----> state = 1, try to lock m1.
                   state = 1: 2;
                   (state = 2) & (m1state = try_lock): 2;
                   (state = 2) & (m1state = locked)  : 3;
                   ----> state 3, m1 locked, get data from sensor.
                   state = 3: 5;
                   state = 4: 0;            -- unlock m1.
                   ----> state = 5, try to lock m2.
                   state = 5: 6;
                   (state = 6) & (m2state = try_lock): 6;
                   (state = 6) & (m2state = locked)  : 7;
                   ----> state = 7, m2 locked, get data from analyzer.
                   (state = 7): 8;
                   (state = 8): 4;     -- unlock m2.
                   1: state;
                 esac;

  next(m1state) :=
           case
             (m1state = unlocked) & (state = 1)  : unlocked;
             (m1state = unlocked) & (state = 2)  : try_lock;
             (m1state = try_lock) & !(owner1=id1): try_lock;
             (m1state = try_lock) &  (owner1=id1): locked;
             (m1state = locked)   & (state = 3)  : locked;
             (m1state = locked)   & (state = 4)  : unlocked;
             1: m1state;
           esac;

  next(m2state) :=
           case
             (m2state = unlocked) & (state = 5)  : unlocked;
             (m2state = unlocked) & (state = 6)  : try_lock;
             (m2state = try_lock) & !(owner2=id2): try_lock;
             (m2state = try_lock) &  (owner2=id2): locked;
             (m2state = locked)   & (state = 7)  : locked;
             (m2state = locked)   & (state = 8)  : unlocked;
             1: m2state;
           esac;

MODULE main
VAR
  m1state1: {unlocked, try_lock, locked};
  m1state2: {unlocked, try_lock, locked};
  m1inherit: boolean;
  m2state1: {unlocked, try_lock, locked};
  m2state2: {unlocked, try_lock, locked};
  m2inherit: boolean;

  owner1: {0, 1, 2};
  owner2: {0, 1, 2};
  psensor  : sensor(1, m1state1, owner1, m2inherit, m1state2);
  panalyzer: analyzer(1, m2state1, owner2, m1inherit);
  preporter: reporter(2, 2, m1state2, m2state2, owner1, owner2);
  m1: rt_mutex(m1state1, m1state2, owner1, m1inherit);
  m2: rt_mutex(m2state1, m2state2, owner2, m2inherit);

-- Sensor process cannot starve.
```

```
--
-- We can bound the time sensor has to wait for m1. We know
-- that in at most 32 states after deciding to lock, the sensor
-- will succeed.
SPEC
  AG ((psensor.state = 2) -> ABF 0..32 (psensor.state = 3))

-- Notice that this is not true for everyone.
-- For example, the reporter can even starve.
SPEC
  EF EG (preporter.state = 2)

```

From these examples we have been able to draw two main conclusions. Although we work with discrete time and synchronized modules, most systems that occur in practice can be modelled. Stuttering can be used to introduce asynchronous behaviour, and to achieve a finer granularity of time. This flexibility makes it possible to specify systems in SMV that would be difficult to express otherwise and overcomes many of the limitations that arise because of synchronization and the discrete model of time that is used. However, the use of stuttering states may increase the size of the model. This may prove to be a problem in large systems.

In the course of writing these programs, we have observed a number of ways that the SMV language could be improved. The fact that we need to introduce stuttering states explicitly makes it more complicated to specify the model. Support for stuttering could be built into the language for describing state transition systems. For example, "stuttering next" might be defined as in **stut_next(a, 10) := b**. This construct would mean that the next state of variable **a** is **b** and it may take from 1 to 10 states for the value to change. Another possible improvement comes from the fact that the SMV language was designed for specifying circuits. Different approaches are used for specifying circuits and writing programs. Writing a program in the SMV language is more complex than defining a circuit. A language with a syntax closer to that of a general programming language could increase the efficiency of the verification of programs.

7 Conclusions

In this paper we have discussed the priority inversion problem. We formalized a solution for a particular instance of this problem and verified that it was correct using temporal logic model checking techniques. The solution that we implemented, *basic priority inheritance*, solves the unbounded priority inversion problem in general. The changes to our basic model that are necessary to represent more general versions of the problem are simple, since no particular restriction was imposed on the model.

This work also demonstrates that non-trivial properties of real-time systems can be proven using symbolic model checking techniques. The time to construct the model and verify both properties of the example described in section 6 was less than three minutes on a i486-based workstation. Approximately 90K BDD nodes were allocated. The transition relation itself required about 18K nodes.

We extended the original CTL model checker to handle properties that are bounded in time. The *bounded until* operator was implemented to allow the expression of such properties. During the construction of the models used in this work, however, some other problems had to be solved to make the new n odel checker more useful. The main problem we faced was that all modules had to be synchronized. This introduced a severe constraint on the systems that could be directly verified by SMV. The use of stuttering states, however, alleviated this problem. We believe that the same idea can be used to obtain representations for many other practical examples as well. In spite of using synchronized modules and discrete time it appears possible to handle most systems that arise in practice without the added complexity caused by the use of continuous time.

Nevertheless, additional research is needed. Stuttering states can increase the size of the model and make it more complicated to define the model. Thus, special techniques for handling stuttering states could help decrease the size of the model and reduce the possibility of state explosion. Finally, research is needed to define a better model definition language that can hide some of the aspects of stuttering.

Acknowledgments

I would like to thank Edmund Clarke for many of the ideas in the paper, and suggestions on how to improve it.

References

[1] Alur, R. and Henzinger, T.A. Logics and models of real-time: a survey. In: *Lecture Notes in Computer Science, Real Time: Theory in Practice*. Springer-Verlag, 1992.

[2] Bryant, R.E. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8), 1986.

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

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

[5] Burch, J.R., Clarke, E.M., Long, D.E. Symbolic model checking with partitioned transition relations. *VLSI 91*, Edinburgh, Scotland, 1991.

[6] Clarke, E.M., Emerson, E.A., Sistla, A.P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Prog. Lang and Syst.*, april 1986, pp. 244–263.

[7] Clarke, E.M., Burch, J.R., Grumberg, O., Long, D.E., McMillan, K.L. *Automatic verification of sequential circuit designs.* Roya! Society of London, Oct. 1991.

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

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

[10] Emerson, E.A. Temporal and modal logic. In: *Handbook of Theoretical Computer Science.* Elsevier Science Publishers, B.V., 1990.

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

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