# Shared Variables and Efficient Synchronization Primitives for Synchronous Symbolic Verifiers

*S. V. Campos*[1]
*Univ. Federal de Minas Gerais, DCC*
*Belo Horizonte — MG — Brasil*
*scampos@dcc.ufmg.br*

**Abstract**

In spite of the increasingly frequent application of formal verification tools today they still are difficult to use for non trivial verification tasks. One reason is that the specification languages used by these systems are still quite different than the programming languages designers use. One significant difference is the absence of shared variables for synchronous verifiers. Many systems use shared variables in their implementation, but most synchronous formal verification tools do not model them. Instead the user must use additional variables to maintain the illusion of shared memory and model the correct behavior. Unfortunately this can cause a significant overhead making the final model much larger than necessary. This work proposes a new technique called *BDD markings* to overcome this problem. BDD markings provide a way of keeping track of control flow information and have allowed us to implement shared variables in Verus, a synchronous symbolic verifier. Using the new method we have been able to verify a variant of a mutual exclusion algorithm used in the Mach operating system and to determine the existence of priority inversion in a complex real-time system. Priority inversion is a serious and subtle problem that can cause real-time systems to fail unexpectedly. Its importance has been highlighted recently when it was discovered that priority inversion has caused system shutdowns in the Pathfinder spacecraft soon after it started collecting Martian meteorological data. This example has been modeled with and without shared variables. The new method has allowed verification to be performed up to 10 times faster and using up to 20 times less memory, demonstrating the efficiency of the technique proposed.

**Keywords**

Symbolic model checking, synchronization variables, BDD, priority inversion

1. This research was sponsored in part by Conselho Nacional de Pesquisa e Desenvolvimento, CNPq.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35394-4_29

S. Budkowski et al. (eds.), *Formal Description Techniques and Protocol Specification, Testing and Verification*

## 1 Introduction

Formal verification tools are used more frequently today than ever before and their application is increasing even more. However, using such systems is still not as simple as using most other tools such as compilers or debuggers. Several reasons make it difficult to employ formal verification tools in practice such as the complexity of the verification algorithms: for example, careless coding can easily lead to state explosion. Another reason is that most languages used to model systems are quite different than the languages used to implement them. Languages such as Esterel (Berry, 1992), Promela (Holzmann, 1996) and Modechart (Jahanian, 1988) have very different syntaxes or semantics than regular programming languages making it difficult to translate a program from C or a similar language to one of these languages.

We have previously addressed this problem by defining a new language called *Verus*, integrated into the Verus verification system (Campos, 1996a and 1995c). The Verus language is an imperative 'C-like' language that has been designed to allow straightforward modeling of systems being verified. Using this tool, the system being verified is compiled into a state-transition graph. The model uses a discrete model of time, in which each transition corresponds to one time unit. A symbolic model checker allows the verification of untimed properties expressed in CTL (Clarke, 1986 and McMillan, 1992) and of time bounded properties using RTCTL model checking (Emerson, 1990). Moreover, algorithms derived from symbolic model checking are used to compute *quantitative* information about the model (Campos, 1996a; 1995a and 1995c). The information produced allows the user to check the temporal correctness of the model as well as determine reaction times to events and several other system parameters. This information provides insight into the behavior of the system and in many cases it can help identify inefficiencies and suggest optimizations to the design. Verus has been used to verify several large complex systems such as aircraft (Campos, 1994), robotics (Campos, 1995c) and medical controllers (Campos, 1995a), as well as the PCI local bus (Campos, 1995b).

However, a Verus program still behaves quite differently than a similar C program. Some differences are needed in order to allow an efficient verification, such as the granularity of discrete time used in Verus. Others have been artificially imposed, and make programming in Verus more difficult than it should be. One of these differences is the lack of shared variables. In Verus, as well as most other synchronous verification tools, multiple processes can be defined. The values of variables can be read by all processes, but they are "owned" by one process, that is, variables can be modified only by that process. The original release of Verus as well as other verifiers such as SMV (McMillan, 1992), CV (Deharbe) and VIS (Brayton, 1996) have this restriction when being used in synchronous mode. This restriction is a significant hurdle for many applications because shared variables are extensively used in the actual code. Implementing an application that uses shared variables in a system that does not allow them is often a complex task. Moreover, it not only increases modeling time but also makes the final model much larger, since additional variables (and even pro-

cesses) may be needed to model the correct behavior. Shared variables are present in asynchronous verifiers, but certain types of systems such as real-time systems cannot be easily verified by such tools.

This work describes a new technique used to implement shared variables in Verus, *BDD Markings*. In the new method BDD variables mark specific points in the control flow of the program in order to identify where variable assignments from multiple processes can potentially conflict. An important advantage of the method is that once the transition relation for the model has been constructed, the variables introduced to mark the control flow can be existentially quantified out. In this way the technique does not impose an overhead in the model because the new variables are removed before the transition relation is constructed.

BDD markings also allow the determination of which transitions in the state-transition graph correspond to given points in the program. This can be used to implement a new feature which allows properties to refer directly to specific statements in the source code. Properties often refer to points in the control flow where some computation is performed. Without BDD markings additional variables have to be declared to keep track of this information. These variables make the final model larger and more expensive to verify. This overhead can be avoided using the method proposed.

We have used the new method to verify two examples, a mutual exclusion and the priority inversion example described in (Campos, 1996b). The mutual exclusion algorithm is particularly appropriate since it relies on shared variables for its correctness. A variant of this algorithm has been used in the Mach operating system kernel (Rashid, 1989). The priority inversion example has previously been implemented without shared variables. Its importance has been highlighted recently when it was shown that priority inversion caused the sporadic system shutdowns experienced by the Pathfinder spacecraft soon after it started collecting Martian meteorological data (Wilner, 1997). By revisiting the example we have been able to compare both alternatives and to see that the shared variable model has performed significantly better. In some cases the new model executes 10 times faster using 20 times less memory!

The paper is organized as follows: we first briefly introduce Verus in section 2. Section 3 discusses how a Verus program can be represented using BDDs. In section 4 we present BDD markings and in section 5 we show how they can be used to implement shared variables. Finally sections 6 and 7 describe the examples implemented using the new method, and section 8 concludes the paper.

## 2 The Verus tool

Verus is a formal verification tool used to verify finite-state real-time systems that can also be applied to systems without real-time constraints. The application being verified is modeled in the Verus language and compiled into a state-transition graph using symbolic algorithms and represented using BDDs. Model checking and quantitative timing algorithms are then applied to the model in order to determine its correctness as well as its timing characteristics. Verus is particularly suited to model and verify

reactive systems defined by a set of concurrent processes. Even though shared variables are used to model communication between processes, this does not restrict the application of the tool. Distributed systems can be efficiently verified by modeling messages as shared variables and the exchange of messages as changes in the values of the corresponding variables.

### *The Verus Language*

The main goal of the Verus language is to allow engineers and designers to describe real-time systems easily and efficiently. It is an imperative language with a syntax resembling that of C. Special primitives are provided for the expression of timing aspects such as deadlines, priorities, and time delays. These primitives make timing assumptions explicit, leading to clearer and more complete specifications. The data types allowed in Verus are fixed-width integer and boolean. Nondeterminism is supported, which allows partial specifications to be described. Further details about the Verus language including its formal semantics can be found in (Campos, 1996a).

A fragment of a simple real-time program is used to give an overview of the language. This program implements a solution for the producer-consumer problem by bounding the time delays of its processes. The code for the `producer` process is shown below. Variable `p` is a pointer to the buffer in which data is stored. The `producer` initializes its pointer `p` to 0 and the `produce` variable to *false*. It then enters a nonterminating loop in which items are produced at a certain rate. Line 7 introduces a time delay of 3 units, after which an item will be produced. Line 8 marks the production of an item by asserting `produce`. In line 9 the pointer is updated appropriately. Line 10 makes sure that the event `produce` is observed. It is needed because the state of a Verus program can only be observed at `wait` statements, since all other statements execute in time zero. This allows a more accurate control of time, eliminating the possibility of implicit delays influencing verification results. It also generates smaller models, since contiguous statements are collapsed into one transition.

```
1   producer(p)
2   {
3     boolean produce;
4     p = 0;
5     produce = false;
6     while(!stop) {
7       wait(3);
8       produce = true;
9       p = p + 1;
10      wait(1);
11      produce = false;
12    };
13  }
```

**Figure 1** Producer code.

```
16  consumer(p, c)
17  {
18    boolean consume;
19    c = 0;
20    consume = false;
21    while (!stop) {
22      wait(1);
23      if (p != c) {
24        consume = true;
25        c = c + 1;
26        wait(1);
27        consume = false;
28      };
29    };
30  }
```

**Figure 2** Consumer code

The consumer code shown above is similar to the producer. the main differences are that it must check to be sure that there are items to be consumed (`p != c`), and that it takes less time to execute than the producer. The fact that the consumer is faster than the producer ensures that the buffer will not overflow, as will be verified later.

```
28  main()
29  {
30    int p, c;
31
32    process prod producer(p),
33            cons consumer(p, c);
34
35    spec MIN[prod.produce, cons.consume]
36         MAX[prod.produce, cons.consume]
37         AG(prod.produce -> AF cons.consume)
38  }
```

**Figure 3** Producer/consumer main function

As in the C language, `main` has a special function in Verus. In this function all processes are instantiated, and global variables can be declared. The variables `p` and `c` (used as pointers in the buffer) are declared and the `producer` and `consumer` processes are instantiated in the `main` function of the example code.

Process instantiation in Verus follows a synchronous model. All processes execute in lock step, with one step in any process corresponding to one step in the other processes. Asynchronous behavior can be modeled by using *stuttering*, as described in (Campos, 1996a). An implicit instantiation of the `main` module is assumed, where the code in `main` executes as another synchronous module.

Specifications can also follow the code as can be seen. The specifications above compute the minimum and maximum time between producing an item and consuming it, as well as checking that a `produce` is always followed by a `consume`.

### *CTL and RTCTL Model Checking*

Verus allows the verification of untimed properties expressed as CTL formulas (Clarke, 1986 and McMillan 1992) as well as of timed properties expressed as RTCTL formulas (Emerson, 1990). CTL formulas allow the verification of properties such "*p* will eventually occur", or "*p* will never be asserted". However, it is not possible to express bounded properties such as "*p* will occur in less than 10ms" directly. RTCTL model checking overcomes this restriction by allowing bounds on all CTL operators to be specified.

### *Quantitative Algorithms*

Most verification algorithms assume that timing constraints are given explicitly. Typically, the designer provides a constraint on response time for some operation, and the verifier automatically determines if it is satisfied or not. Unfortunately, these techniques do not provide any information about how much a system deviates from its expected performance, although this information can be extremely useful in fine-tuning the behavior of the system.

Verus implements algorithms that determine the minimum and maximum length of all paths leading from a set of starting states to a set of final states. It also has algorithms that calculate the minimum and the maximum number of times a specified condition can hold on a path from a set of starting states to a set of final states. Our algorithms provide insight into *how well* a system works, rather than just determining whether it works at all. They enable a designer to determine the timing characteristics of a complex system given the timing parameters of its components. This information is especially useful in the early phases of system design, when it can be used to establish how changes in a parameter affect the global system behavior.

Several types of information can be produced by this method. Response time to events is computed by making the set of starting states correspond to the event, and the set of final states correspond to the response. Schedulability analysis can be done by computing the response time of each process in the system, and comparing it to the process deadline. Performance can be determined in a similar way. The algorithms have been used to verify several real-time and non real-time systems.

## 3 FROM A VERUS PROGRAM TO A STATE-TRANSITION GRAPH

### *Symbolic Representation*

States are defined by the assignment of values to program variables, where each possible assignment to the program variables is a state. Boolean formulas over program

variables can be true or not in a given state. The value of a boolean formula in a state is obtained by substituting into the formula the values of the variables in that state. In general, sets of states can be represented by boolean formulas, where each formula represents the set of states in which the formula is true.

Transitions can also be represented by boolean formulas. A transition $s \rightarrow s'$ is represented using two sets of variables, one for the current state and another for the next state. If state $s$ is represented by the formula $f_s$ over the current state variables, and state $s'$ is represented by formula $f_{s'}$ over the next state variables, then the transition $s \rightarrow s'$ is represented by the formula $f_s \wedge f_{s'}$. The transition relation of a graph is the disjunction of all transitions in the graph. The meaning of the formula representing the transition relation is the following: there exists a transition from $s$ to $s'$ iff the substitution of the variable values for $s$ in the current state variables and $s'$ in the next state variables of the transition relation yields *true*.

### *Tracking the Control Flow — Wait Graphs*

The execution of a Verus statement may change the value of one or more program variables. In general, it changes a given state $s$ into another state $s'$. Executing a sequence of statements in a given state then generates a sequence of states. However, in Verus not all of those states are observable. The state of the program can only be observed at `wait` statements. When a `wait` is executed all changes caused by the execution of the block of statements since the previous `wait` take effect at the same time. Transitions in the graph occur only when `wait` statements are executed. Each transition in the graph corresponds to time elapsing by one unit. The statement `wait(1)` corresponds to one transition in the graph. Longer waits are modeled by a sequence of unit transitions.

It is easier to understand the behavior of a Verus program by concentrating on its `wait` statements. This can be accomplished by translating the program into a *wait graph*. The wait graph corresponding to a Verus program is a graph in which the states are the `wait` statements in the program. It corresponds to an intermediate representation between the Verus program and the corresponding state-transition graph. It is used only to illustrate how this translation occurs and is not actually constructed.

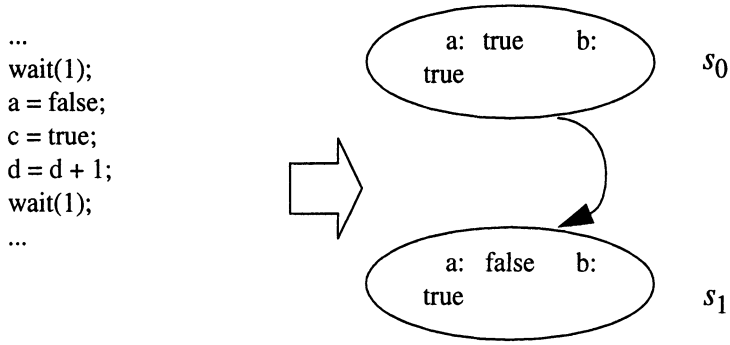


**Figure 4** If $s_0$ is the current state at the first `wait`, $s_1$ will be the current state at the second.

Transitions between `waits` are defined as follows. A transition between $\texttt{wait}_i$ and $\texttt{wait}_j$ ($\texttt{wait}_i$ represents the $i^{th}$ occurrence of a `wait` statement in the program) exists iff $\texttt{wait}_j$ can be reached from $\texttt{wait}_i$ in the control flow of the program without going through intermediate `waits`. The edges of the wait graph are labelled by a relation $T_{ij}$ between any two states in the state-transition graph. This relation describes the state changes caused by the execution of the statements between the corresponding `waits`. It is determined during compilation in the following way:

- When a `wait` statement is parsed an initial relation is set up. It basically states that no variables change value (variables only change value when assigned to): $(v = v')$; and that the next transitions that will be generated will start on this `wait` statement (a wait counter variable is introduced to maintain this information): $(wc = i)$.
- Each statement parsed changes the intermediate relation computed by the previous statement according to what the statement does, and produces a new intermediate relation.

The construction of this relation is described in detail in (Campos, 1996a). Intuitively, given two states $s$ and $s'$, $T_{ij}(s, s')$ means that if the execution of the program is in $\texttt{wait}_i$ and the current state is $s$, then there exists a path in the control flow leading to $\texttt{wait}_j$ (without intermediate waits) and the execution of the statements on this path will change state $s$ into state $s'$. The construction of the relation between wait statements is complex and its explanation is beyond the scope of this paper. However, figure 5 can give an intuition on how these relations are constructed.

## 4 BDD MARKINGS

Let's assume we want to implement a primitive called `assert`. The statement `assert(`*name*`)` can be inserted at any point in the program. Its semantics corre-

spond to creating a variable *name* in the model such that *name* is true in exactly those states reached immediately after executing the `assert` statement.

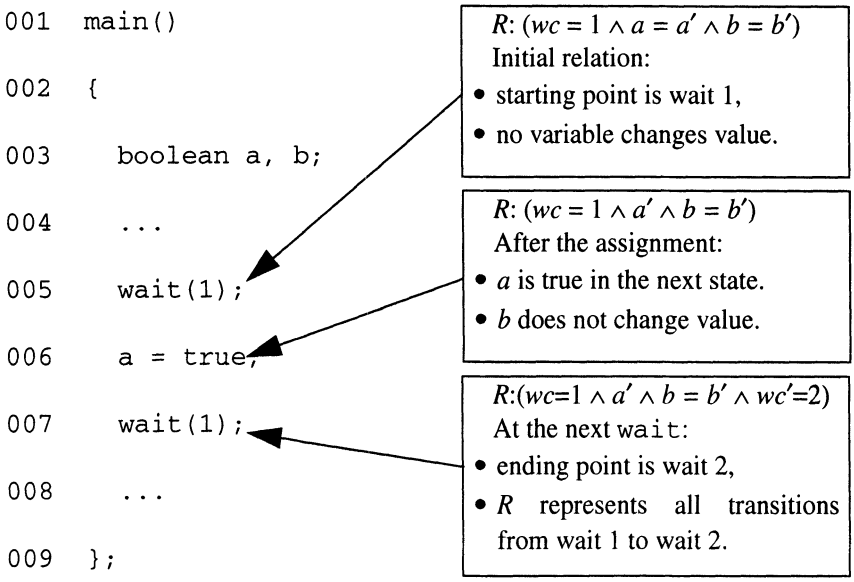


**Figure 5** Constructing the transition relation for a Verus program.

Implementing `assert` requires a method to keep track of the control flow of the program. This is not always an easy task. It is sometimes difficult to determine if (or when) a program executes specific statements. Traditionally a "printf" is inserted in the program, and if the control flow goes through that point a message shows up on the screen. Unfortunately it is not possible to use this technique in Verus, since it does not follow a single execution path, but rather explores all of them. A possible solution in Verus is to insert a `wait` statement at the point of interest in the program and check to see if that `wait` statement can be reached. However, this technique affects the time delays in the program and can change its behavior. BDD markings provide a solution for this problem. A BDD variable is used to mark the transitions that correspond to the execution of `assert` in the following way:

1. Create a BDD variable *name* and make it false in every transition by default. This is accomplished by inserting the clause $\neg name$ in the initial transition relation created at the `wait` statement.
2. When the statement `assert(`*name*`)` is found, assign the value *true* to *name* in the current relation, overriding the clause $\neg name$.
3. Whenever a `wait` statement is parsed, check to see if there are subsets of the current relation that satisfy *name*. Those transitions are the ones that traverse `assert` in the source code.

Step 2 ensures that all transitions that execute `assert` are marked with the variable *name*. Step 1 guarantees that the transitions that do not execute `assert` are not marked. Step 3 is necessary because transitions in the model are created only at `wait` statements. Before reaching the next `wait` statement the current relation describes a partial transition, whose end state may never be reached because the statements following `assert` may change the relation and consequently its end state. So, only when reaching the next `wait` statement we can be sure to have determined a transition that executes `assert`.

The transitions mentioned in step 3 above represent the execution of `assert` in the program. The subset of $R$ satisfying *name* at that point, $R'$, is a formula that expresses when `assert` is traversed. We can then save this formula and associate with it the symbol *name*. Then, the BDD variable *name* can be quantified out of $R'$, since it is no longer needed. References to *name* will point to the formula that represents it. In this way we can determine when a statement is executed in the program without adding extra BDD variables to the model.

The primitive `assert` can be implemented in this way, and it can be used to express properties about the control flow of the program. This is more efficient than the method used otherwise, which is to create additional program variables and assign values to them according to the point in the program that is being executed. The same method is also used to implement shared variables as described in the next section.

## 5 IMPLEMENTING SHARED VARIABLES

In most languages used by synchronous formal verification tools variables are "owned" by processes, that is, they can be read by all processes, but can only be changed by a single process. However, this is an unrealistic assumption, "shared" variables that can be modified by all processes are very frequently used. In fact, many applications rely on shared variables in their implementations, for example, to synchronize processes. But their behavior is difficult to model in symbolic verifiers, because concurrent assignments are hard to identify and model correctly using synchronous composition. Imperative languages in most cases pose yet another problem. The reason is that in those languages when a variable is not assigned to it maintains its value. Unfortunately this means that there is an implicit assignment in every transition that does not mention that variable. Because of this, shared variables are usually not modeled in symbolic verifiers. Unfortunately, modeling a system that uses shared variables in a language that does not allow them is frequently difficult and usually creates a significant overhead because additional variables (and sometimes processes) have to be created, making the final model significantly larger than it needs to be.

Both types of conflict described above are present in Verus. The implicit assignment problem is caused by the clause $v = v'$ inserted in all transitions that do not assign val-

ues to $v$. It maintains the value of variables that have not been assigned to. However, this is no longer valid when using shared variables because a transition that does assign a value to $v$ from another process may be executed concurrently, causing a conflict. In this case the conflict eliminates the global transition from the transition relation, because synchronous composition conjuncts transitions from concurrent processes (Campos, 1996a and McMillan 1992), and the conjunction of the formulas corresponding to the conflicting assignments yields *false*.

To avoid this problem we create a BDD variable $assigned_v$ for each shared variable $v$. This variable is used to mark where assignments to $v$ occur. We proceed by implicitly inserting an `assert(`$assigned_v$`)` everywhere assignments to $v$ are made. The call to `assert` does not actually exist in the source code, but it is modeled in the compiler. At the `wait` statements we "collect" the transitions that assign values to $v$ by disjuncting them into a partial transition relation $TR_v$. At the same time, we eliminate the clause $v = v'$ from all transitions that do not assign values to $v$. After compiling all processes we compute the global transition relation (without the $v = v'$ clause) and in the end conjunct the clause $(v = v' \vee TR_v)$ to the global transition relation. This clause makes sure that for transitions that do not assign values to $v$ from any process ($TR_v$ is *false*) the value of $v$ is maintained ($v = v'$). For transitions that do assign values to $v$ ($TR_v$ true), the clause $v = v'$ does not necessarily apply and the new value does not conflict with the previous one.

The method described above implements shared variables in Verus, but it does not handle explicit concurrent assignments. They still generate a conflict, eliminating the transition from the final model. Consequently, conflicting assignments potentially generate states that have no outgoing transition. Verus checks for the presence of these states and prints a counterexample showing how conflicting assignments can be reached. Racing conditions in the system can be identified this way.

However, sometimes conflicting assignments are necessary. They can be modeled by noticing the following: In most cases when a process writes to a shared variable it must foresee the possibility of not succeeding, that is, another process may overwrite the value written. We can model this behavior by allowing the possibility of *not* writing to the variable and therefore avoiding the conflict. As shown in figure 6 we can model conflicting accesses using a variable `random` (an `extern` variable which changes value nondeterministically (Campos, 1996a)) to allow the process to avoid writing.

## 6 EXAMPLE: FAST MUTUAL EXCLUSION

We have used the method described above to verify Lamport's fast mutual exclusion algorithm (Lamport, 1987) shown in figure 7 below. A variant of this algorithm is used in the Mach operating system kernel (Rashid, 1989) to provide mutual exclusion

in those systems that have no hardware assistance to this end. In this algorithm two variables are used to achieve mutual exclusion. Processes try to write their own value on both variables. A process gains access to the `mutex` area when it succeeds in writing to both variables. If it does not succeed, it tries again. The advantage of this algorithm is that it provides a simple and efficient way of implementing mutual exclusion without hardware assistance. The disadvantage is that it may cause starvation. We have been able to show that the algorithm is correct and that it indeed can cause starvation, as foreseen. However, we have also been able to determine a property that was not described anywhere else. Starvation only occurs when more than two processes compete. For a two process system there is no starvation.

Original program:

```
...
while (M != 1) {
  M = 1;
  delay();
};
...
```

Verus program:

```
...
while (M != 1) {
  if (random) M = 1;
  wait(1);
};
...
```

**Figure 6** Modeling conflicting accesses in Verus

```
start:  x = i;
        if (y != 0) goto start;
        y = i;
        if (x != i)
          delay();
          if (y != i) goto start;
        mutex();
        y = 0;
```

**Figure 7** Lamport's fast mutual exclusion algorithm

## 7 EXAMPLE: PRIORITY INVERSION

We introduce a hypothetical air-traffic control system to illustrate how priority inversion can affect a real-time system and cause it to become unschedulable. It concentrates on 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, the *reporter*, reads the data collected by the *sensor*, and updates the traffic controller screens. The *sensor* is a high priority process since 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 tasks. The *sensor* and the *reporter* share data, accessed in mutual exclusion.

However, this may result in priority inversion, as shown in the following scenario. 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 screens. 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. The behavior of the system becomes unpredictable. This unbounded priority inversion can be seen in figure 8.

Priority inheritance protocols are one way of preventing unbounded priority inversions (Rajkumar, 1989). 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. In our example, 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.

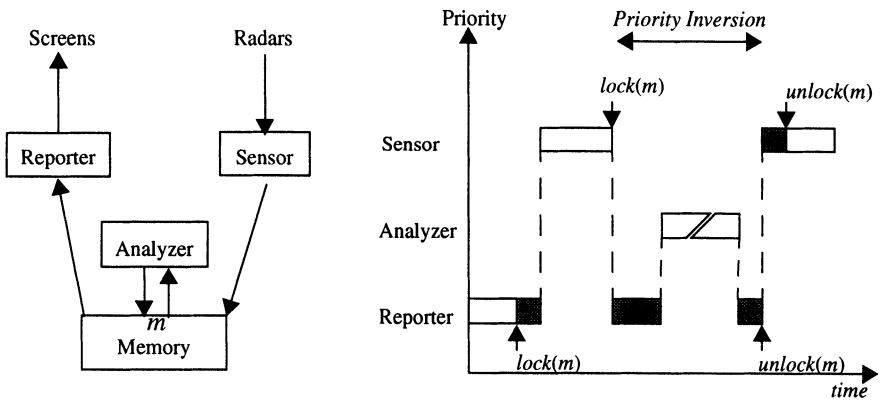


**Figure 8** Unbounded priority inversion

In the original model mutual exclusion is achieved using an additional process that controls the mutual exclusion variable. This process receives requests for mutual exclusion, decides among competing requests and grants mutex to one process at a time. A high overhead is imposed in this model. In addition to the mutual exclusion variable we need one request variable for each process and a new process to control the variable. In the new model we only use the mutual exclusion variable, and

accesses to mutual exclusion are modeled similarly to figure 6, except that we must wait until there is no process in mutex (signalled by `M == 0`) until we try to write to `M`. Priorities are modeled by forcing high priority processes to write to the variable (removing the `random` clause). In this case conflicts are avoided by the low priority process.

```
...
while (M != 1) {
  if (random && M == 0) M = 1;
 wait(1);
};
...
```

**Figure 9** Accessing shared variables in the priority inversion example.

This example has been described in (Campos, 1996b). It has originally been implemented without shared variables. We have been able to reproduce the results previously obtained, but with a significantly smaller model. The old model has 20 boolean variables more than the new one (a savings of about 30%), and verification has been performed up to 10 times faster using up to 20 times less memory.

## 8 CONCLUSIONS

This work describes a new technique for keeping track of control flow information in synchronous symbolic verifiers. It has been used to implement shared variables in the Verus verification system. We have used the new feature to verify a mutual exclusion algorithm and to determine the existence of priority inversion in a real-time system. Using shared variables we have been able to perform the verification up to 10 times faster and using up to 20 times less memory than the same verification without shared variables. These gains demonstrate the efficacy of the new implementation and the importance of efficient synchronization primitives in symbolic verifiers. They show that the new method can help to make possible the verification of even larger systems than previously manageable.

### *Acknowledgment*

The author would like to thank Edmund Clarke for the support and many helpful discussions during the time when part of this work was done at CMU.

## 9 REFERENCES

Berry, G. and Gonthier, G. (1992) The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, **19**.

Brayton, R. et. al. (1996) VIS: a system for verification and synthesis. *8th Conference on Computer-Aided Verification, LNCS,* **1102**. Springer-Verlag.

Campos, S. (1996a) *A quantitative approach to the formal verification of real-time systems.* Ph.D. thesis, SCS, Carnegie Mellon University.

Campos, S. et. al. (1996b) Analysis of real-time systems using symbolic techniques. *Trends in Software — Formal Methods for Real-Time Computing* (ed. C. Heitmeyer and D. Mandrioli). John Wiley & Sons Ltd.

Campos, S. et. al. (1995a) Timing analysis of industrial real-time systems. *Workshop on Industrial-strength Formal specification Techniques.*

Campos, S. et. al. (1995b) Verifying the performance of the PCI local bus using symbolic techniques. *ICCD.*

Campos, S. et. al. (1995c) Verus: a tool for quantitative analysis of finite-state real-time systems. *Workshop on Languages, Compilers and Tools for Real-Time Systems.*

Campos, S. et. al. (1994) Computing quantitative characteristics of finite-state real-time systems. In *IEEE Real-Time Systems Symposium*, 1994.

Clarke, E. et. al. (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, **8**(2):244-263.

Deharbe D. *CV, a VHDL model checker.* `http://www.cs.cmu.edu/~modelcheck/cv/project.html`

Emerson, E. et. al. (1990) Quantitative temporal reasoning. *Conference on Computer Aided Verification.* Springer-Verlag.

Holzmann G. and Peled D. (1996) The state of Spin. *8th Conference on Computer-Aided Verification, LNCS* **1102**. Springer-Verlag.

Jahanian, F. and Stuart, D. (1988) A method for verifying properties of modechart specifications. *IEEE Real-Time Systems Symposium.*

Lamport, L. (1987) A fast mutual exclusion algorithm. *ACM TOCS.* **5**(11).

McMillan, K. (1992) *Symbolic model checking — an approach to the state explosion problem.* Ph.D. thesis, SCS, Carnegie Mellon University.

Rajkumar, R. (1989) *Task synchronization in real-time systems.* Ph.D. thesis, ECE, Carnegie Mellon University.

Rashid, R. et. al. (1989) Mach: a foundation for open systems (operating systems). *Workshop on Workstation Operating Systems.*

Wilner, D. (1997) Keynote Speech. *IEEE Real-Time Systems Symposium.*

## 10 BIOGRAPHY

Sérgio Vale Campos has received the bachelors degree in computer science from Universidade Federal de Minas Gerais (Belo Horizonte — Brasil) in 1986, the masters degree in computer science from the same university in 1990 and the Ph.D. degree in Computer Science from Carnegie Mellon University in 1996. He has spent one year as a post-doctoral researcher at Carnegie Mellon and is now an associate professor at the Universidade Federal de Minas Gerais. His interests include the design, analysis and verification of real-time, parallel and distributed systems.