The Verus Tool: A Quantitative Approach to the Formal Verification of Real-Time Systems[1]

Sérgio Campos, Edmund Clarke and Marius Minea
Carnegie Mellon University
campos@cs.cmu.edu, emc@cs.cmu.edu and marius@cs.cmu.edu

1 Introduction

The task of checking if a computer system satisfies its timing specifications is extremely important. These systems are often used in critical applications where failure to meet a deadline can have serious or even fatal consequences. This work describes *Verus*, an efficient tool for performing this verification task. Using our tool, the system being verified is specified in the Verus language and then compiled into a state-transition graph. A symbolic model checker allows the verification of untimed properties expressed in CTL [8]. Time bounded properties can be verified using RTCTL model checking [7]. Moreover, algorithms derived from symbolic model checking are used to compute *quantitative* information about the model [1]. The information produced allows the user to check the temporal correctness of the model: schedulability of the tasks of the system can be determined by computing their response time; reaction times to events and several other parameters of the system can also be analyzed by this method. 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. The same algorithms can then be used to analyze the performance of the modified design. The evaluation of how the optimizations affect the design can be done *before* the actual implementation, significantly reducing development costs. Another advantage of our approach is that the Verus language has been especially designed to allow a straightforward description of the temporal characteristics of programs. This makes modeling real-time systems in Verus a simpler task.

Verus uses a discrete notion of time. The model of a Verus program is a finite state-transition graph and each transition in the graph corresponds to one time unit. The simplicity of this representation makes it amenable to a symbolic implementation using binary decision diagrams. This representation is very efficient, as attested by the real-time systems verified. One example has 15 concurrent processes and counterexamples that have thousands of states have been produced in seconds. Perhaps more indicative of the usefulness of the method than the size of the counterexamples are the types of systems verified. We have applied this method to the verification of several real systems, such as an aircraft controller [4], a robotics controller [5] and a distributed heterogeneous real-time system [3]. In all cases, the examples verified are either actual existing systems or use components and protocols employed in current industrial products.

2 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. Language constructs have been kept simple in order to make

---

1. 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 compilation into a state-transition graph as efficient as possible. Simple constructs allow the precise expression of the desired features, since complex constructs sometimes force unnecessary details into the specification. Smaller representations can then be generated, which is critical to the efficiency of the verification and permits larger examples to be handled. Details about the Verus language can be found in [1].

Overview

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. No synchronization is needed if the time delays of producer and consumer are defined properly. 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.

```
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

In Verus time passes only on `wait` statements, lines 4, 5 and 6 execute in time zero. This feature allows a more accurate control of time, and eliminates the possibility of implicit delays influencing verification results. It also generates smaller models, since contiguous statements are collapsed into one transition.

The `main` function (not shown for brevity, as well as the consumer code) completes the program by instantiating all processes with the `process` keyword. An implicit instantiation of the `main` module is assumed, where `main` executes as another module. 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*, which introduces nondeterministic transitions and effectively models the desired behavior. This technique is described in detail in [1].

Other Features

Verus has many other features not shown in this program. For example, nondeterminism is implemented in using the `select` statement. To illustrate how `select` works, let's assume that the `producer` is not required to actually produce an item after 3 time units, but may instead leave the value of `p` unchanged. This can be modelled in Verus by changing line 9 to `p = select{p, p+1};`

The timing characteristics of the system can be easily modeled using the periodic, deadline and exception handling statements. For example, the code below specifies that `S1` must execute periodically, once every 100 time units. Also, it must finish execution in less than 100 units, otherwise an exception will be raised:

```
periodic(0, 100, 100) { S1; };
```

The first parameter of `periodic` is the *start_time*, which specifies how many time units the periodic code will idle *before* starting its execution for the first time. In this example it will start immediately. The second parameter is the *period*. In this case the statements following `periodic` will execute once every 100 time units. The third parameter defines a *deadline*. It states that the execution must finish in less than 100 time units or an exception will be raised. Execution may take longer than the sum of the waits because of synchronization with other processes. The `deadline` statement is similar, but it does not specify a period. Exception handling as well as the periodic and deadline statements are explained in detail in [1].

3 The Verification Algorithms

CTL and RTCTL Model Checking

Verus allows the verification of untimed properties expressed as CTL formulas [8] as well as of timed properties expressed as RTCTL formulas [7]. 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 [7].

Many important properties of real-time systems can be verified using both CTL and RTCTL model checking. For example, we have used it to show the existence of priority inversion in a real-time system [2]. In this example, we have modeled a simple real-time system in which processes communicate in a non-regular pattern. The main objective is to determine which problems can arise from this communication and how to avoid them. The bounded until operator allows us to determine the existence of priority inversion, and to check that the solution implemented, priority inheritance, avoids the problem.

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.

Selective Quantitative Analysis and Interval Model Checking

The algorithms described above compute the minimum and maximum time delays along *every* possible execution sequence of a real-time system. In many situations, however, we may be interested in computing time delays that relate only to a subset of the execution sequences that satisfy a given property. For example, in the aircraft controller example [4] the time between requesting the activation of the weapons and their actual firing time is computed. The maximum time in that example is infinity. The weapons may never fire because the firing sequence can be aborted. It may be the case, however, that the designers want to compute the maximum response time of the weapon subsystem *provided that no abort occurs*.

We propose a method for specifying and verifying properties such as these. The user can restrict the set of paths that will be considered by specifying a property that must be satisfied in all paths traversed. This property is expressed using *linear-time temporal logic* (LTL) [6]. Special model checking techniques [6] are then used to ensure that only paths that satisfy the formula are considered by the algorithms. Selective quantitative analysis has been used to analyze the distributed heterogeneous system described in [3].

4 Conclusions

This work describes a new tool to be used in the formal verification of real-time systems, Verus. In Verus a real-time designer specifies the system to be verified in a C-like language, and uses temporal logic model checking and quantitative timing analysis to verify its correctness. The information produced by our algorithms can help in verifying a real-time system in many ways. It not only assists in determining its correctness, but also provides an insight into the behavior of the system. This allows for a better understanding of system behavior and in some cases it can even suggest optimizations to the design.

We have used this tool to analyze several real-time systems of industrial complexity, such as an aircraft controller, a robotics controller and a distributed heterogeneous system. In all cases we have been able to determine the temporal correctness of the system. In several instances the results produced by our algorithms suggested modifications to the design that resulted in more efficient systems. From our experience with Verus we believe that it can be very useful in designing better and more efficient real-time systems.

5 References

[1] S. V. Campos. *A quantitative approach to the formal verification of real-time systems.* Ph.D. thesis, SCS, Carnegie Mellon University, 1996.

[2] S. V. Campos. The priority inversion problem and real-time symbolic model checking. Technical Report CMU-CS-93-125, Carnegie Mellon University, 1993.

[3] S. V. Campos and O. Grumberg. Selective quantitative analysis and interval model checking: verifying different facets of a system. In: *Computer Aided Verification,* 1996.

[4] S. V. Campos, E. M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *IEEE Real-Time Systems Symposium*, 1994.

[5] S. V. Campos, E. M. Clarke, W. Marrero and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. In: *Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.

[6] E. Clarke, O. Grumberg, and H. Hamaguchi. Another look at LTL model checking. In: *Sixth Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 818, pages 415-427. Springer-Verlag, 1994.

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

[8] K. L. McMillan. *Symbolic model checking - an approach to the state explosion problem*. Ph.D. thesis, SCS, Carnegie Mellon University, 1992.