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. A Verus program is modelled by a finite state-transition graph where each transition corresponds to one time unit. The simplicity of this representation makes it amenable to a symbolic implementation using BDDs. The tool is very efficient, as attested by the systems verified. One example has 15 concurrent processes and counterexamples that have thousands of states have been produced in seconds. Perhaps even more indicative of the usefulness of the method 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]. All examples verified are either actual systems or use components and protocols employed in current industrial products.

1. This research was sponsored in part by the National Science Foundation under grant no. CCR-9217549, by the Semiconductor Research Corporation under contract 95-DJ-294 and by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under title "Research on Parallel Computing", ARPA order no. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

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. 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 the compilation into a state-transition graph as efficient as possible. Smaller representations can then be generated, which is critical for 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` is shown below. Variable `p` is an index to the data buffer. After initializing index `p` and variable `produce`, the `producer` 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 production by asserting `produce`. In line 9 the index `p` is updated. Line 10 ensures 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. Process instantiation in Verus follows a synchronous model, all processes execute in lockstep. Asynchronous behavior can be modeled by using *stuttering*, which introduces nondeterministic transitions. This technique is described in [1].

Other Features

Verus has many other features not shown in this program. For example, nondeterminism is implemented 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 and deadline statements. For example, the code below specifies that `S1` must execute 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 code will idle before starting its execution for the first time. The second parameter is the *period*, that is, how often the code will execute. The third parameter defines a *deadline*. If execution does not finish before the deadline an exception will be raised. Execution may take longer than the sum of the waits because of synchronization. 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 [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]. RTCTL extends CTL 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 RTCTL 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 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 and the verifier automatically determines if it is satisfied or not. However, these techniques do not provide any information about how much a system deviates from its expected performance, although this information can be very useful in fine-tuning the system behavior.

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.

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 execution sequences that satisfy a given property. We propose a method for specifying and verifying properties such as these. The user specifies 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 used to ensure that only paths satisfying the formula are considered by the algorithms.

4 Conclusions

This work describes Verus, a new tool to be used in the formal verification of real-time systems. In Verus the 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 tool can help in verifying a real-time system in many ways. It not only assists in determining its correctness, but also provides insight into the behavior of the system. This allows for a better understanding of the system and in some cases it even suggests 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 suggested modifications to the design that resulted in more efficient 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. *Computer Aided Verification,* 1996.
4. S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. *IEEE Real-Time Systems Symposium,* 1994.
5. S. Campos, E. Clarke, W. Marrero and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. *Languages, Compilers and Tools for Real-Time Systems,* 1995
6. E. Clarke, O. Grumberg, and H. Hamaguchi. Another look at LTL model checking. *Computer-Aided Verification,* LNCS vol. 818. Springer-Verlag, 1994.
7. E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. *Computer-Aided Verification,* 1990.
8. K. L. McMillan. *Symbolic model checking - an approach to the state explosion problem.* Ph.D. thesis, SCS, Carnegie Mellon University, 1992.