Analysis and Verification of Real-Time Systems using Quantitative Symbolic Algorithms

Sérgio Vale Aguiar Campos[1]
scampos@dcc.ufmg.br
Universidade Federal de Minas Gerais, Belo Horizonte, Brasil

Edmund Clarke[2]
emc@cs.cmu.edu
Carnegie Mellon University, Pittsburgh, USA

Abstract: 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 paper presents an efficient method for performing this verification task. In the proposed method a real-time system is modeled by a state-transition graph represented by binary decision diagrams. Efficient symbolic algorithms exhaustively explore the state space to determine whether the system satisfies a given specification. In addition, our approach computes quantitative timing information such as minimum and maximum time delays between given events. These results provide insight into the behavior of the system and assist in the determination of its temporal correctness. The technique evaluates how well the system works or how seriously it fails, as opposed to only whether it works or not. Based on these techniques a verification tool called *Verus* has been constructed. It has been used in the verification of several industrial real-time systems such as the robotics system described below. this demonstrates that the method proposed is efficient enough to be used in real-world designs. The examples verified show how the information produced can assist in designing more efficient and reliable real-time systems.

1 Introduction

In many computer applications predictable response times are essential for the correctness of the system. Such systems are called *real-time systems*. They occur in many critical applications in which a late (or sometimes early) response can have severe consequences. Examples of such applications include controllers for aircraft, industrial machinery and robots. Due to the nature of such applications, errors in real-time systems can be extremely dangerous, even fatal. Guaranteeing the correctness of a complex real-time system is an important and non-trivial task.

Several other factors make the verification of real-time and non real-time systems particularly difficult. The architecture of computer applications is becoming more and more complex each day. The more complex a system, the higher the possibility of errors being introduced in its design. Moreover, performance is also becoming a more important factor in the success of new applications. Competition has made design optimizations essential. Consequently, new products have to fully utilize the available resources. A slow component can compromise the performance of the whole system, and consequently its acceptance by the market. Although not traditionally associated with real-time systems, verifying timing properties of these

1. This research was sponsored in part by CNPq — Conselho Nacional de Pesquisa e Desenvolvimento under the project “Métodos Formais para a Verificação de Sistemas Computacionais de Complexidade Industrial”.
2. 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 the title “Research on Parallel Computing”, ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

applications is also critical. The task of verifying that industrial real-time applications satisfy their timing specifications is more critical and difficult today than ever.

This work describes *Verus*, an efficient tool for performing this verification task. A system being verified in this tool is specified in the Verus language. This language has been especially designed to simplify the description of time related characteristics. The description of the system is then compiled into a state-transition graph which can be analyzed using several methods. A symbolic model checker allows the verification of untimed properties expressed in CTL [15]. Time bounded properties can be verified using Real-Time CTL model checking [10]. In addition, 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.

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. An important consequence of this model is that the algorithms count the number of computation steps between events, or the number of occurrences of events in an interval. Because of this the discrete time model finds application in synchronous systems in general, such as computer circuits and protocols. Real-time systems usually do not execute in lock-step since contention for resources such as processor time introduces nondeterminism in its execution. Because of this real-time systems would not seem to fit naturally in our model. However, they are subject to tight timing constraints, which are difficult to satisfy in an asynchronous design. For this reason real-time system developers often significantly reduce asynchronism in their designs to ensure predictability. In fact, most real-time systems we have analyzed present more synchronous features than traditional circuits, and have been successfully verified using Verus [5,6,7,8].

Related Work

There are several other approaches to the verification of timed systems. For example, dense time is modeled by [1,2,15,19]. Those methods provide a very accurate notion of passage of time. However, the state space of dense time models is infinite, and these verification tools rely on the construction of a finite quotient structure called region graph. This construction is extremely expensive, limiting the size of problems that can be handled.

Discrete time is used by other tools such as [11,13,22]. The tool described in [22] also uses symbolic algorithms using BDDs. These tools, however, do not allow the quantitative analysis of systems as the proposed method. In [10] quantitative analysis is implemented, but with a more limited scope.

Analytical methods for analyzing real-time systems also exist, such as the rate-monotonic scheduling theory [16,17,21]. In this method a real-time system is characterized by a set of periodic tasks, each having a period and an execution time. Assumptions about system behavior are made (such as no task preempts itself), and if these assumptions are satisfied, simple formulas determine the schedulability of the system. A strong limitation of this approach is that only systems satisfying several restrictions can be verified such as very limited synchronization models and severe restrictions on analyzing distributed systems. The rate-monotonic theory algorithms have much simpler complexity than the other verification methods discussed, but they also generate more restricted information.

Example: A Robotics Controller

The simplicity of the representation used by Verus makes it amenable to a symbolic implementation using binary decision diagrams. This representation is very efficient, as attested by the real-time systems verified. Several complex systems of industrial origin have been verified such as an aircraft controller [6], the PCI local bus [8] and a distributed heterogeneous real-time system [5]. In all cases, the examples verified are derived from existing systems or use components and protocols employed in current industrial products.

This paper describes in detail one of these systems, the controller for a robot used in nuclear reactors to measure the shapes of pipes by moving around them with a distance sensor [7,14]. In this example we have been able to analyze the behavior of the robot from several perspectives. Our analysis has shown that it would meet its deadlines, but that it was inefficient. We have discussed how to optimize the design and then analyzed the performance of the modified design.

The remainder of the paper is organized as follows. Section 2 presents the Verus language, used to describe the system being verified. In section 3 we briefly present the algorithms used for verification. An example of the verification of a real-time system is discussed in section 4, and conclusions are presented in section 6.

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 the compilation into a state-transition graph as efficient as possible. Simple constructs allow the precise expression of the desired features, whereas 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 on the Verus language can be found in [1].

Overview

A fragment of a simple real-time program is used to illustrate the basic features 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 (addition in Verus is defined modulo its maximum value). 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

`Wait` Statements

In Verus time passes only on `wait` statements, therefore lines 4, 5 and 6 execute in time zero. This feature allows an 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. Notice that this feature affects the behavior of the program significantly. For example, a block of code not containing the `wait` statement executes atomically.

Nondeterminism

To illustrate another characteristic of Verus, 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 modeled in Verus by changing line 9 to:

```
 9        p = select{p, p+1};
```

The `select` statement introduces a nondeterministic choice in the program. The value of `p` after executing `select` can be either `p` or `p+1` (addition in Verus is defined modulo the maximum value for the variable). These choices can characterize the fact that the producer *may* produce an item, but it may also *not* produce it. This way we can model both possibilities without having to specify all the details that are actually needed to decide between these two options. Besides hiding unnecessary details, nondeterminism can be used to verify partial specifications. Whenever the value of a variable hasn't been determined by the design, nondeterministic constructs can specify all possible values for the variable. This approximates the behavior of the actual system by exploring all possibilities. As the design process evolves, the values can be restricted until the correct behavior is determined. Nondeterminism encourages the use of automated verification in earlier phases of the design. Components of the system can be verified before all modules have been specified and errors can be uncovered before propagating to components added later.

```
14    consumer(p, c)
15    {
16      boolean consume;
17
18      c = 0;
19      consume = false;
20      while (!stop) {
21        wait(1);
22        if (p != c) {
23          consume = true;
24          c = c + 1;
25          wait(1);
26          consume = false;
27        };
28      };
}
```

Figure 2. Consumer code

The `consumer` process is very similar to the `producer`. The basic differences are that it waits for less time before consuming, and that it only consumes if `p` and `c` have different values (`p == c` signals an empty buffer). Notice that the `producer` does not check if the buffer is full before inserting another item. The time delays of both processes guarantee that an overflow will never occur.

The `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* [4]. 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 example given specify the computation of 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`.

```
main()
{
  int p, c;

  process prod producer(p),
          cons consumer(p, c);

  spec MIN[prod.produce, cons.consume]
       MAX[prod.produce, cons.consume]
       AG(prod.produce -> AF cons.consume)
}
```

Figure 3. Producer/consumer main function

Timing Constructs

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 delays in the wait statements 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: 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. For example, by choosing as starting states those in which a process requests execution, and as final states those in which the process completes execution we can compute the response time for that process. If we specify as third condition for the same intervals the execution of lower priority processes we can compute the amount of priority inversion time that can affect the process. These algorithms are discussed in detail in [3]. We briefly present the minimum and maximum delay algorithms below.

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. As discussed, 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.

Minimum Delay Algorithm

The algorithm takes two sets of states as input, *start* and *final*. It returns the length of (i.e. number of edges in) a shortest path from a state in *start* to a state in *final*. If no such path exists, the algorithm returns infinity. In the algorithm, the function $T(S)$ gives the set of states that are successors of some state in S. In other words, $T(S) = \{s' \mid N(s, s') \text{ holds for some } s \in S\}$. In addition, the variables R and R' represent sets of states in the algorithm.

```
proc min (start, final)
$i = 0;$
$R = start;$
$R' = T(R) \cup R;$
while $((R' \neq R) \wedge (R \cap final) = \emptyset)$ do
        $i = i + 1;$
        $R = R';$
        $R' = T(R') \cup R';$
if $(R \cap final \neq \emptyset)$
        then return $i$;
        else return $\infty$;
```

Figure 4. Minimum Delay Algorithm

The first algorithm is relatively straightforward. Intuitively, the loop in the algorithm computes the set of states that are reachable from *start*. If at any point, we encounter a state satisfying *final*, we return the number of steps taken to reach the state.

Maximum Delay Algorithm

This algorithm also takes *start* and *final* as input. It returns the length of a longest path from a state in *start* to a state in *final*. If there exists an infinite path beginning in a state in *start* that never reaches a state in *final*, the algorithm returns infinity. The function $T^{-1}(S')$ gives the set of states that are predecessors of some state in S' (i.e. $T^{-1}(S') = \{s \mid N(s, s') \text{ holds for some } s' \in S'\}$). R and R' will once more be sets of states. Finally, we denote by *not_final* the set of all states that are not in *final*.

```
proc max (start, final)
i = 0;
R = true;
R′ = not_final;
while ((R′ ≠ R) ∧ (R′ ∩ start ≠ ∅)) do
    i = i + 1;
    R = R′;
    R′ = T^{-1}(R′) ∩ not_final;
if (R = R′)
    then return ∞;
    else return i;
```

Figure 5. Maximum Delay Algorithm

The upper bound algorithm is more subtle than the previous algorithm. In particular, we must return infinity if there exists a path beginning in *start* that remains within *not_final*. A backward search from the states in *not_final* is more convenient for this purpose than a forward search. At each iteration it finds the set of states which are the beginning of intervals with *i* states, none satisfying *final*. Initially, *i* is 0, and the frontier is *not_final*. At the *i*th iteration the current frontier is the set of states that are the beginning of paths with *i* states completely in *not_final*. We then compute the set of predecessors (in *not_final*) of the current frontier. Those states are the beginning of paths with *i*+1 states completely in *not_final.* The algorithm stops at the *j*th iteration when it detects that there are no states in *start* that are the beginning of paths with *j* states completely in *not_final*. But there is at least one path starting in *start* with *j*-1 states in *not_final*, that is, this is a maximal path starting in *start* completely in *not_final*. We also assume that the transition relation is total, that is, the *j*-1th state has at least one successor, which must be in *final*.

4 A Robotics System

One application of real-time systems that is becoming increasingly common is in robotics. Guaranteeing that tasks are executed within their expected deadline is critical for the integrity of a robot and for its correct operation. The computation of quantitative properties can assist in validating such systems. The example discussed in this section is derived from the one in [14]. It describes a real robot used in nuclear reactors to measure the shapes of pipes by moving around them with a distance sensor. The robot architecture has three subsystems, *motor, measurement* and *command*. The motor subsystem controls the robot movements and position. The function of the measurement subsystem is to activate and control the distance sensors. Finally, the command subsystem receives commands from the communication link and sends them to the appropriate tasks.

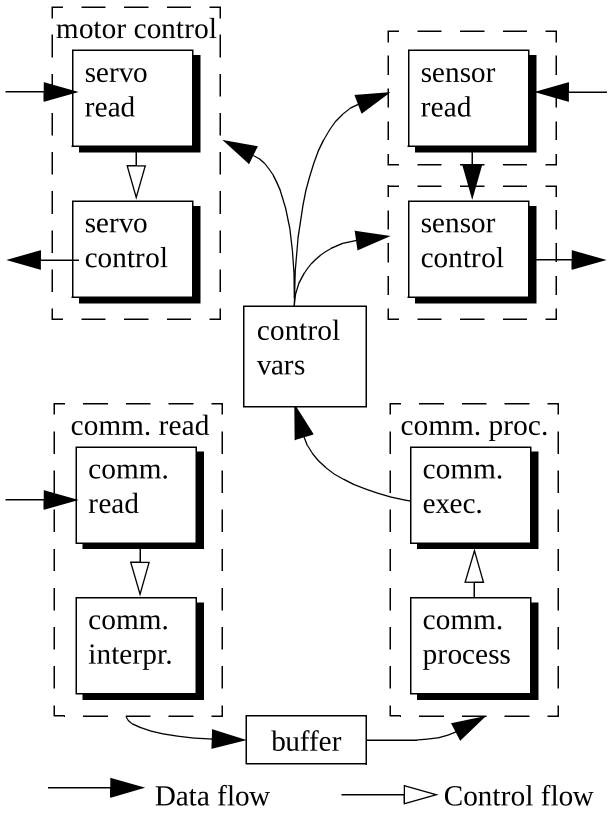

Figure 6. Robot architecture

Each subsystem consists of a set of tasks. The motor subsystem contains one task, *motor_control*. Its function is to receive data from sensors in the servo motors, and actuate them. The task consists of two subtasks, *servo_read* and *servo_control*. The first one is an interrupt routine that reads data directly from the physical devices. The second one processes these data and outputs control signals to the motors at a lower priority. The measurement subsystem has two tasks, *sensor_read* and *sensor_control*. The first task reads data from the distance sensors and preprocesses it. This information is then sent to *sensor_control*, which processes it further and outputs the results to a remote system to be analyzed. Finally, the command subsystem also has two tasks. The *command_read* task receives commands from the communication link and interprets them. It consists of two subtasks: an interrupt routine, followed by a second subtask that has a lower priority. The final task of this subsystem is *command_process*. Its first subtask receives the command interpreted by *command_read*, and the second one then executes the command. Control variables updated by this subtask are used to communicate commands to all other subsystems.

All tasks are periodic, and their timing requirements reflect the characteristics of the environment in which the robot works and the robot's expected response time. These requirements are summarized in the table below. Each task is presented as a sequence of components, each with a different execution time and priority. A component may correspond to a subtask, or subtasks may be split in more than one component due to synchronization. For example, the first components of both *motor_control* and *command_read* correspond to their interrupt routines and execute at high priorities. Synchronization accounts for the other components. For example, the last component of *command_process* updates control variables that will be used by other tasks. Interference from other tasks is avoided by accessing those variables at a high priority level. The other components have been created to reflect the synchronization pattern between processes sharing data (in this case *sensor_read* and *sensor_control*), and between *command_read* and *command_process*. Priority inheritance protocols have been used to avoid priority inversion [20]. These protocols change the priority of the tasks at synchronization points, thus dividing the tasks into components.

Task	**Period**	**Exec. Times**			**Deadline**	**Priorities**		
		C_1	C_2	C_3		P_1	P_2	P_3
Motor control	40	1	5	-	40	10	7	-
Sensor read	100	10	5	5	100	4	8	4
Sensor control	50	8	12	-	50	5	8	-
Command read	200	10	20	3	200	9	2	3
Command process	400	2	12	10	400	3	1	6

Figure 7. Timing requirements for the controller

Specifying the Robotics System in Verus

The periodic statement. A Verus program has been written describing this system. The five processes have been implemented using the basic structure shown below for the `motor_control` process. These processes and their corresponding scheduler have been instantiated in the final model.

```
motor_control() {
  boolean start, end;

  start = false; end = false;
  periodic(0, 40, 40) {
    start = true;
    priority(10) {
      wait(1);
      start = false;
    };
    priority(7) {
      wait(5);
    };
    end = true;
    wait(1);
    end = false;
  };
}
```

Figure 8. The basic structure for the Verus program

However, because of efficiency in the verification some modifications have been made to this structure. The main modification arises from realizing that each process needs one counter for each `periodic` statement [3]. It is more efficient to group all periodic statements using only one counter. This can be done by declaring only one counter and raising a “periodic” flag whenever this counter reaches values that are multiples of the desired period. For example, in this system the periods of all processes are 40, 50, 100, 200 and 400. The code below shows how to implement periodic statements for these periods using one counter (the current version of Verus does not implement the `#define` directive as well as other minor sintactic constructs shown. They are presented here to simplify the presentation):

```
int t;

#define timeout40 ((t == 0) || (t == 40) || (t == 80) ||
                   (t == 120) || (t == 160))
#define timeout50 ((t == 0) || (t == 50) || (t == 100) ||
                   (t == 150))
...
count() {
  t = 0;
  while (true) {
    t = t + 1 % 400;
  };
}
```

Finally, the code shown on the left side below is then replaced by the code on the right:

```
periodic(0, 40, 40) {        while (true) {
  S1;                          while (!timeout40) wait(1);
};                             S1;
                             }
```

Figure 9. An efficient grouping of `periodic` statements

The scheduler. The scheduler is implemented with the aid of auxiliary variables. A variable granted is used to determine which process will execute next. Each process uses a variable `req`*i* to signal to the scheduler that it is requesting execution and at which priority level. Whenever requesting execution, process `p`*i* sets variable `req`*i* to its priority level. When it finishes executing it resets the variable. During execution it must wait until the variable `granted` has its index before proceeding (this level of detail in the code can be hidden with the use of preprocessors):

```
/* Beginning of execution */
req1 = 10;
while (granted != 1) wait(1);
wait(1);  /* execute for one time unit */
req1 = 7;
while (granted != 1) wait(1);
wait(1);
...
while (granted != 1) wait(1);
wait(1);
/* End of execution */
req1 = 0;
wait(1);
```

The scheduler reads all `req` variables and decides which process will execute next. It then sets the variable `granted` to the index of the process with highest priority, as shown in the simplified scheduler below:

```
scheduler(req1, req2, req3, granted)
{
  while (true) {
    if (req1 >= req2) {
      if (req1 >= req3)  granted = 1; else
```

```
                          granted = 3;
        } else {
          if (req2 >= req3)  granted = 2; else
                          granted = 3;
        };
        wait(1);
      };
```

The final model has the code for the five processes, the counting process and the scheduler. The specifications complete the code:

```
      spec
        MIN(p1.start, p1.end);
        MAX(p1.start, p1.end);
        MIN(p2.start, p2.end);
        MAX(p2.start, p2.end);
        MIN(p3.start, p3.end);
        MAX(p3.start, p3.end);
        MIN(p4.start, p4.end);
        MAX(p4.start, p4.end);
        MIN(p5.start, p5.end);
        MAX(p5.start, p5.end);
```

The model can be executed with the command below, where `robot.ord` is the BDD variable ordering file used for this example:

```
>verus -i robot.ord robot.ver
```

A summary of the results produced is (the running times refer to a Pentium 200 with 64 megs of memory):

```
...

Time to construct the model: User :    19.37 s  System   :     0.12 s

Result is      6 : MIN(p1.start, p1.end);
Result is     16 : MAX(p1.start, p1.end);
Result is     45 : MIN(p2.start, p2.end);
Result is     95 : MAX(p2.start, p2.end);
Result is     20 : MIN(p3.start, p3.end);
Result is     49 : MAX(p3.start, p3.end);
Result is    181 : MIN(p4.start, p4.end);
Result is    190 : MAX(p4.start, p4.end);
Result is    219 : MIN(p5.start, p5.end);
Result is    223 : MAX(p5.start, p5.end);


Execution information:

Time              - User               :    43.28 s  System   :     0.14 s
BDD nodes used    - Transition relation:      35690  Total    :     123463
Bytes allocated   -                         3139320
Boolean variables -                             115
States            - Total              : 2.36118e+22  Reachable:        400
```

The Analysis of the Robotics System

Computing response times for all processes generated the results in the table below. This table shows that the task set is schedulable. Moreover, the maximum execution times of many tasks are close to their deadlines. This indicates a high load on the system; it is unlikely that adding more tasks to the task set would produce a schedulable system. This information allows the designer to optimize the system.

Task	**Deadline**	**Exec. times**	
		min	max
Motor control	40	6	16
Sensor read	100	45	95
Sensor control	50	20	49
Command read	200	181	190
Command process	400	219	223

Figure 10. Schedulability analysis for original system

Using the results computed by our algorithms, we have been able to suggest changes to the design and to analyze the effects of such changes. In the original design *sensor_read* generates data that are used by *sensor_control*. However, the two tasks execute independently of one another. In some cases *sensor_control* might execute even if data are not yet available. In this case, *sensor_control* uses data generated by the previous instantiation of *sensor_read*, which may be obsolete. We have changed the system to avoid this problem and have analyzed the resulting design. The modification consists of making the termination of *sensor_read* trigger the execution of *sensor_control*. Care must be taken, however, because the processes involved have different periods; *sensor_read* executes every 100 ms, while *sensor_control* executes every 50 ms. We change the system so that *sensor_read* signals the execution of *sensor_control* every 100 ms, but *sensor_control* also executes independently 50 ms after *sensor_read* runs. In this case one instantiation of *sensor_control* is synchronized with *sensor_read* while the other is independent. The schedulability analysis of the modified example is given in the following table:

Task	**Deadline**	**Exec. times**	
		min	max
Motor control	40	6	16
Sensor read	100	20	36
Sensor control	50	21	121
Command read	200	91	91
Command process	400	96	296

Figure 11. Schedulability analysis for modified system

The new design is not schedulable, since *sensor_control* can take up to 121 ms to execute. We can use the same quantitative algorithms to find out more about the behavior of the system and to correct the problem. A more detailed analysis reveals that the two instantiations of *sensor_control* have very distinct behaviors. Whenever executing periodically (and independent of *sensor_read*), *sensor_control* takes between 21 and 121 ms to finish. However, whenever executing after *sensor_read*, it takes exactly 26 ms to execute in the modified model. This shows that the periodic execution of *sensor_control* is the bottleneck of the system. One solution to the problem is simply removing the periodic instantiation of *sensor_control*. This solution was easily implemented, and the schedulability analysis is presented in table:

Task	**Deadline**	**Exec. times**	
		min	max
Motor control	40	6	16
Sensor read	100	20	36
Sensor control	50	26	26
Command read	200	91	91
Command process	400	70	270

Figure 12. Schedulability analysis for final system

The system is again schedulable, but now *sensor_control* executes only once every 100 ms. Is this a satisfactory solution? Again, we can use the same algorithms to analyze the modified design. By computing the time between the end of the execution of *sensor_read* and the beginning of *sensor_control* we can verify if data produced by the first task is being consumed timely by the second one. In the modified model this time is between 1 and 7 ms, meaning that data produced by *sensor_read* are promptly consumed by *sensor_control*. Therefore we can conclude that in spite of changing the periodicity of *sensor_control* we are still maintaining correctness. The condition counting algorithms have also been useful in analyzing the performance of this model. We have been able to verify how the old periodicity of *sensor_control* relates to the new model. We can consider all execution paths from the time *sensor_read* starts until *sensor_control* finishes as the active period for the measurement subsystem. During such a period, how many times can the 50 ms time-out occur? In other words, how many times would *sensor_control* be activated using the original periodicity during an active period? The result is from 1 to 3 times. We conclude that the modified system satisfies the original timing constraints, even though it has a lighter load.

5 Other Verification Algorithms in Verus

CTL and RTCTL Model Checking

Verus allows the verification of untimed properties expressed as CTL formulas [15] and of timed properties expressed as RTCTL formulas [10]. 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 introducing time bounds on all CTL operators [10]. For example, the formula below specifies that requests will always be acknowledged in 10 time units or less:

```
AG (req -> AF 0..10 ack)
```

Many important properties of real-time systems can be verified using both CTL and RTCTL model checking. For example, we have used this method to show the existence of priority inversion in a real-time system [4,20]. Priority inversion occurs when a higher priority process is blocked by a lower priority one. It can be caused by unconstrained process communication. 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.

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 [6] 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) [9]. Special model checking techniques [9] 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 [5].

6 Conclusions

In this paper we present a new method for analyzing and verifying real-time systems. The system being analyzed is described in the Verus language, which has been especially designed to simplify the expression of timing characteristics. Symbolic algorithms are then used to compute quantitative timing information about the system such as response times and the number of occurrences of events in given intervals. The results produced by the algorithms can assist in determining the correctness of the system as well as provide important information about system behavior that can help in understanding how the system behaves under different conditions.

We have used this tool to analyze several real-time systems of industrial complexity, such as an aircraft controller [6], the PCI local bus [8] and a distributed heterogeneous system [5]. 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.

7 References

[1] R. Alur, C. Courcourbetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Symposium on Logics in Computer Science*, pp. 414-425, 1990.

[2] R. Alur and D. Dill. Automata for modeling real-time systems. In *Lecture Notes in Computer Science, 17th ICALP*. Springer-Verlag, 1990.

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

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

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

[6] 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.

[7] 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.

[8] S. V. Campos, E. M. Clarke, W. Marrero and M. Minea. Verifying the performance of the PCI local bus using symbolic techniques. In: *ICCD*, 1995.

[9] 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.

[10] P. Clements, C. Heitmeyer, G. Labaw, and A. Rose. MT: a toolset for specifying and analyzing real-time systems. In *IEEE Real-Time Systems Symposium*, 1993.

[11] W. Elseaidy, R. Cleaveland and J. Baugh. Modeling and verifying active structural control systems. In: *Science of Computer Programming*, 29(1-2):99-122, July 1997.

[12] 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.

[13] A. N. Fredette and R. Cleaveland. RTSL: a language for real-time schedulability analysis. In *IEEE Real-Time Systems Symposium*, 1993.

[14] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering*, 20(1), 1994.

[15] T. Henzinger, P. Ho, H. Wong-Toi. HyTech: the next generation. In: *IEEE Real-Time Systems Symposium,* 1995.

[16] J. P. Lehoczky, L. Sha, J. K. Strosnider, and H. Tokuda. Fixed priority scheduling theory for hard real-time systems. In *Foundations of Real-Time Computing - Scheduling and Resource Management*. Kluwer Academic Publishers, 1991.

[17] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.

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

[19] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. In *Lecture Notes in Computer Science, Real-Time: Theory in Practice*. Springer-Verlag, 1992.

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

[21] L. Sha, M. H. Klein, and J. B. Goodenough. Rate monotonic analysis for real-time systems. In *Foundations of Real-Time Computing - Scheduling and Resource Management*. Kluwer Academic Publishers, 1991.

[22] J. Yang, A. K. Mok, and F. Wang. Symbolic model checking for event-driven real-time systems. In *IEEE Real-Time Systems Symposium,* 1993.