Verus: A Tool for Quantitative Analysis of Finite-State Real-Time Systems *

S. Campos E. Clarke W. Marrero M. Minea
School of Computer Science
Carnegie Mellon University

January 19, 1995

Abstract: Symbolic model checking is a successful technique for verifying extremely large finite-state concurrent systems. Models with up to 10^{30} states can often be verified in minutes.In this paper we extend model checking to compute timing constraints for real-time systems automatically. We have designed a new language, called *Verus*, for the description of real-time systems. Such a system is compiled into a state-transition graph and represented symbolically using binary decision diagrams. A general method is given for exploring the state space and computing *quantitative* information about the system. We have developed algorithms that determine exact bounds on the delay between two specified events and on the number of occurrences of an event in a given interval. This technique allows us to determine performance measures such as schedulability, response time, and system load. Our algorithms produce more detailed information than traditional methods such as rate monotonic scheduling analysis. This information leads to a better understanding of the behavior of the system, in addition to verifying if its timing requirements are satisfied. Moreover, the description of the system in the Verus language provides an accurate specification that can reveal ambiguities which could lead to implementation problems. The algorithms presented in this paper have also been incorporated into the SMV model checker and used to verify a robotics control system. The results obtained demonstrate that our method can be successfully applied in the analysis of real-time system designs.

1 Introduction

Symbolic model checking is a technique for specifying and verifying finite-state concurrent systems [1, 8]. It determines automatically if a system satisfies its specifications. Models with up to 10^{30} states can often be verified in minutes. The method has been used successfully to verify a number of real-world applications. For example, it has been used to find errors in the Futurebus+ cache coherence protocol, adopted as a standard by both IEEE and the U.S.Navy [3].

Many real-time systems can be represented by finite-state models. In [2] we have shown how to apply symbolic model checking techniques to analyze finite-state real-time systems. The method presented in that paper differs significantly from other verification methods. It computes quantitative timing information about a model rather than just determining whether it satisfies a given specification or not. Those algorithms provide insight into how well a system works, in addition to determining its temporal correctness.

In this paper, we extend our technique in two ways. Variations of the algorithms previously described are

*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.

explored. The original algorithms compute the exact lower and upper bounds on the amount of time that elapses between two events. Alternatively, we may be interested not only in the length of the time interval between two events, but also in the number of times a third event occurs within such an interval. For example, priority inversion time may be determined by computing the number of time steps executed at lower priority levels from the moment a high priority process requests execution until it has finished. We describe in detail algorithms to compute such information and prove their correctness. The addition of these algorithms increases the power and applicability of our technique.

We also extend our previous work by defining a new language for describing real-time systems, the Verus language. This language has been specifically tailored to simplify the expression of real-time properties and constraints. Verus provides special primitives to express timing aspects such as deadlines, priorities and delays. Nondeterminism is also supported, which partial specifications to be described. Moreover, the syntax of Verus resembles the C language syntax. We believe that the familiarity with a well known language will help non-experts to use the new tool.

Finally, we integrate all these ideas into a tool for the analysis and verification of real-time systems, the Verus tool, currently under development. The real-time application being analyzed is described in the Verus language. This description is then compiled into a labeled state-transition graph (which is used by the verification algorithms) that formally models the behavior of the system. The quantitative algorithms described above can be used to analyze its behavior. Moreover, a CTL symbolic model checker [8] is implemented to augment the power of the tool. This model checker has been extended to handle the RTCTL logic [4], allowing the expression of time bounded properties.

To demonstrate how our tools work, we verify a robotics example derived from [7]. The robot we describe is used in nuclear reactors to measure the shapes of pipes by moving around them with a distance sensor. Its controller consists of a set of periodic processes that control each subsystem of the robot. We model this controller, and use the algorithms described in this paper to gather timing information about it. We determine the schedulability of the system using this data. Moreover, the type of information computed by our algorithms allows us to identify inefficiencies, to suggest optimizations to the design and to analyze the performance of these proposed changes.

Several other methods for analyzing real-time systems exist. The *rate monotonic scheduling theory* (RMS) is one such method [10, 7]. The RMS theory proposes a schedulability test based on total CPU utilization. However, there are a number of limitations on the type of processes that can be analyzed by this method, including restrictions on periodicity and synchronization. Another approach to schedulability analysis uses algorithms for computing the set of reachable states of a finite-state system [5, 6]. No restrictions are imposed on the model but the algorithm only checks if exceptions can occur or not, and other types of properties can only be verified if encoded as exceptions. A model checker for real-time systems is proposed in [**?**] using symbolic techniques. However, in this approach quantitative information is not generated, and the verifier only determines if the model satisfies a given property or not.

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

  periodic(0, 40, 40) {
    start = 1;
    priority(10){
      data_in = dev_ready & !abort;
      wait(1);
    };
    priority(7){
      wait(5);
      data_out = data_in & !abort;
    };
    finish = 1;
  };
}
```

Figure 1: The motor control task

By contrast, our analysis method only requires that the model be finite-state. Moreover, the computation of quantitative characteristics provides other valuable information about the system being modeled. For example, the reaction time to a stimulus can be easily computed. This type of information can give the user insight into the behavior of the system, as opposed to only determining if it satisfies a given property. It can lead to a better understanding of system behavior and can be essential in improving performance.

The remainder of the paper is organized as follows. The next section discusses the robotics example and shows how it can be analyzed using our techniques. In Section 3 the symbolic algorithms for computing the longest and shortest paths between two state sets are presented. Symbolic algorithms for counting the number of states that satisfy a given condition along a path between two sets of states are described in section 4. Section 5 concludes the paper with directions for future work.

2 The Verus Language

Verus is the language we use to specify the real-time systems that will be verified. It is an imperative language with a syntax resembling that of the C language. Special primitives are provided to allow the expression of timing aspects such as deadlines, priorities and time delays. The data types allowed are integer and boolean. Nondeterminism is supported to allow partial specifications to be described.

Verus was designed to allow a straightforward description of a real-time system. In addition, its language constructs have been kept simple. This allows an efficient compilation into a state-transition graph, which is a key factor in generating smaller models. It also puts more expressive power into the hands of the user. TO FIX ???

We will briefly describe the syntax of the Verus language using as an example part of the code for the robot model to be analyzed in this paper.

For example, the `periodic` statement causes the periodic execution of a Verus statement according to the parameters it is given. It has three parameters followed by the code that will be executed periodically. The first parameter 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 40 time units. The last parameter defines a *deadline* for the code. It specifies that the code must finish execution in less than 40 time units or an exception will be raised. Deadlines can also be defined in a similar manner without forcing the process to execute periodically by using the primitive `deadline`.

Passage of time is controlled by the `wait` statement. For example, in line 12 the `motor_control` task will wait for 5 time units before resuming execution. Unlike regular imperative languages, in Verus time passes only on `wait` statements. This feature allows a more accurate control of time, and eliminates the possibility of implicit delays influencing the verification results. It also generates models with less states, since consecutive statements not separated by a wait statement are compiled into a single (state/change of state). Notice that this feature affects the behavior of the program significantly. For example, a block of code not containing the `wait` statement executes atomically.

The `select` statement (not shown in the sample code) is used to introduce nondeterministic choice into a program. If, for example, `motor_control` was allowed not to signal end of execution, we could replace line 15 with `finish = select{0,1};`. In this case, the value of `finish` after executing `select` can be either 0 or 1. These choices indicate that `motor_control` *may* signal termination, but does not have to do so. In 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 in the design, nondeterminism can be used to specify all possible values the variable could take. 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 validated before all modules have been specified. In this way errors can be uncovered before they propagate to components added later in the design.

3 Lower and Upper Bound Algorithms

This section presents algorithms for computing minimum and maximum time delays between specified events. We first describe how a state-transition graph can be used to model the real-time system being verified. Propositional variables label the states in this graph. A state $\overline{v}$ in this model is represented by a vector assigning values to the state variables $v_1, v_2, \ldots, v_n$. The transition relation $N(\overline{v}, \overline{v'})$ evaluates to true when there is a transition in the model from the state $\overline{v}$ to the state $\overline{v'}$, where $\overline{v} = \langle v_1, \ldots, v_n \rangle$ and $\overline{v'} = \langle v'_1, \ldots, v'_n \rangle$. A *path* in the transition graph is defined as a sequence of states $\overline{v_0}, \overline{v_1}, \overline{v_2}, \ldots$ such that $N(\overline{v_i}, \overline{v_{i+1}})$ is true for every $i \geq 0$. All computations are performed on states reachable from a predefined set of initial states.

The algorithms described in this work are implemented using *symbolic model checking* techniques [1]. Boolean formulas can be constructed from the propositional variables of the model. A formula is said to be satisfied in a state if and only if the assignment of variable values in the state to the corresponding variables in the formula makes it true. In general, a formula can be satisfied in many states, and we identify a formula with the set of states that satisfy it. The transition relation can also be represented by a boolean formula constructed from two copies of the propositional variables one for the current state and one for the next state. There is a transition from state v to state v' if the assignment of the variable values in state v to the current state variables, and the assignment of variable values in state v' to the next state variables satisfy the formula.

Our algorithms work on boolean formulas representing sets of states. For example, the formula representing the set of all successors of states in a state set S, $T(S) = \{s' \mid N(s, s') \text{ holds for some } s \in S\}$, can be easily constructed from the formula for S and the formula for the transition relation in one step, regardless of the number of states in S and $T(S)$. The fact that all operations consider sets of states instead of individual states is one of the main reasons for the efficiency of our method. Moreover, boolean formulas are implemented by BDDs, enabling the use of efficient algorithms for their manipulation [1].

We consider the lower bound algorithm first (figure 2). 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. Recall that the function $T(S)$ gives the set of states that are successors of some state in S. The algorithm also uses two variables R and R' to represent sets of states. The function T, the sets R and R', and the operations of intersection and union can all be easily implemented using BDDs.

proc *lower* (*start, final*)
$i = 0$;
$R =$*start*;
$R' = T(R) \cup R$;
while $(R' \neq R \wedge R \cap \mathit{final} = \emptyset)$ **do**
    $i = i + 1$;
    $R = R'$;
    $R' = T(R') \cup R'$;
if $(R \cap \mathit{final} \neq \emptyset)$
    then return i;
    else return ∞;

Figure 2: Lower Bound 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.

Next, we consider the upper bound algorithm (figure 3). 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. $T^{-1}(S') = \{s \mid N(s, s')$ holds for some $s' \in S'\}$). R and R' will again be sets of states. We also denote by *not_final* the set of all states that are not in *final*. As before, the algorithm is implemented using BDDs.

proc *upper* (*start, final*)
$i = 0$;
R =*TRUE*;
R' =*not_final*;
while $(R' \neq R \wedge R' \cap start \neq \emptyset)$ **do**
    $i = i + 1$;
    $R = R'$;
    $R' = T^{-1}(R') \cap not\_final$;
if $(R = R')$
    then return ∞;
    else return i;

Figure 3: Upper Bound 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.

4 Condition Counting Algorithms

In many situations we are interested not only in the length of a path from a set of starting states to a set of final states, but also in measures that depend on the number of states on the path that satisfy a given condition. For example, we may wish to determine the minimum (maximum) number of times a condition holds on a path, or the minimum (maximum) percentage of states that satisfy a given condition, on any path from starting to final states.

Both algorithms in this section take as input three sets of states: *start*, *cond* and *final*. The algorithms compute the minimum and the maximum number of states that belong to *cond*, over all finite paths that begin with a state in *start* and terminate upon reaching *final*.

To guarantee that the minimum (maximum) is well-defined, we assume that any path beginning in *start* must reach a state in *final* in a finite number of steps. This can be checked using the upper bound algorithm described in the previous section. Finally, we ensure that all computations involve only reachable states, by intersecting *start* with the set of reachable states computed *a priori*.

To keep track at each step of the number of states in *cond* that have been traversed, we define a new state-transition system, in which the states are pairs consisting of a state in the original system and a positive integer. Thus, if the original state-transition graph has state set S, then the augmented state set will be $S_a = S \times \mathbb{N}$.

If $N \subseteq S \times S$ is the transition relation for the original state-transition graph, we define the augmented transition relation $N_a \subseteq S_a \times S_a$ as

$$N_a(\langle s, k\rangle, \langle s', k'\rangle) = N(s, s') \wedge (s' \in \mathit{cond} \wedge k' = k + 1 \vee s' \notin \mathit{cond} \wedge k' = k)$$

In other words, there will be a transition from $\langle s, k\rangle$ to $\langle s', k'\rangle$ in the augmented transition relation N_a iff there is a transition from s to s' in the original transition relation N and either $s' \in \mathit{cond}$ and $k' = k+1$ or $s' \notin \mathit{cond}$ and $k' = k$. We also define T to be the function that for a given set $U \subseteq S_a$ returns the set of successors of all states in U. More formally, $T(U) = \{u' \mid N_a(u, u')$ holds for some $u \in U\}$. In the actual BDD-based implementation, an initial bound k_{max} can be selected to achieve a finite representation for k, and new BDD variables can be added dynamically if this bound is exceeded. The system is still finite-state because all paths we consider are finite and k is bounded by their maximum length.

proc *mincount* (*start, cond, final*)
current_min = ∞;
$R = \{\langle s, 1\rangle \mid s \in \mathit{start} \cap \mathit{cond}\} \cup \{\langle s, 0\rangle \mid s \in \mathit{start} \cap \overline{\mathit{cond}}\}$;
loop
    $\mathit{Reached\_final} = R \cap \mathit{Final}$;
    if $\mathit{Reached\_final} \neq \emptyset$ **then**
        $m = \min\{k \mid \langle s, k\rangle \in \mathit{Reached\_final}\}$;
        if $m < \mathit{current\_min}$ **then** $\mathit{current\_min} = m$;
    $R' = R \cap \mathit{Not\_final}$;
    if $R' = \emptyset$ **then return** *current_min*;
    $R = T(R')$;
endloop;

Figure 4: Minimum Condition Count Algorithm

The algorithm for computing the minimum count is given in figure 4. In the algorithm text, *Final* and *Not_final* denote the sets of states in *final* and $S - \mathit{final}$, paired with all possible values of k. More formally:

$$\mathit{Final} = \{\langle s, k\rangle \mid s \in \mathit{final}, k \in \mathbb{N}\} \quad \text{and} \quad \mathit{Not\_final} = \{\langle s, k\rangle \mid s \notin \mathit{final}, k \in \mathbb{N}\}$$

The algorithm uses R to represent the state set in S_a reached at the current iteration, while *Reached_final* and R' are its intersections with *Final* and *Not_final* respectively. Variable *current_min* denotes the minimum count for all previous iterations. The minimum computation over the set of values of k in a formula S can be done by existentially quantifying the state variables (computing $K = \{k \mid \exists \langle s, k\rangle \in S\}$) and following the leftmost nonzero branch in the resulting BDD, provided it uses an appropriate variable ordering.

At iteration i, the algorithm considers the endpoints of paths with i states. The reached states that belong to *final* are terminal states on paths that we need to consider. The minimum count for these paths is computed, using the counter component of the path endpoints, and the current value of the minimum is updated if necessary. For the reached states that do not belong to *final*, we continue the loop after computing their successors. If all reached states are in *final*, there are no further paths to consider and the algorithm returns the computed minimum.

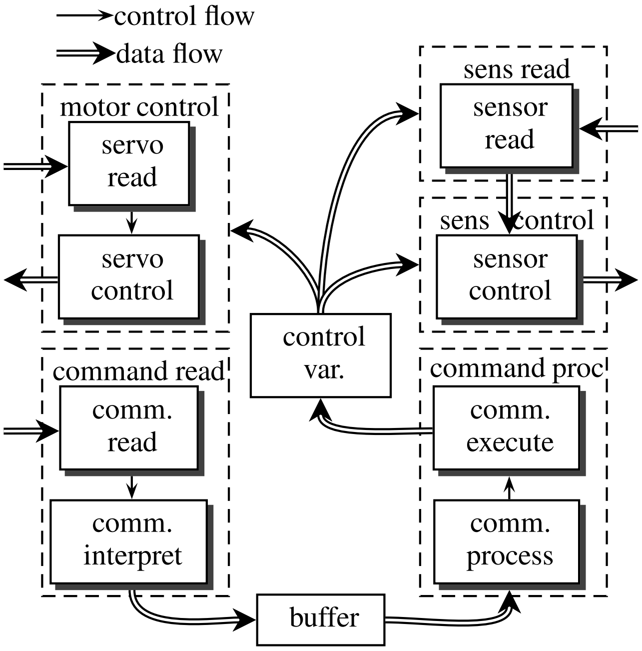

Figure 5: Robot Architecture

Finally, we note that the algorithm for the maximum count has the same structure and can be obtained by replacing min with max and reversing the inequalities. Variants of both algorithms can be used to compute other measures that are a function of the number of states on a path that satisfy a given condition. For example, we can determine the minimum and the maximum number of states belonging to a given set $cond$ over all paths of a certain length l in the state space.

5 A Robotics Example

One application of real-time systems that is becoming increasingly common is in robotic systems. Guaranteeing that tasks are executed within their expected deadline is critical for the integrity of a robot and for the success of its mission. We show how the computation of quantitative properties can assist in validating such systems. The example discussed in this section is derived from the one in [7]. 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 has three subsystems, *motor, measurement* and *command* and can be seen in figure 5. 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 is responsible for receiving commands from the communication link and sending those commands to the appropriate tasks.

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 servomotors, and actuate them. The task consists of two subtasks,

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 6: Timing requirements for the task set

`servo_read` and `servo_control`. The first one is an interrupt routine that reads data directly from the physical devices. The second one processes this 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, the first being an interrupt routine. Upon completion of the interrupt routine, the second subtask starts executing at 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 that are 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. Each subtask has different timing characteristics according to the function it serves. These requirements are summarized in figure 6. Each task consists of several sequential 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 between `sensor_read` and `sensor_control`), and between `command_read` and `command_process`. Priority inheritance protocols have been used to avoid priority inversion [9]. These protocols change the priority of the tasks at synchronization points, thus dividing the tasks into components.

The Analysis of the Robotics Example

The algorithms described can be used to analyze real-time systems in various ways. The schedulability of a task set can be determined by computing the minimum and maximum response times for each task. Performance

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 7: Schedulability analysis

can also be measured in the same way as well as the reaction time to an event. By choosing the initial and final conditions adequately one can identify various timing characteristics of the model. These results can assist in understanding the behavior of the system as well as in evaluating its performance. The condition counting algorithms can help in determining specific timing characteristics of particular paths, allowing fine tuning of the system. This subsection presents an analysis of the robotics example that shows how these techniques can be used to improve the reliability and efficiency of real-time systems design.

Next, we analyze the schedulability of the system. These results allow us to identify inefficiencies and suggest optimizations. The algorithms are also used to analyze the modified design, and to evaluate its usefulness. In this analysis we were able to identify unexpected behaviors that could lead to subtle problems, and to determine how changes in the parameters affect the global system. The objective of this analysis is to show that, although easy to use, these techniques are versatile enough to be of great assistance in specifying and verifying real-time systems.

The schedulability of the original example can be determined in a straightforward way. For example, the response time of the `motor_control` task can be computed by the inserting the following lines in the model description file.

```
COMPUTE MIN[motor_control.start,motor_control.finish]
COMPUTE MAX[motor_control.start,motor_control.finish]
```

Computing response times for all processes generated the table in figure 7. We can see from this table that the task set is schedulable. Moreover, we can also see that many tasks can execute almost until their deadlines arrive. This indicates a high load on the system, and that it is unlikely that adding more tasks to the task set would produce a schedulable system. This type of information gives the designer knowledge about the behavior of the system that can be used to optimize or fine tune the system in many ways.

For example, by looking at the system architecture and the results computed by our algorithms, we were able to suggest changes to the design and to analyze the effects of such changes. In the original design `command_read` generates data that will be used by `command_process`. However, the two tasks execute independently of one another. In some cases `command_process` might execute even if data is not yet available. The same communication pattern occurs between `sensor_read` and `sensor_control`. An inefficiency in the design can be seen in the communication of the sensor tasks. The `sensor_control` task finishes execution before the `sensor_read` task. However, it uses data generated by `sensor_read`, but this data is not available by the time it executes. As a result, `sensor_control` uses data generated by the previous instantiation of `sensor_read`, which may be obsolete. We will change the system to avoid this problem and analyze the resulting design. The modification consists of making `command_read` and `sensor_read` send a termination signal to `command_process` and `sensor_control` respectively. Upon receiving the termination signal, these tasks will initiate execution. Care must be taken, however, because the processes involved have different periods. For example, `command_read` executes twice as often as `command_process`. In this case, `command_process` starts execution every time `command_read` finishes, but it only executes to completion once every two instantiations, otherwise it suspends itself at the beginning of execution. The other two tasks have the opposite behavior. `sensor_read` executes every 100ms, while `sensor_control` executes every 50ms. We change the system so that `sensor_read` signals the execution of `sensor_control` every 100ms, but `sensor_control` also executes independently 50ms 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 below:

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

The new design is not schedulable, since `sensor_control` can take up to 121ms to execute. We can use the same quantitative algorithms to find out more about the behavior of the system and try to correct the problem. The versatility of our method allows us to determine much richer information about the system than just schedulability. 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 121ms to finish. However, whenever executing after `sensor_read`, it takes exactly 26ms to execute in the modified model. Moreover, we can compute that between the start of a periodic execution and its end, `sensor_control` requests execution up to 4 times. 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 below:

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

The system is again schedulable, but now `sensor_control` executes only once every 100ms. 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 7ms, meaning that data produced by `sensor_read` is promptly consumed by `sensor_control`. Therefore we can conclude that in spite of changing the periodicity of `sensor_control` we are still maintaining predictability. The condition counting algorithms are also be used to analyze this model, with some unexpected results. We can 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 management subsystem. During such period, how many times can the 50ms timeout occur? In other words, in an active period how many times should `sensor_control` be activated in the original model? The result is from 1 to 3 times. That is, even though `sensor_control` is producing results within the expected time bounds, it is executing much less frequently than before. This shows that the modified system has a lighter load, but also satisfies the timing constraints of the original system. Notice that simple schedulability tests and reachability analyzers cannot provide enough information for such an analysis. Moreover, not only does our technique allows such results, but it also allows the modelling of realistic examples, which cannot be easily analyzed by non-symbolic tools.

A model of this real-time system was verified using the algorithms described in this paper. This subsection explains the model and briefly discusses the Verus language. The model consists of one process per task. This process executes each component of the task in the appropriate order. A priority driven preemptive scheduler is used to guarantee predictability. An example of the code in the Verus language for one process is given below.

6 Conclusion

This paper describes a tool for computing quantitative properties of real-time systems. A formal state-transition graph model of the system is constructed from a description of the system written in the Verus language. We show how this graph can be used to determine the minimum and maximum delay between two events. We also show how to compute the minimum and maximum number of times a given event can occur on any path between two given state sets. Our approach is very general, since we only require that the systems we analyze be represented as state-transition graphs.

Because our techniques are based on symbolic model checking, they can be applied to designs of realistic size and complexity. Frequently it is possible to search exhaustively state spaces with 10^{30} states in a matter of minutes. We demonstrate the power of these techniques by analyzing a complex robotics system and showing that its timing requirements are satisfied. In addition, our approach provides the user with quantitative information rather than simply verifying that a formula is true in the model. In other words, our algorithms can compute performance measures in addition to verifying correctness. Our techniques can also be used during the design process itself. In the robot controller example we have shown how this information can be used to suggest optimizations to the system. These modifications can again be analyzed to see how the system behavior has changed.

In addition, there are a number of possibilities for extending the range of problems that can be handled by our techniques. In the future we intend to augment the model to allow transitions to be labelled with probabilities. Statistics about different paths in the graph can then be generated. In some cases, we only want paths satisfying certain properties to be considered by our algorithms. Linear temporal logic formulas seem to be ideal for restricting the set of paths considered. It is clear that these extensions would make our approach more powerful and would allow us to handle an even larger class of systems.

References

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

[2] 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.

[3] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications*. North-Holland, 1993.

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

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

[6] R. Gerber and I. Lee. A proof system for communicating shared resources. In *IEEE Real-Time Systems Symposium*, 1990.

[7] 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.

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

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

[10] 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.