Computing Quantitative Characteristics of Finite-State Real-Time Systems

S. Campos E. Clarke W. Marrero M. Minea
H. Hiraishi*

May 4, 1994
CMU-CS-94-147

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

* Dept. of Information and Communication Sciences
Kyoto Sangyo University
Kyoto, Japan

Abstract

Current methods for verifying real-time systems are essentially decision procedures that establish whether the system model satisfies a given specification. We present a general method for computing *quantitative* information about finite-state real-time systems. We have developed algorithms that compute 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. This information leads to a better understanding of system behavior, in addition to determining its correctness. We also show that our technique can be extended to a more general representation of real-time systems, namely, timed transition graphs. The algorithms presented in this paper have been incorporated into the SMV model checker and used to verify a model of an aircraft control system. The results obtained demonstrate that our method can be successfully applied in the verification of real-time system designs.

This research was sponsored in part by the National Science Foundation under grant no. CCR-8722633, by the Semiconductor Research Corporation under contract 92-DJ-294, and by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, SRC, or the U.S. government.

Keywords: real-time systems, finite-state models, formal verification, symbolic model checking, rate monotonic scheduling, schedulability, response time

1 Introduction

A number of algorithms have recently been proposed for verifying the behavior of finite-state real-time systems [3, 6, 8, 9, 10]. These algorithms assume that timing constraints are given explicitly in some notation like temporal logic. Typically, the designer provides a constraint on response time for some operation, and the verifier automatically determines if it is satisfied or not. Unfortunately, after performing the verification, the designer only knows if the timing constraint was met, or if it was violated. In particular, 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.

In this paper we give algorithms to compute quantitative timing information, such as exact upper and lower bounds on the time between a request and the corresponding response. 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 not all parameters have been fixed. In this case, the information provided by our algorithms can be used to establish how changes in a parameter affect the global behavior of the system.

We model a real-time system as a labeled state-transition graph, where each path corresponds to an execution trace of the actual system. We show how to determine the length of the paths leading from a set of starting states (representing the request) to a set of final states (representing the response). In particular, we develop algorithms to compute the minimum and maximum lengths over all such paths, where the minimum length corresponds to the time before which no response can arrive, and the maximum length represents the time after which the given event is guaranteed to have occurred.

Alternatively, we may be interested not only in the length of a path but also in the number of states on this path that satisfy a given condition. For example, a subsystem may request access to a shared bus. The time until it accesses the bus is an important measure of performance. However, the bus may be granted to other subsystems before this request is satisfied. The number of times this happens is a significant indication of the load on the bus and can be computed by algorithms similar to those discussed above. These new algorithms 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.

These algorithms are also extended to *timed transition graphs* (TTG) [6]. TTGs are state-transition graphs where transitions may take more than one time unit to occur. These extensions

allow us to compute quantitative characteristics of real-time systems modelled as TTGs. We believe that the techniques developed can be adapted to other models of computation as well.

All of our algorithms use a discrete model of time. In recent years, there has been considerable research on continuous time models [1, 2, 11, 13, 18]. Most of these models use a transition relation with a finite set of real-valued clocks and constraints on times when transitions may occur. It can be argued that such models lead to more accurate results than discrete time models. However, continuous time models require an infinite state space because the time component in the states can take arbitrary real values. Most verification procedures based on this type of model depend on constructing a finite quotient space called a *region graph* out of the infinite state space. Unfortunately, the region graph construction is very expensive in practice and current implementations of algorithms that use it can only handle at most a few thousand states. Because we use a discrete model of time, we are able to take advantage of *symbolic* techniques [5, 17] in which the transition relation is represented by a binary decision diagram (BDD). This enables us to handle systems that are several orders of magnitude larger than can be handled using continuous time techniques.

To demonstrate how our tools work, we verify a simplified aircraft control system. We model the software that controls the various components of an airplane, and gather timing information about the system using the tools described above. The system consists of set of priority driven processes, where each process is responsible for a subsystem of the aircraft. Subsystems being controlled include navigation, display, radar and weapons. We use the algorithm defined by the *rate monotonic scheduling* theory (RMS) [12, 15, 20] to make the system predictable. This algorithm assigns higher priorities to processes with shorter periods. Optimal response time is guaranteed by the RMS theory if priorities are assigned according to this rule [15].

The RMS theory proposes a schedulability test based on total CPU utilization; a set of processes (which have priorities assigned according to RMS) is schedulable if the total utilization is below a computed threshold. If the utilization is above this threshold, schedulability is not guaranteed. This analysis imposes a series of restrictions on the set of processes. Only certain types of processes are considered with limitations, for example, on periodicity and synchronization.

Another approach to schedulability analysis uses algorithms for computing the set of reachable states of a finite-state system [9, 10]. The algorithms construct the model with the added constraint that whenever an exception occurs (e.g. a deadline is missed) the system transitions to a special exception state. Verification consists of computing the set of reachable states and checking whether the exception state is in this set. No restrictions are imposed on the model in this approach, but the algorithm only checks if exceptions can occur or not. Quantitative information is not generated, and other types of properties cannot be verified, unless encoded in the model as exceptions.

We develop an analysis method that does not impose any restriction except that the system be

modeled as a set of processes that run in parallel and are defined by state-transition graphs. For example, the actual functional behavior of each process can be modeled and analyzed. Schedulability is determined by computing the minimum and maximum execution times for all processes. The process set is schedulable if and only if each process is guaranteed to finish execution before its next period starts. Our technique always determines if the set of processes is schedulable or not, unlike RMS analysis, which may not provide any schedulability information if utilization is above the computed threshold. If the processes are not schedulable, our algorithms determine which specific deadlines are missed and by how much. When no deadline is missed, the same results provide response times for each process, an important performance measure for real-time systems.

Besides determining schedulability, the computation of quantitative characteristics can also provide other valuable information about the system being modeled. Some of the results we have obtained for our example are:

- The amount of time the processor can remain idle while waiting for processes to be scheduled. This information helps the designer to decide if and by how much the load on the processor can be increased without losing schedulability.
- The overhead associated with preemption by other processes. This information is extremely important for determining the amount of priority inversion in a system.
- How responsive subsystems like weapon control are to commands issued by the pilot. In our example, if the pilot presses the fire button a complex sequence of events is generated. We were able to determine the overhead imposed by the firing protocol. This helps the designer understand how this protocol affects the overall response time of the system.

The different types of properties described above show how versatile this approach is. Many other quantitative characteristics can be computed by our algorithms. Moreover, in each case we were able to provide the user with insight into the behavior of the system, as opposed to only asserting its correctness. This information leads 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 defines BDDs, which play an important role in our symbolic methods. Section 3 explains how we model real-time systems. In Section 4 the algorithms for computing the longest and shortest paths between two state sets are presented. Algorithms for counting the number of states that satisfy a given condition along a path between two sets of states are described in section 5. An extension of our methods to timed transition graphs is presented in section 6. Section 7 contains a detailed description of the aircraft control system example mentioned earlier, while section 8 discusses the experimental results obtained. Section 9 concludes the paper with directions for future work.

2 Binary Decision Diagrams

Binary decision diagrams (BDDs) are a canonical representation for Boolean formulas [4]. A BDD is similar to a binary decision tree except that its structure is a directed acyclic graph rather than a tree. This allows nodes and substructures to be shared. The vertices of the graph are labeled with the variables of the Boolean formula, except for the two "leaves" which are labeled with 0 and 1. To insure canonicity, a strict total order is placed on the variables as one traverses a path from the "root" to a "leaf." The edges are labeled with 0 or 1. For every truth assignment there is a corresponding path in the BDD such that at vertex x, the edge labeled 1 is taken if the assignment sets x to 1; otherwise, the edge labeled 0 is taken. If the path ends in the "leaf" labeled 0, then the assignment does not satisfy the formula, and conversely, if the "leaf" reached is labeled 1, then the formula is satisfied by the assignment. Figure 1 illustrates the BDD for the Boolean formula $(a \wedge b) \vee (c \wedge d)$.

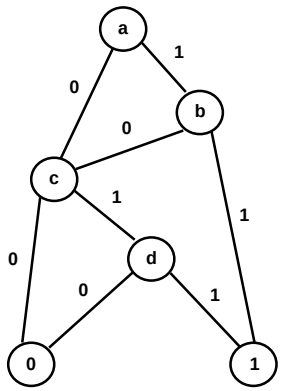

Figure 1: BDD for $(a \wedge b) \vee (c \wedge d)$

In [4], Bryant shows that given a variable ordering, the BDD for a formula is unique. The paper also gives efficient algorithms for computing the BDDs for $\neg f$ and $f \vee g$ given the BDDs for f and g. For the purposes of symbolic model checking, it is also necessary to quantify over Boolean formulas. Bryant describes an algorithm for computing the BDD of a restricted formula such as $f\,|_{v=0}$ or $f\,|_{v=1}$. This allows us to compute the BDD for the formula $\exists v[f]$, where v is a Boolean variable and f is a Boolean formula, as $f\,|_{v=0} \vee f\,|_{v=1}$. However, our implementation uses other known algorithms for performing quantification which are more efficient when multiple variables need to be quantified.

All of the formulas used in our algorithms are represented by BDDs. The BDDs for these formulas are built up in a bottom-up manner. The set of atomic propositions in these formulas is precisely the set of state variables, therefore the BDD for an atomic proposition consists simply of a single BDD variable. Since a formula is built up from atomic propositions using Boolean

connectives, the BDDs for a formula can be constructed using the BDD operations discussed in the previous paragraph. In fact, the implementation allows arbitrary state formulas of computation tree logic (CTL) [7]. These formulas may contain branching time operators as well as logical connectives, but for the sake of simplicity, this discussion is limited to Boolean formulas.

3 Modeling Real Time Systems

The real-time systems we verify are modeled using state-transition graphs. A state $\overline{v}$ in this model can be thought of as a vector assigning values to the state variables $v_1, v_2, v_3, \ldots, v_n$. 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 $i \geq 0$. In addition, we define a set of initial states, and all computations are performed on states reachable from this set.

The transition relation $N(\overline{v}, \overline{v'})$ for the state-transition graph is easily represented using a BDD. The variables of the BDD will consist of two copies of the state variables, one for the current state and the other for the next state. The BDD is the characteristic function of $N(\overline{v}, \overline{v'})$. To see if there is a transition from s to s', we simply assign the values of the state variables in state s, $\langle s_1, s_2, s_3, \ldots, s_n \rangle$, to $\overline{v}$, and similarly for s' and $\overline{v'}$. If the path in the BDD for this assignment ends in the node labeled 1, then there is a transition from s to s', otherwise there is no transition. An example of a state transition graph is given in Figure 2 where the state variables are a and b, and the transition relation N can be represented by the formula $N(\langle a, b \rangle, \langle a', b' \rangle) = (b \wedge b') \vee (a \wedge b') \vee (\neg a \wedge \neg b \wedge a' \wedge \neg b')$.

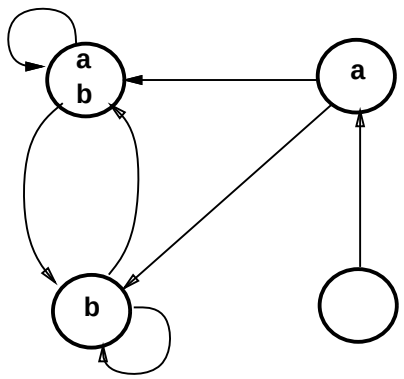

Figure 2: A sample state transition graph

Finally, we demonstrate a close relationship between Boolean formulas, BDDs, and state sets. As stated previously, we use BDDs as a canonical form representation for Boolean formulas. Also, when we consider a formula, we are usually interested in the set of states which satisfy it. We can

find these states easily, using the BDD for the formula. We check that a state satisfies the formula, simply by assigning the values of its state variables to the corresponding variables in the BDD. For this reason, we identify a formula with the set of states which satisfy it. If S denotes a formula (or a set of states), we define $S(\overline{v})$ to be the BDD representing S. We also note that a state satisfies the conjunction of two formulas if and only if it is in the intersection of the sets identified with the formulas. We can make similar statements about disjunction and union, and about negation and complementation. Because discussing sets of states is more intuitive than discussing operations on formulas, we present our algorithms using sets and set operations but the implementation uses BDDs and the corresponding BDD operations.

4 Lower and Upper Bound Algorithms

This section contains the lower and upper bound algorithms. We consider the lower bound algorithm first (figure 3). 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. The function $T(S)$ gives the set of states that are successors of some state in S (i.e. $T(S) = \{s' \mid N(s,s')$ holds for some $s \in S\}$). 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 $(R' \neq R \wedge R \cap$ *final* $= \emptyset)$ **do**
    $i = i + 1$;
    $R = R'$;
    $R' = T(R') \cup R'$;
if $(R \cap$ *final* $\neq \emptyset)$
    then return i;
    else return ∞;

Figure 3: 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. In the formal proof of correctness, we use the following notation:

- S_i is the set of states reachable in i or fewer steps from a state in *start*.
- L is the length of a shortest path from a state in *start* to a state in *final*.

The correctness of the algorithm follows from the following loop invariants:

- $i \leq L$
- $R = S_i$
- $R' = S_{i+1}$

We observe that the three initializing statements insure that the invariants are satisfied before entering the loop. Next we show that the body of the loop maintains the invariants, provided the loop test is satisfied.

- The loop invariant on R, $R = S_i$, and the second half of the loop test, $R \cap \mathit{final} = \emptyset$ imply that $i < L$, otherwise some state in S_i would belong to *final*. This inequality implies $i+1 \leq L$ so we can safely increment i without violating the invariant on i.
- The second statement sets R to the value of R', so now $R = S_{i+1}$. This means that R now satisfies its invariant with the new value of i.
- The last statement sets R' to the union of R' and the image of R'. So by construction, we know that $R' = S_{i+2}$ Therefore, R' satisfies its invariant with the new value for i.

Next we argue about termination. By the definition of S_i, we must have $S_0 \subseteq S_1 \subseteq S_2 \subseteq \cdots$. Since the number of states is finite, only a finite number of the inclusions can be proper, and it must be the case that $S_k = S_{k+1}$ for some $k \geq 0$. From the loop invariant we know that $R = S_i$ and $R' = S_{i+1}$; therefore, the loop cannot execute more than k times without the loop test $R' \neq R$ becoming false.

We finish the proof by analyzing what happens at the final conditional. If $R \cap \mathit{final} \neq \emptyset$, then by the loop invariant, $R = S_i$, s belongs to *final* for some $s \in S_i$. From the definition of S_i, we know that this state is reachable in i or fewer steps from a state in *start*. This gives us an upper bound on L, $L \leq i$. The invariant on i, however, is $L \geq i$. Therefore, it must be the case that $L = i$.

If the test is false, then we must have exited the loop because $R' = R$. From the invariant, $R = S_i$ and $R' = S_{i+1}$; therefore, $S_i = S_{i+1}$. This in turn means that after reaching all the states in S_i we cannot reach any new states (all the edges of states in S_i lead to states in S_i). The false test tells us that no state in R belongs to *final*, and we have just argued that R is the set of all reachable states. Therefore, there is no path from a state in *start* to a state in *final*, so we return infinity.

Next, we consider the upper bound algorithm (figure 4). 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 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' ≠ R ∧ R' ∩ start ≠ ∅) do
        i = i + 1;
        R = R';
        R' = T^{-1}(R') ∩ not_final;
if (R = R')
        then return ∞;
        else return i;
```

Figure 4: 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. We use the following two definitions in proving the algorithm correct:

- S_i is the set of states at the beginning of a path containing i states, all contained in *not_final*.
- M is the number of states in a longest path beginning inside *start* and contained within *not_final*.

Although ultimately we are interested in the number of edges in a longest path, it is easier to reason when we count the number of states in a path. The correctness of the algorithm then follows from the following loop invariants:

- $i \leq M$
- $R = S_i$
- $R' = S_{i+1}$

We observe that the three initializing statements insure that the invariants are satisfied before entering the loop. We can also show that the statements within the loop maintain the invariants, provided the loop test is satisfied.

- The invariant on R', $R' = S_{i+1}$, and the second half of the loop test, $R' \cap start \neq \emptyset$ imply that $i + 1 \leq M$. Therefore, we can increment i without violating the invariant on i.
- The second statement sets R to the value of R' so we know that now $R = S_{i+1}$. This means that R now satisfies its invariant with the new value for i.
- The third statement sets R' to the inverse image of R' intersected with *not_final*. The invariant gave us that $R' = S_{i+1}$. By construction, we now have that $R' \subseteq S_{i+2}$. For the inclusion

in the other direction, we observe that any path of $i+2$ states contained in *not_final* can be thought of as a state labeled with *not_final* that has an edge to a path of $i+1$ states labeled with *not_final*. In other words, the states in S_{i+2} are states in *not_final* with an edge to a state in S_{i+1}. But these are precisely the states just computed for the new value of R' so we get that $S_{i+2} \subseteq R'$. This means that R' also satisfies its invariant with the new value for i.

Now we argue about termination. First, it should be clear from the definition of S_i that $S_0 \supseteq S_1 \supseteq S_2 \supseteq \cdots$. Since we are dealing with a finite number of states, the initial value, S_0 must be a finite set, which in turn means that only a finite number of the inclusions are proper. Therefore, it must be the case that $S_k = S_{k+1}$ for some $k \geq 0$. By the invariant, $R = S_i$ and $R' = S_{i+1}$; thus, the loop cannot execute more than k times without the loop test $R' \neq R$ becoming false.

Before continuing, we make an observation about the loop test. It can never be the case that both parts of the loop condition are false. If we assume that both parts of the loop condition are false, then both $R = R'$ and $R' \cap \mathit{start} = \emptyset$ giving us that $R \cap \mathit{start} = \emptyset$. If we then unroll the loop once, we notice that at the previous iteration, R was assigned the value of R' which would mean that we would have had $R' \cap \mathit{start} = \emptyset$ and we would have exited the loop at that point.

We complete the proof by analyzing what happens at the final conditional. We first consider the case where we exit the loop because $R = R'$. In this case, we have reached a fixed-point. By the invariant, $R = S_i$ and $R' = S_{i+1}$; therefore we have $S_i = S_{i+1}$. We argued previously that the states in S_{i+1} have edges to states in S_i. Since $S_{i+1} = S_i$, we know that every state in S_{i+1} has an edge to another state in S_{i+1}. So every state in S_{i+1} is the beginning of an infinite path of states remaining in $S_{i+1} \subseteq \mathit{not_final}$. The previous observation tells us that $R' \cap \mathit{start} \neq \emptyset$, therefore some state $s \in S_{i+1}$ belongs to *start*. This state then is the beginning of an infinite path starting at a state in *start*, which never reaches a state in *final*, so we return infinity.

If $R' \cap \mathit{start} = \emptyset$, then by the invariant $R' = S_{i+1}$, we know that there is no path of $i+1$ states contained in *not_final* beginning in a state in *start*. No longer path can exist since this would contradict the absence of a path of $i+1$ states, so we have $M \leq i$. But we also have the invariant $i \leq M$, so it must be the case that $M = i$. All edges coming out of the last state on the path lead to states in *final* (otherwise there would be a longer path). Since the transition relation is total, there must exist at least one such edge. Therefore, the longest path from a state in *start* to a state in *final* contains $i+1$ states and has length i.

5 Condition Counting Algorithms

In many situations we are interested not only in the length of a path leading 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 simplify the algorithms, we assume that any path beginning in *start* must reach a state in *final* in a finite number of steps. This requirement is necessary to ensure that the minimum (maximum) is well-defined. It can be checked using the upper bound algorithm described in the previous section. Finally, we assume that all computations involve only reachable states. This can be achieved 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.

The algorithm for computing the minimum count is given in figure 5. 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 can be done by quantifying out the state variables and following the left-

```
proc mincount (start, cond, final)
current_min = ∞;
R = {⟨s,1⟩ | s ∈ start ∩ cond} ∪ {⟨s,0⟩ | s ∈ start ∩ ¬cond};
loop
    Reached_final = R ∩ Final;
    if Reached_final ≠ ∅ then
        m = min{k | ⟨s,k⟩ ∈ Reached_final};
        if m < current_min then current_min = m;
    R' = R ∩ Not_final;
    if R' = ∅ then return current_min;
    R = T(R');
endloop;
```

Figure 5: Minimum Condition Count Algorithm

most nonzero branch in the resulting BDD, provided it uses an appropriate variable ordering. An efficient algorithm that does not depend on the variable ordering is given in [14].

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.

We reason about the correctness of the algorithm by showing that the following invariants are true before the i^{th} iteration of the loop:

- I_1: A pair $\langle s,k\rangle$ belongs to R iff s can be reached from *start* on a path with i states, on which k states are in *cond*, and only the last state is allowed to be in *final*.
- I_2: *current_min* is the minimum number of states in *cond* over all paths with less than i states that begin in *start* and terminate upon reaching *final*, or infinity if there are no such paths.

Initially, R contains the states in *start*, paired with 1 if they belong to *cond* and with 0 otherwise, and *current_min* is infinity. Therefore, both invariants hold before the first loop iteration.

By invariant I_1, the intersection $Reached_final = R \cap Final$ contains all states in *final* reached for the first time by a path containing i states. The count component k of a reached state is, again by I_1, the number of states in *cond* on such a path. Computing the minimum m of these values and setting $current_min = m$ if m is smaller ensures that *current_min* now accounts for paths with up to i states. Therefore, I_2 will hold at the beginning of the next iteration.

Since we only consider paths that reach *final* once, it is correct to continue the state traversal

```
proc maxcount (start, cond, final)
current_max = −∞;
R = {⟨s, 1⟩ | s ∈ start ∩ cond} ∪ {⟨s, 0⟩ | s ∈ start ∩ ¬cond};
loop
    Reached_final = R ∩ Final;
    if Reached_final ≠ ∅ then
        m = max{k | ⟨s, k⟩ ∈ Reached_final};
        if m > current_max then current_max = m;
    R′ = R ∩ Not_final;
    if R′ = ∅ then return current_max;
    R = T(R′);
endloop;
```

Figure 6: Maximum Condition Count Algorithm

only from states in $R' = R \cap Not_final$. If this set is empty, there are no further paths, with more that i states, that reach *final*. Therefore, by invariant I_2, $current_min$ is the correct return value. For the case where the loop is continued, the definition of transition relation ensures that the count component in the augmented state space is incremented on a transition step if and only if the new state is in *cond*. This implies that the count component k represents at all times the number of states in *cond* traversed on a path. Consequently, I_1 will hold again for the new value of R obtained as the image of R' under T.

Next, we argue that the algorithm terminates. The precondition ensures that all paths from *start* reach *final* in a finite number of steps. Thus, we will eventually have $R' = R \cap Not_final = \emptyset$, and the algorithm correctly returns the value $current_min$.

As an optimization, the number of iterations required in certain cases can be reduced by introducing the line

$$R' = R' \cap \{\langle s, k\rangle \mid s \in S \wedge k < current_min\}$$

before testing $R' = \emptyset$. All paths with a count of at least $current_min$ can be safely discarded, which reduces the search to those paths on which the count for *cond* is still smaller than the currently achieved minimum.

Finally, we note that the algorithm for the maximum count, given in figure 6, 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.

6 Timed Transition Graphs

Until this point, we have based our algorithms on a system model in which all transitions take one time unit. However, in actual systems, this assumption is not realistic, because events take different amounts of time to occur. Moreover, the time taken by a transition may change for different executions of the system. These characteristics can be more accurately represented in different models, such as *timed transition graphs* (TTG) [6].

A TTG model is an extension of a state-transition graph, where each transition has associated with it a time range of the form $[l, u]$, where $l, u \in \mathbb{N}$. Formally, the transition relation is given by $N \subseteq S \times \mathbb{N} \times \mathbb{N} \times S$, where S is the set of states of the model. $N(s, l, u, s')$ denotes that the transition between state s and s' can nondeterministically take any number of steps between l and u. The implementation uses a transition relation $\mathcal{N} \subseteq \mathcal{S} \times \mathbb{N} \times \mathcal{S}$ derived from N, such that $\mathcal{N}(s, d, s')$ is true iff there exist $l, u \in \mathbb{N}$ such that $N(s, l, u, s')$ holds, and $l \leq d \leq u$.

In order to compute quantitative characteristics for TTG models we use the same technique as in the algorithms of the previous section. There we augmented the state space with an additional integer variable to count the number of states satisfying a given condition. We will use the same technique to count the number of time steps needed to reach a given state. Thus, we define the augmented state space to be $S_a = S \times \mathbb{N}$ and the transition relation $N_a \subseteq S_a \times S_a$ as

$$N_a(\langle s, t\rangle, \langle s', t'\rangle) = \mathcal{N}(s, d, s') \wedge t' = t + d$$

Informally, there exists a transition in S_a from $\langle s, t\rangle$ to $\langle s', t'\rangle$ if and only if there exists a transition in S between s and s' that may take d steps to occur, and $t' = t + d$.

We can initialize the time component of a set of starting states *start* to 0, and compute the set *Reach* of states in S_a reachable from *start*. We will then have that $\langle s, t\rangle \in$ *Reach* if and only if s can be reached from *start* in t steps. The same *mincount* and *maxcount* algorithms can be used to compute the minimum and maximum number of time steps needed to reach a state set *final* from *start*.

Similarly, we can compute the number of time units on a path spent in states that belong to a given state set *cond*. In TTGs time spent in a state s is defined as the number of time steps it takes to transition out of s. Therefore, to compute the time spent in states belonging to *cond* on a given path, we modify the transition relation for the augmented state space as follows:

$$N_a(\langle s, t\rangle, \langle s', t'\rangle) = \mathcal{N}(s, d, s') \wedge ((s \in cond \wedge t' = t + d) \vee (s \notin cond \wedge t' = t))$$

Notice that this transition relation only increments the time count when $s \in$ *cond*. This means that if a state $\langle s, t\rangle$ is reachable from a state set *start*, then there exists a path leading to s such that t is equal to the number of time units spent in states that belong to *cond* on that path. The

same algorithms, using the new transition relation, will then compute the minimum and maximum time spent in states that belong to *cond*, over all paths leading from *start* to *final*.

By using these variants of the algorithms presented earlier we show how to compute the same type of properties for a more powerful model. This demonstrates that the technique of augmenting the state space can be easily adapted to express and compute different kinds of quantitative information.

7 Example — An Aircraft Control System

One of the most critical applications of real-time systems is in aircraft control. It is extremely important that time bounds are not violated in such systems. Because of the risks involved in the failure of an aircraft, only conservative approaches to design and implementation are routinely used. Many modern techniques for software design such as formal methods are not commonly employed. We believe that formal verification can be very useful in increasing the reliability of these systems by assisting in the validation of schedulability and response times of the various components. This section will briefly describe an aircraft control system used in military airplanes. We have attempted to make this model as realistic as possible. We will then show how some of its timing constraints can be checked using by computing quantitative properties.

System Description

The control system for an airplane can be characterized by a set of sensors and actuators connected to a central processor. This processor executes the software to analyze sensor data and control the actuators. Our model describes this control program and defines its requirements so that the specifications for the airplane are met. The requirements used are similar to those of existing military aircraft, and the model is similar to the one described in [16].

The aircraft controller is divided into systems and subsystems. Each system performs a specific task in controlling a component of the airplane. The most important systems are implemented in our model to provide a realistic representation of the controller. The systems being controlled are:

- Navigation: Computes aircraft position. Takes into account data such as speed, altitude, and positioning data received from satellites or ground stations.
- Radar Control: Receives and processes data from radars. It also identifies targets and target position.
- Radar Warning Receiver: This system identifies possible threats to the aircraft.
- Weapon Control: Aims and activates aircraft weapons.

- Display: Updates information on the pilot's screen.
- Tracking: Updates target position. Data from this system is used to aim the weapons.
- Data Bus: Provides communication between processor and external devices.

Each system is composed of one or more subsystems. Timing constraints for each subsystem are derived from factors such as required accuracy, human response characteristics and hardware requirements. For example, the screen must be updated frequently enough so that motion appears continuous. To accomplish this, the update must occur at least once every 50ms. The following table presents the subsystems being modelled, as well as their major timing requirements. The priority assignment will be explained subsequently.

System	Subsystem	Period	Exec.	% CPU	Priority
Display	status update	200	3	1.50	12
	keyset	200	1	0.50	16
	hook update	80	2	2.50	36
	graphic display	80	9	11.25	40
	store update	200	1	0.50	20
RWR	contact mgmt.	25	5	20.00	72
Radar	target update	50	5	10.00	60
	tracking filter	25	2	8.00	84
NAV	nav update	50	8	16.00	56
	steering cmds.	200	3	1.50	24
Tracking	target update	100	5	5.00	32
Weapon	weapon protocol	200*	1	0.50	28
	weapon aim	50	3	6.00	64
	weapon release	200**	3	1.50	98
Data Bus	poll bus devices	40	1	2.50	68

* Weapon protocol is an aperiodic process with a deadline of 200ms.

** Weapon release has a period of 200ms, but its deadline is 5ms.

Concurrent processes are used to implement each subsystem. Communication among the various processes is done indirectly. No data is shared directly by two subsystems. Processes communicate only through data servers called *monitor tasks*. Each system maintains a server process that accepts requests for data, and returns the desired information. The various subsystems in each system update the data in the servers. Monitor tasks only accept requests, respond to them, and then block. They are assigned low priority, and priority inheritance is used to maintain predictability [6, 19].

With the exception of the weapon system, all other systems contain only periodic processes, which are scheduled to execute at the beginning of their period. When a process is granted the CPU it acquires the data it needs through the monitor tasks, executes, updates information on its own data server, and blocks waiting for its next execution period.

The weapon system contains a mixture of periodic and aperiodic processes. It is activated when the display keyset subsystem identifies that the pilot has pressed the firing button. This event causes the weapon protocol subsystem to be activated. It then signals the weapon aim subsystem that had been blocked. Weapon aim is then scheduled to be executed every 50ms. It aims the aircraft weapons based on the current position of the target. It also decides when to fire and then starts the weapon release subsystem. The firing sequence can be aborted until weapon release is scheduled, but not after this point. Weapon release then executes periodically and fires the weapons 5 times, once per second.

In order to enforce the different timing constraints of the processes, priority scheduling is used. Predictability is guaranteed by scheduling the processes using *Rate Monotonic Scheduling* (RMS) [12, 15]. The RMS theory provides an algorithm for assigning priorities to processes in order to predict their behavior. Higher priority is given to processes with shorter periods. This algorithm has been shown to be an optimal static priority algorithm with respect to response time requirements. This means that if a process set is schedulable using a fixed priority scheduling policy, then it is schedulable using this algorithm. However, the RMS algorithm itself does not determine if a process set is schedulable or not.

8 Verification of the Aircraft Control System

Implementation

We have implemented this control system in the SMV language [17]. The SMV model checker has been used to verify its functional correctness, while its timing correctness has been checked using the quantitative algorithms described in this paper. Most of the characteristics described above were implemented, although some abstractions have been performed for simplicity. A more detailed description of the implementation follows.

A time quantum of 1ms was used, in other words, a transition corresponds to a delay of 1ms in our model. A global timer is implemented that starts periodic processes when their period arrives. Whenever awakened, a process requests execution and waits until it has been granted the CPU. The process then runs for its defined execution time. An internal counter stores the time since execution has started. After executing, the process releases the CPU and blocks, waiting for the next period.

The time to request data from a monitor task and wait for the response is assumed to be small compared to the total execution time. This is reasonable if we assume an efficient implementation. Sending request and response messages takes only a small amount of time. Processing in the monitor tasks is also fast, considering the limited range of functions performed. The assumption can only be violated if blocking due to synchronization is long. The access pattern to the monitor

tasks, however, minimizes this possibility. They simply receive requests, retrieve the data from memory, and return it. There are no nested critical sections. Moreover, the priority inheritance protocols used maintain predictability and eliminate the possibility of unbounded blocking due to synchronization [6, 19]. Since blocking times can be computed, we assume they are included in the execution time defined. A more detailed model can be constructed to remove this assumption, but because of the reasons outlined above, we believe this would not change the results significantly. In order to optimize response time, we have implemented a preemptive scheduler. It accepts requests for execution and chooses the highest priority process requesting the CPU. If a request arrives from a higher priority process after execution has started, the scheduler preempts the executing process and starts the higher priority one. When a process finishes executing it resets its request, and the scheduler chooses another process. If data was shared directly, synchronization could cause deadlocks. This could happen, for example because of cyclic dependencies among locks. Monitor tasks avoid this problem because they eliminate the possibility of complex data dependencies.

We have also implemented a non-preemptive scheduler. Preemptability is a feature that may not always be available, and we wanted to observe the effects of removing this feature from the model. In this case, once a process starts executing, it continues executing until it voluntarily releases the CPU. If a higher priority process requests execution, it has to wait until the running process finishes. Non-preemptive schedulers usually cause response time for higher priority processes to be higher. They are however simpler to implement, and allow for simpler programs (for example, the deadlock problem described above does not exist if no preemption occurs). Having both types of scheduler in our model allowed us to extend our results to a larger class of systems.

Verification Results

Schedulability is one of the most important properties of a real-time system. It states that no process will miss its deadline. In this example the deadlines are the same as the periods (except for the weapon release subsystem). The RMS theory checks for schedulability by computing the CPU utilization of the process set. Our algorithms, however, use a different approach. We compute the minimum and maximum execution times for each process, and check if they always finish before their deadline. Notice that our approach always determines the schedulability. The RMS analysis may not determine it, if certain conditions do not hold for the system. Another advantage of this technique is that our algorithms only require that processes be modelled as state graphs, while RMS imposes restrictions on their behavior.

The following table summarizes the execution times computed by the algorithms. Processes are shown in decreasing order of priority. Deadlines are also shown so that schedulability can be easily checked. The minimum and maximum execution times are given for both the preemptive

and non-preemptive schedulers.

Subsystem	Deadline	Execution Times			
		Preemptive		Non Preempt.	
		Min	Max	Min	Max
Weapon release	5	3	3	3	9
Radar tracking filter	25	2	5	2	10
RWR contact mgmt.	25	7	10	7	15
Data bus poll	40	1	11	1	14
Weapon aim	50	10	14	2	18
Radar target update	50	12	19	12	19
NAV update	50	20	34	20	27
Display graphic	80	10	44	10	43
Display hook update	80	14	46	14	47
Tracking target update	100	26	51	26	51
Weapon protocol	200	1	21	3	46
NAV steering cmds.	200	35	85	36	74
Display store update	200	36	95	37	97
Display keyset	200	37	96	38	98
Display status update	200	40	99	41	101

We can see from the table above that the process set is schedulable using preemptive scheduling. An analysis of a similar process set using RMS showed that only the first eight processes were guaranteed to meet their deadlines [16]. From our results we can also identify many important parameters of the system. For example, the response time is usually very low for best-case computations, but it is also good for the worst case. Most processes take less than half their required time to execute. This indicates that the system is still not close to saturation, although the total CPU utilization is high.

Notice also that preemption does not have a big impact on response times. Except for the most critical process, all others maintain their schedulability if a non-preemptive scheduler is used. Moreover, we can see that although non-preemption causes weapon release to miss its deadline, but by a relatively small amount. If a preemptive scheduler were expensive, reducing the CPU utilization slightly might make the complete system schedulable without changing the scheduler. By having such information the designer can easily assess the impact of various alternatives to improve the performance, without having to change the implementation. It should be noted that an analysis of this type can't be done using methods like the RMS utilization test or reachability computation.

As another example of how the designer can use these results, we can analyze the response time for the display graphic subsystem. The periodicity of this subsystem is 80ms and a shorter period might be desired to make motion look continuous. However, the response time of this process can be as high as 46ms. Changing the period to 40ms would most likely make it miss its deadline. The designer may choose to decrease it only to 50ms, but this is still close to the response time, and the

increased load might make the system not schedulable. The model can be easily changed to check this hypothesis, but this analysis shows that it is unlikely that decreasing the period will maintain schedulability.

The results obtained about execution times can also be used to infer how other properties of the system might be affected by changes. For example, the maximum continuous executing time and the maximum continuous idle time for the CPU can be computed. In this model, the CPU executes for a maximum of 99ms before becoming idle, and can remain idle for at most 18ms before executing again. The maximum idle time shows that the processor is not saturated, and more processes could be added to the process set. On the other hand, the maximum executing time indicates that the processor load is high, considering that most processes have periods smaller than 100ms. This means that although more processes could be added, the response time of lower priority processes might become significantly higher.

The effect of preemption on execution time can be assessed as well. We have computed the maximum and minimum execution times for processes *after* they have been granted the CPU. If minimum and maximum are not the same, the process can be preempted after starting execution. For example, the display graphic subsystem can finish in as little as 7ms and in as much as 14ms after it starts execution. In other words, preemption overhead can be as high as 7ms for this subsystem. The NAV steering subsystem has a minimum of 1ms and a maximum of 44ms. This means that other processes can delay it for 43ms. It is clear that NAV steering can be preempted for a longer time than display graphic, since it has lower priority. Our results, however, allow us to determine how much longer it can be preempted. As an important variation of this property, we can compute the priority inversion time for high priority processes. This can help identify the reasons why a system is not predictable, and help correct its behavior.

We examine one more property of this particular model. The weapons system is critical to the aircraft. It is very important that it responds quickly to the pilot's command. However, when a pilot presses the firing button, many subsystems are involved in identifying and responding to this event. We can determine its response time using the algorithms described previously. By computing the minimum and maximum times between pressing the fire button and the execution of the weapon release process we are able to determine if the weapon system responds quickly enough to satisfy the aircraft requirements. In our example, the minimum time between detecting that the fire button has been depressed and the end of execution of weapon release is 120ms. The maximum time is 167ms, not accounting for the possibility that the firing sequence may be aborted, or that weapon aim may lose contact with the target. Of course external events have to be added to these numbers, such as the time between pressing the button and it being detected by display keyset, or the time it takes to actually fire the weapons. But the designer of the system

now knows how much time the firing protocol adds to these external factors in the actual airplane. Again, this type of analysis may be difficult to do with other tools. The RMS schedulability test cannot give tight bounds on specific response times for such properties, since its only parameter is CPU utilization. Algorithms that use reachability analysis are also inappropriate for such analysis. Specific exceptions, with previously defined time bounds, would have to be added to the model to observe these characteristics.

The finite-state model was implemented in about 600 lines of SMV code. The final model has about 10^{15} states, and the transition relation uses approximately 4600 BDD nodes. To compute each property described above took between 5 and 15 seconds using an i486 based workstation.

9 Conclusion

This paper proposes a general framework for computing quantitative characteristics of finite-state real-time systems. We have devised algorithms that calculate exact numerical bounds on the delay between two specified events, as well as on the frequency of the occurrence of a condition within a given interval. Rather than just determining the correctness of the model, the results computed by our algorithms provide hints about its behavior that can be useful in improving the performance of the system.

Our method can be easily integrated with model checking techniques. In fact, the *lower* and *upper bound* algorithms have been added to the most recent version of the SMV model checking system. Using this implementation we demonstrate the practical importance of our approach by analyzing a model of an aircraft control system. We have been able to obtain stronger results than those produced using traditional methods for real-time system verification.

We have found this approach to be very flexible. We have shown how quantitative characteristics can be computed for state-transition graphs. In addition, we have extended the algorithms to models in which transitions may take more than one time unit. We also plan to investigate the application of these techniques to other models of computation, such as continuous time and hybrid systems.

We believe that the quantitative information that our method provides can be extremely useful to designers during the development of real-time systems. We are confident that these techniques will prove practical in the verification of a variety of other realistic designs.

References

[1] R. Alur, C. Courcourbetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Symp. on Logic in Computer Science*, pages 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] R. Alur and T. A. Henzinger. Logics and models of real-time: a survey. In *Lecture Notes in Computer Science, Real-Time: Theory in Practice*. Springer-Verlag, 1992.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[5] 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.

[6] S. V. Campos and E. M. Clarke. Real-time symbolic model checking for discrete time models. In *First AMAST International Workshop in Real-Time Systems*, 1993.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

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

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

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

[11] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proceedings of the 7th Symp. on Logic in Computer Science*, 1992.

[12] 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.

[13] H. Lewis. A logic of concrete time intervals. In *Proceedings of the 5th Symp. on Logic in Computer Science*, pages 380–389, 1990.

[14] B. Lin and A. R. Newton. Efficient symbolic manipulation of equvialence relations and classes. In *Proceeding of the Int. Workshop on Formal Methods in VLSI Design*, 1991.

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

[16] C. D. Locke, D. R. Vogel, and T. J. Mesler. Building a predictable avionics platform in ada: a case study. In *IEEE Real-Time Systems Symposium*, 1991.

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

[18] 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.

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

[20] 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.