Timing Analysis of Industrial Real-Time Systems*

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

Abstract

In this paper, we describe a formal method for modelling real-time systems and a procedure to compute the model's timing characteristics automatically. We present algorithms that compute exact bounds on the delay between two specified events. We also describe an algorithm to count the minimum and maximum number of times an event occurs between a given starting condition and an ending condition. These algorithms are based on symbolic model checking *techniques [6, 24] which have been successfully used to find bugs in several industrial designs. Such techniques can be used to search exhaustively state spaces with up to 10^{30} states. To illustrate the usefulness of our method, we describe the timing analysis for a patient monitoring system with more than 10^{13} states. We also present the timing analysis and verification for an aircraft controller. The sizes of the examples we verify demonstrate that our tool can be applied to realistic industrial designs.*

1 Introduction

Symbolic model checking is today an industrial-strength formal specification and verification method. It has been applied successfully in the verification of several industrial designs. It has been used to find bugs in the Futurebus$^+$ cache coherence protocol [11] which is an IEEE standard and which has been adopted by the U.S. Navy. It is also currently being used by a number of semiconductor companies in the validation of their new products. Using symbolic model checking techniques it is possible to verify finite state systems with an extremely large number of states. State spaces with up to 10^{30} states can be exhaustively searched in minutes. Models with more than 10^{120} states have been verified using special techniques. This paper briefly introduces the symbolic model checking approach and describes how it can be used to verify properties of real-time systems. It also shows how these techniques can be extended to compute *quantitative* timing information that can help in understanding the behavior of the system as well as evaluating its performance.

The model checker accepts the description of the system being verified in a formal specification language and then compiles this specification into a finite state-transition graph. Properties about the system are expressed as formulas in a temporal logic which uses the state-transition graph as a model. Model checking consists of traversing such a graph and verifying if it satisfies the formula representing the property [9, 10]. Symbolic model checking uses boolean formulas to represent the state-transition graph [6, 24] and to represent sets of states. This representation makes it possible to do computations, such as computing successors, on sets of states instead of on individual states. These formulas are implemented using *binary decision diagrams* (BDDs) [4] which can be manipulated efficiently. BDDs usually generate very compact representations by eliminating redundancy in formulas.

Model checking and several other methods recently proposed [3, 7, 15, 16, 17] to verify real-time systems assume that timing constraints are given explicitly in some notation like temporal logic. The verifier then determines if the system satisfies the timing constraint or not. No other information about its performance and behavior is provided. The algorithms proposed in this work extend the above technique by computing *quantitative timing information* about the system. This allows for a more detailed analysis than currently available in similar tools. These algorithms provide insight into *how well* a system works, rather than just determining whether it works at all. Our approach enables 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.

The first two algorithms compute the exact lower and upper bounds on the amount of time that elapses between two events, such as a request and a corresponding response. In our state-transition graph used to model the system, this corresponds to the minimum and maximum length of a path between two sets of states. 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 any such interval. For example, a subsystem may request execution. The time until it finishes execution can be critical for system correctness. However, before the subsystem completes its task the processor may be granted to other processes. The amount of time spent on other tasks while the subsystem is waiting is an important performance measure and can be computed by algorithms similar to those

*This research is sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under Grant F33615-93-1-1330. and in part by the National Science Foundation under Grant No. CCR-9217549 and in part by the Semiconductor Research Corporation under Contract 92-DJ-294. 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 the U.S. government.

mentioned above. We also present algorithms to compute this kind of information. Specifically, in the state-transition graph model, these 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.

All of our algorithms use a discrete model of time. In recent years, there has been considerable research on continuous time models [1, 2, 13, 18, 20, 25]. 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 [6, 24] in which the transition relation is represented by a binary decision diagram (BDD). This enables us to handle systems that are many orders of magnitude larger than can be handled using continuous time techniques.

Other approaches for analyzing real-time systems exist. The *rate monotonic scheduling* theory (RMS) [19, 22, 26] is one example. Given a set of processes and their timing constraints, it proposes a priority assignment algorithm that 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 [22]. 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 [16, 17]. 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.

In comparison, our method 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.

Several industrial real-time systems have been modelled and verified using the algorithms described in this paper. Model checking techniques have been used to verify their logical correctness, while quantitative algorithms have been used to evaluate their performance. The first example is a medical monitoring system. Sensors connected to a patient continuously measure various parameters of his or her condition. The system records this data for analysis by physicians and also issues an alarm when abnormal conditions occur. Priority driven concurrent processes are used to control the various components of the monitor. The analysis of the system consists of verifying if the performance of the controller satisfied its expected response time. The results produced by our quantitative algorithms also allowed us to identify inefficiencies in the design and suggest optimizations. The modified model was then analyzed and its performance once again evaluated. The information generated by the algorithms made it possible not only to analyze the original design, but also to improve it.

The second example is an aircraft control system. This example is derived from the one described in [23]. Its timing requirements are representative of those found in actual aircraft. 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 [19, 22, 26] to make the system predictable. The original analysis of the example was able to show that only some of the processes were schedulable, while no information was given on the others [23]. Using the algorithms presented in this paper, we were able to determine the schedulability of the complete task set for this example. We were also able to determine other critical performance information, such as the reaction time of the weapons subsystem. In both examples the state space of the final model has between 10^{13} and 10^{15} states, but its logical properties and timing characteristics can be computed in few seconds on a i486 based workstation. The memory requirement for this computation was about two megabytes.

These examples demonstrate that our tools can be used for the specification and verification of designs of real-time systems used in industry . The fact that most properties could be computed in seconds shows that even larger examples can be modelled and verified. We believe that the techniques described are mature enough to be used in an industrial environment, and that they can be of significant assistance in improving the efficiency and reliability of real-time designs.

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 symbolic model checking. 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. Sections 6 and 7 present the verification and timing analysis of a medical monitoring system and an aircraft controller respectively. Section 8 concludes the paper.

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 $(a \wedge b) \vee (c \wedge d)$.

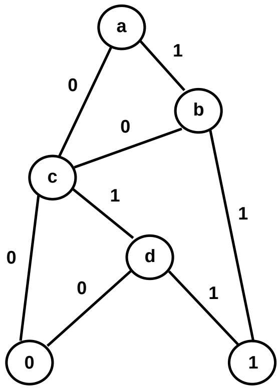

Figure 1: BDD for $(a \wedge b) \vee (c \wedge d)$

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 $f\mid_{v=0}$ or $f\mid_{v=1}$. 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 $f\mid_{v=0} \vee f\mid_{v=1}$. 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) [10]. 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 Symbolic Model Checking

Temporal logic model checking is a technique for determining the correctness of finite-state systems [9, 10]. In this technique, specifications are written as formulas in a propositional temporal logic and computer systems are represented by state-transition graphs. Verification is accomplished by an efficient breadth first search procedure that views the transition system as a model for the logic, and determines if the specifications are satisfied by that model. There are several advantages to this approach. An important one is that the procedure is completely automatic. Another advantage is that, if the formula is not true, the model checker will provide a counterexample. The counterexample is an execution trace that shows why the formula is not true. This is an extremely useful feature because it can help locate the source of the error and speed up the debugging process. Another advantage is the ability to verify partially specified systems. Useful information about the correctness of the system can be gathered before all the details have been determined. This allows the verification of a system to proceed concurrently with its design. Consequently verification can provide valuable hints that will help designers eliminate errors earlier and define better systems. Model checkers achieve great efficiency through the use of *symbolic* implementation techniques [24]. Symbolic model checkers represent states and transitions using boolean formulas. Implementing these boolean formulas as BDDs leads to very efficient algorithms for model checking that are able to verify systems with extremely large state spaces. This section will first describe the method used to represent the state-transition graph using boolean formulas. It will then briefly describe the logic used to express the properties to be verified. The model checking algorithm will not be presented here for brevity. More information on symbolic model checking can be found in [7, 5, 6, 12, 24].

Representing the Model

A model of the system in our algorithm is a labeled state-transition graph $\mathcal{M}$. The key to the efficiency of the algorithm is to use BDDs to represent the labeled state-transition graph and to verify if the formula is true or not. The following method will be used to represent the transition relation as a BDD. Assume that system behavior is determined by the boolean variables $V = \{v_0, \ldots, v_{n-1}\}$. Let $V' = \{v'_0, \ldots, v'_{n-1}\}$ be a second copy of these variables. We will use the variables in V to represent the value of the variables in the current state, and the variables in V' to represent the value in the next state. The relationship between values of variables in the current and the next states is written as a boolean formula using V and V'. This will generate the boolean formula N representing the transition relation. This formula will then be converted to a BDD.

$$N(v_0, \ldots, v_{n-1}, v'_0, \ldots, v'_{n-1})$$

Computation Tree Logic

Computation tree logic, CTL, is the logic used by in our model checker to express properties that will be verified. *Computation trees* are derived from state transition graphs. The graph structure is unwound into an infinite tree rooted at the initial state, as seen in figure 2. Paths in this tree represent all possible computations of the program being modelled. Formulas in CTL refer to the computation tree derived from the model. CTL is classified as a *branching time* logic, because it has operators that describe the branching structure of this tree.

Formulas in CTL are built from atomic propositions, where each proposition corresponds to a variable in the model, boolean connectives $\neg$ and $\wedge$, and *temporal operators*. Each operator consists of two parts: a path quantifier followed by a temporal operator. Path quantifiers indicate that the property should be true of *all* paths from a given state (**A**), or *some* path from a given state (**E**). The temporal quantifier describe how events should be ordered with respect to time for a path specified by the path quantifier. They have the following informal meanings:

- **F** φ (φ holds sometime in the future) is true of a path if there exists a state in the path that satisfies φ.
- **G** φ (φ holds globally) is true for a path if φ is satisfied by all states in the path.
- **X** φ (φ holds in the next state) means that φ is true in the next state of the path.
- φ **U** ψ (φ holds until ψ holds) is satisfied by a path if ψ is true in some state in the path, and in all preceding states, φ holds.

Some examples of CTL formulas are given below to illustrate the expressiveness of the logic.

- $\mathbf{AG}(req \rightarrow \mathbf{AF}\ ack)$: It is always the case that if the signal req is high, then eventually ack will also be high.
- $\mathbf{EF}(started \wedge \neg ready)$: It is possible to get to a state where $started$ holds but $ready$ does not hold.
- $\mathbf{AG}\ \mathbf{EF}\ restart$: From any state it is possible to get to the $restart$ state.
- $\mathbf{AG}(send \rightarrow \mathbf{A}[send\ \mathbf{U}\ recv])$: It is always the case that if $send$ occurs, then eventually $recv$ is true, and until that time, $send$ must remain true.

4 Lower and Upper Bound Algorithms

This section presents the first two algorithms for computing quantitative information of real-time systems. These algorithms compute minimum and maximum time delays between specified events. A real-time system is modelled as a state-transition graph in the way described previously. Recall that our algorithms work on boolean formulas representing sets of states. For example, given a formula representing a set of states S, the formula for $T(S) = \{s' \mid N(s, s') \text{ holds for some } s \in S\}$, the set of all successors of states in S, can be 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 $T(S)$. In particular, if $S(v_0, \ldots, v_{n-1})$ is the formula for S then the formula for $T(S)$ is $\exists v_0, \ldots, v_{n-1}[S(v_0, \ldots, v_{n-1}) \wedge N(v_0, \ldots, v_{n-1}, v'_0, \ldots, v'_{n-1})]$. 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.

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. 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 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.

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.

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*.

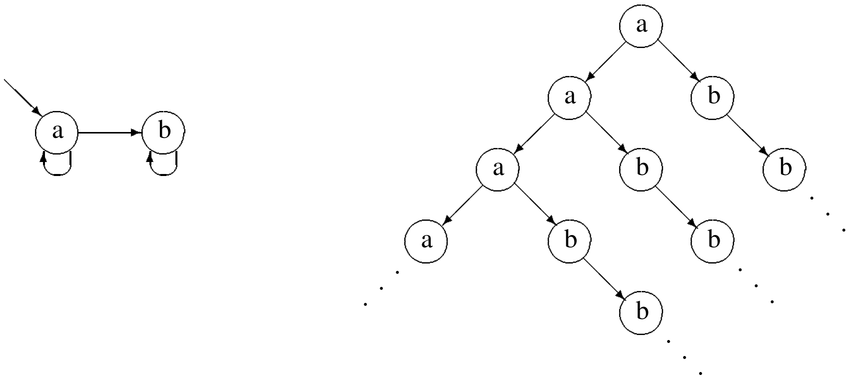

Figure 2: State transition graph and corresponding computation tree

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 4: Upper Bound Algorithm

- 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 proving that the following expressions are loop invariants: $i \leq M$; $R = S_i$; and $R' = S_{i+1}$. The proof can be found in [8].

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

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

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 $s' \in cond$ and $k' = k + 1$ or $s' \notin cond$ 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 $S - final$, paired with all possible values of k. More formally:

$$Final = \{\langle s, k\rangle \mid s \in final, k \in \mathbb{N}\}$$

and

$$Not\_final = \{\langle s, k\rangle \mid s \notin final, k \in \mathbb{N}\}$$

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

Figure 5: Minimum Condition Count Algorithm

proc *maxcount* (*start, cond, final*)
current_max = $-\infty$;
$R = \{\langle s, 1\rangle \mid s \in \mathit{start} \cap \mathit{cond}\} \cup$
$\quad \{\langle s, 0\rangle \mid s \in \mathit{start} \cap \overline{\mathit{cond}}\}$;
loop
$\quad \mathit{Reached\_final} = R \cap \mathit{Final}$;
$\quad$**if** $\mathit{Reached\_final} \neq \emptyset$ **then**
$\quad\quad m = \max\{k \mid \langle s, k\rangle \in \mathit{Reached\_final}\}$;
$\quad\quad$**if** $m > \mathit{current\_max}$ **then**
$\quad\quad\quad \mathit{current\_max} = m$;
$\quad R' = R \cap \mathit{Not\_final}$;
$\quad$**if** $R' = \emptyset$ **then return** *current_max*;
$\quad R = T(R')$;
endloop;

Figure 6: Maximum Condition Count Algorithm

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. An efficient algorithm that does not depend on the variable ordering is given in [21].

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: $\mathit{current\_min}$ 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 $\mathit{current\_min}$ is infinity. Therefore, both invariants hold before the first loop iteration.

By invariant I_1, the intersection $\mathit{Reached\_final} = R \cap \mathit{Final}$ 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 $\mathit{current\_min} = m$ if m is smaller ensures that $\mathit{current\_min}$ 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 only from states in $R' = R \cap \mathit{Not\_final}$. If this set is empty, there are no further paths, with more that i states, that reach *final*. Therefore, by invariant I_2, $\mathit{current\_min}$ 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 $R' = R \cap \mathit{Not\_final} = \emptyset$, and the algorithm correctly returns the value $\mathit{current\_min}$.

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

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

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 A Medical Monitoring Example

This section presents a patient monitoring system derived from the one presented in [14]. It is a realistic example that models many features existing in actual systems. The example has been expanded to show how the algorithms described in this paper can be used to analyze models of industrial complexity. The resulting model for this example has more than 10^{13} states but its timing characteristics can be computed in a few seconds.

The system consists of a set of processes and can be seen in figure 7. The *acquire* process is the only periodic process in the system, all others are aperiodic. *Acquire* executes every 20ms, and its function is to read data from sensors monitoring the patient. Usually, the data read by the sensors contain spurious information. In order to eliminate erroneous data, the output of *acquire* is sent to the *filter* process. *Filter* is an aperiodic process. It is triggered whenever data is read from the sensors, that is, whenever *acquire* finishes its execution. The *filter* process is dependent on data generated by the *acquire* process. The same dependency pattern is also used to trigger execution of the other aperiodic processes. After *filter* executes, its results are analyzed by the patient condition detection processes. *Filter* preprocesses the data generated, and may decide to start the detection processes or not, depending on the data available. Three such processes are modelled in this example to detect abnormal conditions in the blood pressure, heart rate and temperature. The detection processes can issue an alarm after analyzing the data. If the *alarm* process is executed, it also starts the *audio* process that generates the actual alarm signal. Finally, the *filter* process also sends its data to the *display* and *recorder* processes, that display the data on the screens and record it in some non-volatile media for future analysis. The execution times for the processes in the system can be summarized as follows. The *acquire* process executes for 1ms, the *filter* process executes for 3ms, and all other processes execute for 2ms.

Most processes in this system are aperiodic in nature. Because of this, methods such as the rate monotonic scheduling (RMS) [19, 22, 26] cannot be directly used to analyze this process set. For example, the assignment of priorities to processes is more complex than in the periodic case which can use the RMS algorithms. In this example priorities have been assigned heuristically, and quantitative algorithms have been used to investigate the efficiency of the assignment. Initially, the priority order defined was, from the highest to the lowest priority process: *acquire, filter, blood_pressure, heart_rate, temperature, display, recorder, alarm*, and *audio*.

The aperiodic nature of the processes also makes it difficult to determine the schedulability requirements. Except for the *acquire* process, no other process has a deadline. Nevertheless, the timing constraints of the system can be easily identified. The *acquire* process has a period and a deadline of 20ms. The timing constraints for the other processes can be defined in several ways. A straightforward way is to require that all processes to finish before the next execution of *acquire*. Our algorithms can determine if the process set satisfies this constraint by computing minimum and maximum times between the moment when *acquire* requests execution and the moment when each process terminates. However, this requirement can be too restrictive in some cases. Overlapping the execution of consecutive process instantiations is acceptable if the response time can still be bounded. The algorithms described in this paper can determine response times for all processes by checking if there exists a process that can execute for an unbounded amount of time. If there is such a process, then the system is not schedulable. If not, these results allow the designers to check if the response times are acceptable. Both results have been computed for this example, and are presented in the following table.

Process	Period	Execution Times			
		(1)		(2)	
		min	max	min	max
acquire	20	1	1	1	1
filter	-	4	4	3	3
blood pressure	-	6	∞	2	2
heart rate	-	6	∞	2	4
temperature	-	6	∞	2	6
display	-	6	12	2	8
recorder	-	8	14	4	10
alarm	-	12	∞	6	10
audio	-	14	∞	2	2

- (1) Minimum and maximum times between the start of *acquire* and the end of execution of the process. If the maximum time is less than the period of *acquire*, then the process will finish execution before the next instantiation of *acquire* is started.
- (2) Minimum and maximum times between the start and end of execution of each process. If this time is less than infinity, then the system is schedulable.

In some cases, it is possible that the condition detection processes are never executed, as well as the *alarm* and *audio* processes. Because of this, the maximum time from the start of *acquire* until these processes finish is infinity. However, in many situations it is important to know the maximum time until an event provided it will occur. We can change the model to reflect that an alarm will always be issued, and compute such information. In this model, we determined that from the moment *acquire* reads abnormal data until the alarm sounds, less than 18ms will elapse (16ms for *alarm* and 18ms for *audio*).

The results produced by our algorithms can provide more information about the behavior of the system than just determining its schedulability. For example, we can see from the data presented that the *alarm* and *audio* processes are the ones with highest response times. However, sounding the alarm is a critical function that should not be postponed by other functions such as recording the data on tape. One way to avoid this problem is by raising the priority of *alarm* to avoid interference from less important processes and compute the response times for the modified model. We raised the priority of the *alarm* process by changing the priority order to: *acquire, filter, alarm, blood_pressure, heart_rate, temperature, display, recorder*, and *audio*. The response times were computed again, and the results are presented in the table below:

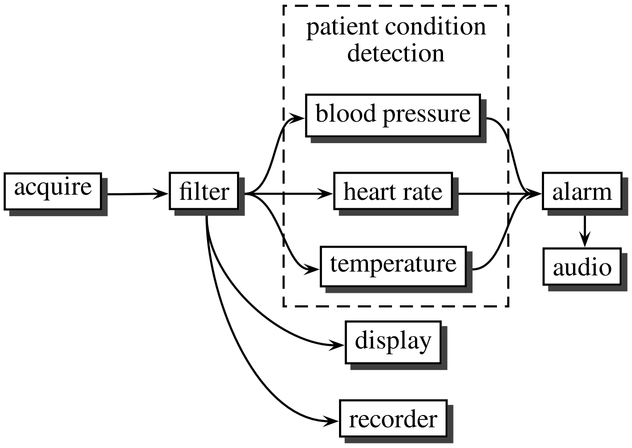

Figure 7: The patient monitoring system

Process	Period	Execution Times			
		(1)		(2)	
		min	max	min	max
acquire	20	1	1	1	1
filter	-	4	4	3	3
blood pressure	-	6	∞	2	2
heart rate	-	6	∞	2	6
temperature	-	6	∞	2	10
display	-	6	18	2	14
recorder	-	8	20	4	16
alarm	-	8	∞	2	2
audio	-	14	∞	6	∞

Some unexpected results can be seen in this table. The system is no longer schedulable. The *audio* process can execute for an unbounded amount of time. By comparing the two tables we see that the maximum execution times of most processes increased. But no additional load has been added to the system. In order to verify why this behavior was occurring we used the *counterexample* feature of the SMV model checking system [24]. A counterexample is an execution trace that violates a property specified. By expressing the property that the *audio* process would always finish execution, we were able to produce a counterexample which showed that this property was false. The execution trace revealed the following execution sequence leading to the problem:

acquire; filter; blood_pressure; alarm; heart_rate; alarm; temperature; alarm; display; recorder; acquire; filter; ...

We can see from the trace above that the problem is caused by the fact that *alarm* executes three times for the same instantiation of *acquire* when all detection processes find abnormalities. This causes an overload in the system making it unschedulable. The reason this did not happen before was that every time a detection process triggered the *alarm* process, it requested execution, but it would only execute after all detection processes executed. One execution responded to all alarm conditions. A simple solution to this problem is to lower the priority of *alarm* and change the design so that multiple alarms are handled correctly. The final priority order is: *acquire, filter, blood_pressure, heart_rate, temperature, alarm, display, recorder*, and *audio*. The results computed using this priority order showed that the system was schedulable.

The condition counting algorithms can also be used to analyze the behavior of the system. If the designer believes that the *alarm* process is being blocked by less important processes, he or she can use the condition counting algorithms to quantify this effect. For example, we can compute how much time is spent on the execution of the *display* or the *recorder* processes while *alarm* is requesting execution. The parameters of *mincount* and *maxcount* can be specified as follows. The initial state is the start of *alarm*, the final state is the end of execution of *alarm*, and the condition to be counted is the processor granted to either *display* or *recorder*. Using the first priority order presented, the time spent on *display* and *recorder* while *alarm* is blocked is 4ms. With the last priority order this time is zero, as expected.

The algorithms described in this paper allow us to analyze the medical monitoring example in many ways. Schedulability is determined by computing the response times of all processes. The reaction time to an event is computed in the same manner. We can determine the minimum and maximum latencies between the occurrence of an abnormal event and its recognition by the system (in this case by sounding the alarm). The algorithms also allow us to study how changes in the parameters affect global behavior. In this example we can see the impact that the priority order has on response times. This type of analysis can be very useful in validating the design of industrial real-time systems.

7 An Aircraft Controller

As another example of how our techniques can be applied in the verification of realistic real-time systems, this

System	Subsystem	Per.	Exec	%cpu	Pri
Display	status update	200	3	1.50	12
	keyset	200	1	0.50	16
	hook update	80	2	2.50	36
	graph. displ.	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
	track filter	25	2	8.00	84
NAV	nav update	50	8	16.00	56
	steer cmds.	200	3	1.50	24
Track	target update	100	5	5.00	32
Weapon	weapon prot.	200*	1	0.50	28
	weapon aim	50	3	6.00	64
	weapon rel.	200**	3	1.50	98
Dat Bus	poll device	40	1	2.50	68

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

Subsystem	dead line	Execution Times			
		preempt		no preempt	
		min	max	min	max
Weapon release	5	3	3	3	9
Radar track filter	25	2	5	2	10
Contact mgmt.	25	7	10	7	15
Data bus poll	40	1	11	1	14
Weapon aim	50	10	14	2	18
Radar target upd	50	12	19	12	19
NAV update	50	20	34	20	27
Display graphic	80	10	44	10	43
Display hook upd	80	14	46	14	47
Track target upd	100	26	51	26	51
Weapon protocol	200	1	21	3	46
NAV steer cmds.	200	35	85	36	74
Display store upd	200	36	95	37	97
Display keyset	200	37	96	38	98
Display status upd	200	40	99	41	101

section briefly presents the verification of an aircraft controller. A complete analysis of this example can be found in [8]. 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 [23].

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 include navigation, radar control, weapons and display. 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. The following table presents the subsystems being modelled, as well as their timing requirements. Concurrent processes are used to implement each subsystem. In order to enforce the different timing constraints of the processes, priority scheduling is used. Predictability is guaranteed by scheduling the processes using RMS [19, 22].

We have implemented this control system in the SMV language [24]. 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. Both a preemptive scheduler and a non-preemptive scheduler were implemented to analyze the effects of preemption in the response times. Schedulability was determined by computing response times of each process and checking that each process met its deadline. In this example the deadlines are the same as the periods (except for the weapon release subsystem). 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.

We can see from the table above that the process set is schedulable using preemptive scheduling. Notice however 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 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.

8 Conclusion

This paper presents a formal method to express and compute timing characteristics of real-time systems. A description of the system is first compiled into a state-transition graph represented using binary decision diagrams. Symbolic model checking algorithms compute the minimum and maximum lengths of paths between two state sets. In addition, an algorithm for computing the exact upper and lower bounds on the number of times a condition can hold on any path between two state sets is presented. Using these techniques we have verified two examples which model actual industrial applications. These examples demonstrate that our tools can handle applications of realistic size and be useful in the design process of industrial real-time systems.

The symbolic techniques employed have made our algorithms very efficient. BDDs provide a concise representation for the state-transition graph and for state sets. This representation allows us to handle examples of realistic complexity. State-transition graphs with 10^{30} states can be traversed in minutes.

Our method computes quantitative information that cannot be directly obtained using other approaches. The bounds computed by our algorithms allow us to make assertions about system performance rather than just about its correctness. Furthermore the versatility of our method is indicated by the fact that practically any real-time design can be represented. The only restriction imposed on the system being analyzed is that it be modeled as a state-transition graph.

Finally, our techniques can be used during the design process to evaluate design decisions. For example, in the medical monitoring system, inefficiencies were identified and the computed information led to suggestions for possible improvement. The model was modified to account for these changes. The analysis of the new system confirmed that the changes were indeed optimizations.

The examples analyzed in this work indicate that the information obtained by our method can be extremely useful in the development of real-time systems. We are confident that this method can be used successfully in improving the efficiency and reliability of real-time system design.

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, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27^{th} ACM/IEEE Design Automation Conference*, 1990.

[6] 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.

[7] 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.

[8] S. V. Campos, E. M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. Technical Report CMU-CS-94-147, Carnegie Mellon University, School of Computer Science, 1994.

[9] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*. Springer-Verlag, 1981. volume 131 of *Lecture Notes in Computer Science*.

[10] 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.

[11] 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, Apr 1993.

[12] E. M. Clarke, O. Grumberg, and D. E. Long. Verification tools for finite-state concurrent systems. In *REX '93 School/Workshop: A Decade of Concurrency*, Noordwijkerhout, The Netherlands, June 1993. to appear in Springer Lecture Notes in Computer Science.

[13] C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. *Formal Methods in System Design*, 1, 1992.

[14] P. J. Drongowski. Software architecture in realtime systems. In *IEEE Workshop on Real-Time Applications*, 1993.

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

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

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

[18] 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.

[19] 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.

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

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.