Computing Quantitative Characteristics of Finite State Real-Time Systems *

S. Campos, E. Clarke, W. Marrero, M. Minea
Carnegie Mellon University

H. Hiraishi
Kyoto Sangyo University

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 asserting 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 very useful in verifying real-time system designs.

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

1 Binary Decision Diagrams

Binary decision diagrams (BDDs) are a canonical representation for Boolean formulas [1]. A BDD is similar to a binary decision tree except that its structure is a directed acyclic graph rather than a tree. This allows

*This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597 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 and in part by the Wright Laboratory, Aeronautical Systems Center Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. 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.

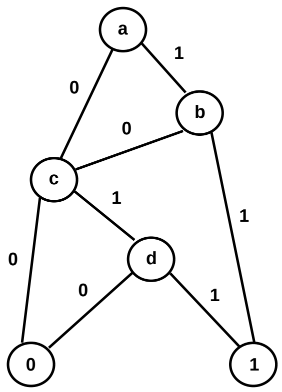

Figure 1: BDD for $(a \wedge b) \vee (c \wedge d)$

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

In [1], Bryant shows that given a variable ordering, the BDD for a formula is unique. Bryant 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 these formulas to be abitrary computation tree logic (CTL) [2] formulas which allow branching time operators as well as logical connectives, but for the sake of simplicity, this discussion is limited to Boolean formulas.

2 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 for 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 $i \geq 0$.

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

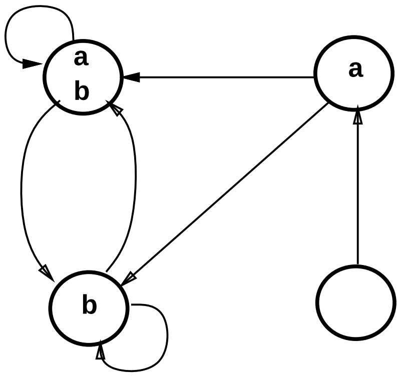

Figure 2: A sample state transition graph

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, $\langle s_1, s_2, s_3, \ldots, s_n \rangle$, the values of the state variables in state $\overline{s}$, 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 is given by the formula $N(\langle a, b \rangle, \langle a', b' \rangle) = (b \wedge b') \vee (a \wedge b') \vee (\neg a \wedge \neg b \wedge a' \wedge b')$.

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, in the paper 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 conjcuntion 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 conjunction 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.

3 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

```
proc lower (start, final)
i = 0;
R = start;
R' = T(R) ∪ R;
while (R' ≠ R ∧ R ∩ final = ∅) do
        i = i + 1;
        R = R';
        R' = T(R') ∪ R';
if (R ∩ final ≠ ∅)
        then return i;
        else return ∞;
```

Figure 3: Lower Bound Algorithm

algorithm returns infinity. The function $T(S)$ gives the set of states that are successors of some state in S (i.e. $T(S) = \{s' \mid N(s, s') \text{ holds for some } s \in S\}$). 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. 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 imples $i + 1 \leq L$ 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.

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

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 $R \cap final \neq \emptyset$, 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 tells us 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 gives 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 ∞.

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. $T^{-1}(S') = \{s \mid N(s, s')$ 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*.
- M is the number of states in a longest path starting 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 that 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 gives us 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 $R' = S_{i+1}$; therefore 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 $R' \cap$ *start* $= \emptyset$ giving us that $R \cap$ *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 $R' \cap$ *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 $S_{i+1} \subset$ *not_final*. The previous observation tells us that $R' \cap$ *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 ∞.

If $R' \cap$ *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 length of the longest path from a state in *start* to a state in *final* contains $i + 1$ states and has length i.

References

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

[2] 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.