Selective Quantitative Analysis and Interval Model Checking: Verifying Different Facets of a System*

Sérgio Campos[1] and Orna Grumberg[2]

[1] Carnegie Mellon University School of Computer Science, Pittsburgh, PA 15213, USA.
[2] The Technion Department of Computer Science, Haifa 32000, Israel.

Abstract. In this work we propose a verification methodology consisting of *selective quantitative timing analysis* and *interval model checking*. Our methods can aid not only in determining if a system works correctly, but also in understanding how *well* the system works. The selective quantitative algorithms compute minimum and maximum delays over a selected subset of system executions. A linear-time temporal logic (LTL) formula is used to select either infinite paths or finite intervals over which the computation is performed. We show how tableaux for LTL formulas can be used for selecting either paths or intervals and also for model checking formulas interpreted over paths or intervals.
To demonstrate the usefulness of our methods we have verified a complex and realistic distributed real-time system: Our tool has been able to analyze the system and to compute the response time of the various components. Moreover, we have been able to identify inefficiencies that caused the response time to increase significantly (about 50%). After changing the design we not only verified that the response time was lower, but were also able to determine the causes for the poor performance of the original model using interval model checking.

1 Introduction

This work presents a verification methodology that can provide both quantitative and qualitative analysis of systems. The analysis can aid not only in determining system correctness, but also in understanding how *well* the system works. The method consists of *selective quantitative timing analysis* and *interval model checking* and is based on two concepts: *quantitative timing analysis*, and *tableaux* for linear-time temporal logic.

In [8] we have shown how quantitative symbolic algorithms can be used to analyze the behavior of a system. Our method computes minimum and maximum delays between the occurrence of two events, as well as the number of times a specified condition occurs in such an interval. The timing correctness of a system can be evaluated by this method. Reaction time to important events can be computed as well as analyzing how the system behaves during the interval between event and response. In general, performance parameters can be analyzed using this technique.

* 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.

Typically, the quantitative analysis investigates *all* intervals between a set of initial states *start* and a set of final states *final*. In many cases, however, it is desirable to restrict the consideration to only execution paths that satisfy a certain condition. This can help in understanding how the system reacts to different conditions. For example, one common technique for achieving good performance is to optimize a design for the most common cases, while maintaining correctness for the uncommon ones. The designer can optimize response time by restricting system behavior to the most frequent cases. Correctness can then be checked by removing the restrictions.

In this work we use a formula of the linear-time temporal logic LTL to specify a set of paths selected to be verified. Quantitative analysis is then applied only to those paths along which the formula holds. We also extend the technique for cases in which a more precise analysis is needed, by requiring that the selecting formula be true exactly on the investigated interval and not just anywhere on the path.

To strengthen our verification methodology, we combine selective quantitative analysis with model checking. Traditionally, LTL model checking procedures [11, 19, 27] accept a structure that models the system, a set of initial states, and an LTL formula. The procedures determine whether the formula holds on all infinite paths of the structure starting on some initial state. In this work we extend the construction of [11] also for *interval model checking*, that is, checking a formula with respect to finite intervals.

Main Characteristics: Both interval model checking and selective quantitative analysis can be used to extract information related to specific "parts" of a system *without* changing the model. Similar information sometimes can be obtained by restricting the model to disable uninteresting behaviors, or by marking the interesting ones using observer modules. However, these techniques frequently modify system behavior, and consequently properties are checked on a model different than the original one, possibly hiding important errors, or introducing false ones.

Moreover, the fact that properties are verified over finite intervals, allows very different types of properties to be expressed. It is possible to check for "traditional" properties such as safety and liveness, but also to investigate system behavior in more detail. In the real-world not all possible execution sequences are equally interesting. Nor are all possible time intervals within a path. Understanding how the system reacts in different situations allows for a detailed analysis that can aid not only in determining if the system works, but also in understanding how *well* the system works.

Related Methods: There are several other approaches to the verification of timed systems. For example, dense time is modeled by [1, 2, 17, 23, 28]. Those methods model time very accurately. However, the state space of dense time models is infinite, and these tools rely on the construction of a finite quotient structure called region graph. This construction is extremely expensive, limiting the size of problems that can be handled. Dense time models seem to be better suited for smaller problems in which time accuracy must be very high. On the other hand, models such as the one proposed are well suited to model large complex systems in which the accuracy can be easily handled by choosing an appropriate time quantum, as can be seen by the example in this paper.

Discrete time is used by other tools such as [16, 29]. These tools, however, do not allow the quantitative analysis of systems as the proposed method. In [14] quantitative

analysis is implemented, but with a more limited scope. Dense time models allowing restricted quantitative analysis can be found in [17, 28].

Linear-time temporal logics interpreted over both infinite paths and finite intervals have been introduced in [20, 21]. However, they use tableau only for satisfiability and do not handle either quantitative analysis or interval model checking. Interval logics are also used in [25], but in a theorem proving context.

More important when comparing these methods, however, is the fact that these tools do not allow a selective verification of properties as the proposed method. They provide no natural way in which a subset of behaviors can be analyzed in isolation, not allowing as rich an analysis as the proposed method. The closest method to our selection of paths or intervals is the use of fairness constraints in model checking [13, 15, 22]. However, there a fairly restricted types of properties were used for selection, while we can handle any LTL formula. Moreover, only infinite paths can be selected in these works.

A Distributed Real-Time System: To demonstrate the usefulness of our method, we have applied it to a distributed real-time system of realistic complexity, derived from the example described in [26]. Real-time systems are used in many critical applications such as aircraft control or medical monitoring systems. Because of the consequences of failures in such systems, determining their correctness is a vital task.

Several features of this example make it an interesting target for our techniques. It is a system of realistic complexity, its components are existing systems and protocols executing a mixture of multimedia, traditional real-time and non-real time tasks. Also, the distributed nature of the system makes the interaction among its various components much richer. This also makes its analysis more difficult.

Our tools have been able to analyze the system and verify that the deadlines are met by the design. Moreover, we have been able to identify inefficiencies that caused the response time to increase significantly (about 50%). After changing the design we not only verified that the response time was lower, but were also able to determine the causes for the poor performance of the original model using interval model checking.

2 A tableau for LTL

Our specification language is a *linear-time temporal logic* called LTL [24]. The logic is used for two different purposes. One is to specify a property of the system that needs to be verified. The other is to specify a set of selected paths that will be verified. In both cases we use a *tableau* [19, 27, 11] for the formula.

We first give the syntax of LTL. Given a set of atomic propositions AP, LTL is defined inductively as follows. Every atomic proposition is an LTL formula. If f and g are LTL formulas then $\neg f$, $f \vee g$, $\mathbf{X}\, f$ and $f\ \mathbf{U}\ g$ are also LTL formulas.

The semantics of LTL is defined with respect to a labeled state transition graph called *Kripke Structure*. A Kripke structure $M = (S, R, L)$ has a finite set of states S, a transition relation $R \subseteq S \times S$, and a labeling function $L : S \to \mathcal{P}(AP)$, associating with each state the set of atomic propositions true in that state.

An infinite sequence $s_0, s_1, \ldots$ of states in S is a *path* in the structure M from a state s iff $s = s_0$ and for every $j \geq 0$, $(s_j, s_{j+1}) \in R$. A finite sequence $[s_0, \ldots, s_n]$

is an *interval* in a structure M from a state s iff $s = s_0$ and for every $0 \leq j < n$, $(s_j, s_{j+1}) \in R$. An interval may be a prefix of either a finite or an infinite path. Thus, s_n may or may not have successors in M. The size of interval $\sigma = [s_0, \ldots, s_n]$, denoted $|\sigma|$, is n. σ^j is defined iff $0 \leq j \leq n$ and it denotes the suffix of σ, starting at s_j.

For a formula f, a path π, and an interval σ, $M, \pi \models_{path} f$ means that f holds along path π in the Kripke structure M. $M, \sigma \models_{int} f$ means that f holds along interval σ in M. Given a set of initial states S_0, we say that $M, S_0 \models_{path} f$ iff for every path π from every state in S_0, $M, \pi \models_{path} f$. Given two sets of states $start$ and $final$, we say that $M, [start, final] \models_{int} f$ iff for every interval σ from some state in $start$ to some state in $final$, $M, \sigma \models_{int} f$. In this work whenever we refer to a path (an interval) that satisfies a formula, satisfaction is with respect to $\models_{path}$ ($\models_{int}$). The relation $\models_{path}$ is the standard satisfaction relation for LTL (see, for example, [11]). The relation $\models_{int}$ is identical to $\models_{path}$ for atomic propositions and boolean connectives. For temporal operators it is defined by (M is omitted if clear from the context):

4. $\sigma \models_{int} \mathbf{X} g_1 \quad \Leftrightarrow \quad |\sigma| > 0$ and $\sigma^1 \models_{int} g_1$.
5. $\sigma \models_{int} g_1 \mathbf{U} g_2 \Leftrightarrow \quad \exists k[0 \leq k \leq n \wedge \sigma^k \models_{int} g_2 \wedge \forall j[0 \leq j < k \rightarrow \sigma^j \models_{int} g_1]]$.

When writing LTL formulas, we use the abbreviations $\mathbf{F} f = true \mathbf{U} f$ and $\mathbf{G} f = \neg \mathbf{F} \neg f$. Note that, in the definition of $[s_0, \ldots s_n] \models f$ we do not consider successors of s_n (whether exist or not). This definition is meant to capture the notion of an interval satisfying a formula independently of its suffix (satisfaction is always defined independently of the prefix).

Let f be an LTL formula. We construct a Kripke structure $T(f)$, called the *tableau* for f, that contains all paths and intervals satisfying f. To identify paths in the tableau that satisfy f we will use *fairness constraints*. A *fairness constraint* for a structure M can be an arbitrary set of states in M, usually described by a formula of the logic. A path in M is said to be *fair* with respect to a set of fairness constraints if each constraint holds *infinitely often* along the path.

Let AP_f be the set of atomic propositions in f. $T(f) = (S_T, R_T, L_T)$ has AP_f as its set of atomic propositions. The set of tableau states is $S_T = \mathcal{P}(el(f))$, where $el(f)$ is the set of *elementary formulas* defined by:

- $el(p) = \{p\}$ if $p \in AP_f$
- $el(\neg g) = el(g)$
- $el(\mathbf{X} g) = \{\mathbf{X} g\} \cup el(g)$
- $el(g \vee h) = el(g) \cup el(h)$
- $el(g \mathbf{U} h) = \{\mathbf{X}(g \mathbf{U} h)\} \cup el(g) \cup el(h)$

Let $sat(f)$ be the set of states in the tableau that should satisfy f. It is defined by:

- $sat(g) = \{s \mid g \in s\}, g \in el(f)$
- $sat(\neg g) = \{s \mid s \notin sat(g)\}$
- $sat(g \vee h) = sat(g) \cup sat(h)$
- $sat(g \mathbf{U} h) = sat(h) \cup (sat(g) \cap sat(\mathbf{X}(g \mathbf{U} h)))$

The transition relation R_T is $R_T(s, s') = \bigwedge_{\mathbf{X} g \in el(f)} (s \in sat(\mathbf{X} g) \Leftrightarrow s' \in sat(g))$. Finally, the labeling function, $L_T(s) = s \cap AP_f$ and the set of fairness constraints for f, $Fair(f) = \{sat(\neg(g \mathbf{U} h) \vee h) \mid g \mathbf{U} h$ occurs in $f\}$.

The constructed tableau $T(f)$ includes every path and every interval which satisfies f. The following theorem characterizes those paths and intervals.

Theorem 2.1 *For every path π in $T(f)$, if π starts from a state $s \in sat(f)$ and π is fair for $Fair(f)$ then $T(f), \pi \models_{path} f$. Moreover, For every interval $\sigma = [t_0, \ldots, t_n]$ in $T(f)$, if $t_0 \in sat(f)$ and $t_n \in \mathcal{P}(AP_f)$ then $T(f), \sigma \models_{int} f$.*

In the algorithms presented later we will use the *product* $P = (S_P, R_P, L_P)$ of $T(f) = (S_T, R_T, L_T)$ with the verified structure $M = (S_M, R_M, L_M)$. We restrict the atomic propositions of f, AP_f to be a subset of AP:

- $S_P = \{(s,t) \mid s \in S_M, t \in S_T$ and $L_M(s) \cap AP_f = L_T(t)\}$.
- $R_P((s,t),(s',t'))$ iff $R_M(s,s')$ and $R_T(t,t')$.
- $L_P((s,t)) = L_T(s)$.

3 CTL Model Checking

CTL [4, 12] is a *branching-time temporal logic* that is similar to LTL except that each temporal operator is preceded by a *path quantifier* – either **E** standing for "there exists a path" or **A** standing for "for all paths". CTL is interpreted over a state in a Kripke structure. The path quantifiers are interpreted over the *infinite paths* starting at that state.

CTL *model checking* is the problem of finding the set of states in a Kripke structure where a given CTL formula is true. One approach for solving this problem is *symbolic* model checking using a representation called *binary decision diagram* (BDD) [5] for the transition relation of the structure. This representation is often very concise. We use the SMV model checking system [22] that takes a CTL formula f, and the BDD that represents the transition relation. SMV computes exactly those states of the system that satisfy the formula f. SMV can also handle model checking of a CTL formula with respect to a structure with fairness constraints. The path quantifiers in the CTL formula are then restricted to fair paths. The CTL model checking under given fairness constraints can also be performed using BDDs.

4 Quantitative Timing Analysis

Several methods have been proposed to verify timed systems, as has been discussed in the introduction. Typically, verifiers assume that timing constraints are given explicitly in some notation like temporal logic and determine if the system satisfies the constraint. In [8] we have described how to verify timing properties using algorithms that explicitly compute timing information as opposed to simply checking a formula. This section briefly describes that approach, which is later used in this work.

A Kripke structure is the model of the system in our method. Currently the system is specified in the SMV language [22]. The structure is represented symbolically using BDDs. It is then traversed using algorithms based on symbolic model checking techniques [6]. All computations are performed on states reachable from a predefined set of initial states. We also assume that the transition relation is total.

We consider first the algorithm that computes the minimum delay between two given events (figure 1). Let *start* and *final* be two nonempty sets of states, often given as formulas in the logic. The *minimum* algorithm returns the length of (i.e. number of edges

in) a shortest interval from a state in *start* to a state in *final*. If no such interval exists, the algorithm returns infinity. The function $T(S)$ gives the set of states that are successors of some state in S. The function T, the state sets I and I', and the operations of intersection and union can all be easily implemented using BDDs [6, 22]. The *minimum* 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 that state.

```
proc minimum (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 ∞;
```

```
proc maximum (start, final, not_final)
if (start ∩ (final ∪ not_final) = ∅)
      then return ∞;
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;
```

Fig. 1. Minimum and Maximum Delay Algorithms

The second algorithm returns the length of a longest interval 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. *not_final* represents the states that do not satisfy *final* (except in the interval selective case, see below).

The initial conditional is only used when computing properties over intervals. As will be seen later, in this case *not_final* correspond to states not in *final*, but which eventually lead to *final*. Therefore, if no starting state is in *final*, or leads to *final*, the algorithm returns infinity.

Informally, the algorithm computes at stage i the set R' of all states at the beginning of an interval of size i, all contained in *not_final*. The algorithm stops in one of two cases. Either R' does not contain states from *start* at stage i. Since it contained states from *start* at stage $i-1$, the size of the longest interval in *not_final* from a state in *start* is $i-1$. Since the transition relation is total, this interval has a continuation to a state outside *not_final*, i.e. to a state in *final*. Thus, there is an interval of length i from *start* to *final* and the algorithm returns i. In the other case, a fixpoint is reached meaning that there is an infinite path within *not_final* from a state in *start*. The algorithm in this case returns infinity. Both minimum and maximum algorithms are proven correct in [8].

5 The Proposed Method

5.1 Selective Quantitative Analysis — Over Paths

Given two sets of states *start* and *final* in M and an LTL formula f, we compute the lengths of a shortest interval and a longest interval from a state in *start* to a state in

$final$ along paths from $start$ that satisfy f. The formula f is interpreted over infinite paths and is used to select the paths over which the computation is performed. The *minimum* and *maximum* algorithms with path selection are:

1. Construct the tableau for f, $T(f)$.
2. Construct the product P of $T(f)$ and M.
3. Use the model checking system SMV on P to identify the set of states *fair* in P, where a state $(s,t) \in S_P$ ($s \in M$, $t \in T(f)$) is in *fair* iff t is the beginning of a path which is fair with respect to *Fair(f)*.
4. Construct P', the restriction of P to the state set *fair*. $P' = (S'_P, R'_P, L'_P)$ is defined by: $S'_P = fair$, $R'_P = R_P \cap (S'_P \times S'_P)$ and for every $s \in fair$, $L'_P(s) = L_P(s)$.
5. Apply $minimum(st, fn)$ and $maximum(st, fn, not\_fn)$ to P', with $st = (start \times sat(f)) \cap fair$, $fn = (final \times S_T) \cap fair$, and $not\_fn = fair - fn$.

The algorithms work correctly because P contains all paths of M that are also paths of $T(f)$ (proof in the full paper). P' is restricted to the fair paths of $T(f)$. Thus, every path in P' from $(start \times sat(f)) \cap S'_P$ satisfies f. Consequently, applying the algorithms to P' from $(start \times sat(f)) \cap S'_P$ to $(final \times S_T) \cap S'_P$ over states in $fair$ gives the desired results.

As mentioned before, in order to work correctly, the algorithm *maximum* must work on a structure with a total transition relation. The transition relation of P is not necessarily total. However, the transition relation of P' is total since every state in $fair$ is the beginning of some infinite (fair) path.

We have applied the method in the analysis of the PCI Local Bus [10], where it has been used to limit the number of transaction aborts being considered.

5.2 Selective Quantitative Analysis — Over Intervals

Given two sets of states $start$ and $final$ and an LTL formula f, we compute the lengths of a shortest and a longest intervals from a state in $start$ to a state in $final$ such that f holds along the interval. Here the formula f is interpreted over intervals and we consider only the intervals between $start$ and $final$ that satisfy f. We will use a special formula *prop* to identify the set of tableau states that contain only atomic propositions.

$$prop = \{s \in S_T | s \in \mathcal{P}(AP_f)\}.$$

The formula *prop* is a set of states in $T(f)$. We extend *prop* to $prop_p$, which is the corresponding set of states in P. The formula $final_p$ is the similar extension of $final$:
- $prop_p = \{(s,t) \in S_P \mid s \in S_M, t \in prop\}$
- $final_p = \{(s,t) \in S_P \mid s \in final, t \in S_T\}$

We will also use a CTL formula $\mathcal{C}$ to identify the set of states over which the *maximum* algorithm is computed.

$$\mathcal{C} = \neg final_p \wedge \mathbf{E}[\neg final_p \ \mathbf{U} \ (prop_p \wedge \mathbf{EF}\ final_p)].$$

States in $\mathcal{C}$ lead to states that are endpoints of intervals satisfying f (states in $prop_p$, see theorem 2.1), and then lead to states in $final_p$. The requirement that $final_p$ does not hold until $prop_p$ is needed because an interval ending in $final_p$ without going through $prop_p$ does not satisfy f.

The *minimum* and *maximum* algorithms with interval selection are:

1. Construct the tableau for f, $T(f)$.
2. Construct the product P of $T(f)$ and M.
3. Use the model checking system SMV on P to identify the set of states that satisfy the CTL formula $\mathcal{C}$.
4. Let $st = (start \times sat(f)) \cap S_P$ and let $fn = (final \times prop) \cap S_P$. The algorithms $minimum(st, fn)$ and $maximum(st, fn, \mathcal{C})$ when applied to P will return the length of the shortest and longest intervals, respectively, between $start$ and $final$ that satisfy f.

The correctness of the algorithm relies on the fact that P contains all intervals that are both in $T(f)$ and M. Moreover, intervals of $T(f)$ from $sat(f)$ to $prop$ satisfy f. Thus, the algorithms compute shortest and longest lengths over intervals from $start$ to $final$ that satisfy f. The proof can be found in the full paper.

When the *maximum* algorithm is computed over the set *not_final* of states not in *final*, it is necessary to require that the transition relation of the structure is total in order to guarantee that the computed intervals terminate at a state in *final*. Here the *maximum* algorithm is computed over the set of states satisfying the formula $\mathcal{C}$. This guarantees that the computed intervals terminate at *final* without the need to require that the transition relation is total.

5.3 Interval Model Checking

Given a structure M and two set of states $start$ and $final$, we say that an interval $\sigma = [s_0, \ldots, s_n]$ from a state in $start$ to a state in $final$ is *pure* iff for all $0 < i < n$, s_i is neither in $start$ nor in $final$.

Given a structure M, two sets of states $start$ and $final$, and an LTL formula f, the *interval model checking* is the problem of checking whether the formula f, interpreted over intervals, is true of all pure intervals between $start$ and $final$ in M.

Interval model checking is useful in verifying *periodic* behavior of a system. A typical example is a behavior occurs in a transaction on a bus. If we want to verify that a certain sequence of events, described by an LTL formula f, occurs in a transaction we can define $start$ to be the event that starts the transaction and $final$ to be the event that terminates the transaction. Interval model checking will verify that f holds on all intervals between $start$ and $final$.

Let M, $start$, $final$, and f be as above. The algorithm given below determines the interval model checking problem using the algorithm *minimum* of figure 1.

1. Construct the tableau for $\neg f$, $T(\neg f)$.
2. Compute the product P of $T(\neg f)$ and M.
3. Apply the algorithm $minimum(st, fn)$ to P with $st = (start \times sat(\neg f)) \cap S_P$ and $fn = (final \times prop) \cap S_P$.
4. If the *minimum* is ∞ then there is no pure interval from $start$ to $final$ that satisfies $\neg f$. Thus, every such interval satisfies f. If *minimum* returns some value k, then the interval found by *minimum* can serve as a counterexample to the checked property.

6 A Distributed Real-Time System

In this section we analyze a distributed real-time system using the techniques presented in this paper. This is a complex and realistic application, its components are existing systems and protocols that are actually used in many real situations. The example consists of three main components, a FDDI network, a multiprocessor connected to this network and one of the processors in the multiprocessor, the control processor [26].

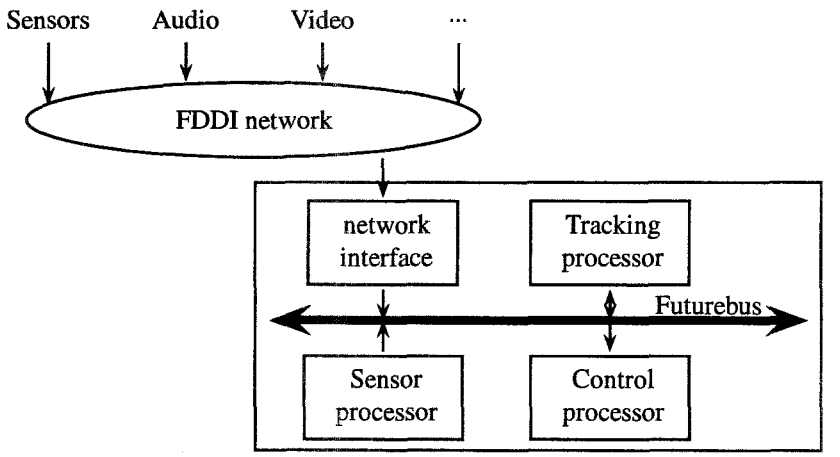

Fig. 2. System Architecture

The FDDI network is a 100Mb/s local/metropolitan area network that uses a token ring topology [3]. There are several stations connected to the network in the system. They generate multimedia and sensor data sent to the control processor, as well as additional traffic inside the network. The traffic in the network has been modeled as proposed in [26]. At every 16 time units the stations utilize the network as follows: *video* station, 6 units; *audio* station, 1 unit; and remainder network traffic, 8 units.

In the multiprocessor, four active processors are connected through a Futurebus+ [18]. The first is the *network interface*, it receives data from the network and sends it to the *control* processor. The network interface uses the bus for 7ms at each time. A *sensor* processor reads data from sensors every 40ms. It buffers this data and sends it once every four readings to the tracking processor. The *tracking* processor processes this data and sends it to the control processor. Both sensor and tracking data use the bus for 3ms each. The deadline for sensor data to be processed is 785ms. Access to the bus is granted using priority scheduling. Priorities are assigned according to the rule: processors with shorter periods have higher priority.

In the control processor there are several periodic tasks. The timing requirements for these tasks can be seen in figure 3. Priority scheduling is also used in the control processor, using the same priority assignment rule. Two tasks in the control processor have special functions, τ_3 processes sensor data, and τ_5 processes multimedia data.

Each of the components of the system (FDDI, network and control processor) has been implemented separately. No data is actually exchanged between the components in the model. Data has been abstracted out of the model, because data dependencies would

Process	τ_1	τ_2	τ_3	τ_4	τ_5
Period	100	150	160	300	100
Exec. time	5	78	30	10	3

Fig. 3. Timing requirements for tasks in the control processor (times in ms)

significantly increase the size of the model and the complexity of verification. However, while simplifying verification, abstractions can also introduce invalid execution sequences. The constraints imposed by data dependencies significantly reduce the number of reachable execution sequences. In an abstract model such dependencies do not exist. We have used selective quantitative analysis to ensure that only execution sequences that are valid (and all such sequences) have been considered during verification.

Using this model we have checked the deadline between a sensor reading in the sensor processor and the processing of this data by τ_3 in the control processor. This deadline is 785ms. Ideally, we would like to compute these time bounds using `MIN{MAX}[sensor_observation, t3.finish]`. However, since in our model there is no synchronization between tasks, this would consider intervals in which `t3` finishes executing just after `sensor`, without going through `track`. To identify the valid intervals in the model, we must consider only intervals that satisfy the constraint:

```
F(sensor.finish & F(track.start & F(track.finish & F t3.start)))
```

This formula guarantees that the correct ordering of events is maintained during verification. We have computed the time between sensor observation and τ_3 processing to be in the interval [197, 563], well within the deadline. However, by looking into the design we noticed a potential source for inefficiencies in the Futurebus. Using standard model checking techniques we then printed a counterexample for the longest response time. It confirmed our speculations.

In this system both sensor and tracking processors access the bus periodically, sending data every 160ms. In the counterexample, however, data required two periods of 160ms to reach the control processor. It was sent by the sensor processor to the tracking processor, but this processor would only send it to the control processor in the next period. Further investigation of the model showed that this was caused by the priority order in which processors accessed the bus. The tracking processor had a higher priority than the sensor processor. This means that when the sensor processor sends data to the tracking processor, it had already used the bus for this period, and would only request access again in 160ms. We modified the design by changing the priorities, and the response time became [37, 403], an improvement of almost 50%.

We have been also able to compare the performance of both designs using interval model checking. We have analyzed the behavior of the system between the time the sensor produces data until the time the tracking processor processes it. Bus utilization is inefficient in this interval if the bus is idle or a lower priority process is executing.

Using interval model checking we have been able to check the LTL formula `G !(bus_idle | bus_granted = lower_priority)` on the intervals between sensor finishing sending data and tracking sending its data to the control processor. The original design showed the existence of priority inversion, as expected. In the modified design, on the other hand, the formula above is true in all intervals under consideration. Notice that the formula is clearly false outside these intervals. This shows that the modified design is optimal with respect to the prioritized utilization of the bus.

The modified design has a better response time, and is clearly preferred in this application. But in other applications this might not be true. There might be cases, for example, in which the tracking processor sends data to the sensor processor. In those cases the modified design is worse than the original one. This again shows how selective quantitative analysis and interval model checking can be used to analyze the different facets of a system. The designer can choose to optimize the behavior of a critical application, even if at the expense of a less critical one.

7 Conclusion

In this paper we have described a method that can produce both quantitative and qualitative information about the behavior of a system. Quantitative analysis and model checking can be performed on state-transition graphs representing the system to be verified. Moreover, the user can specify a subset of execution sequences satisfying a given property using an LTL formula, and verification is performed only on those paths (or intervals) that satisfy that property. The results produced can be used not only to determine the correctness of the system, but also to analyze (and optimize) its performance.

We have used this method to analyze a real-time distributed system. This example shows how the proposed method can assist in understanding the behavior of complex systems. We have been able not only to check properties of the whole system, but also to analyze specific execution sequences of interest. This allowed us to uncover subtleties about the application that might have been very difficult to discover otherwise. We believe that this method can be of great use in analyzing and understanding other complex systems, as it has been in analyzing this one.

Acknowledgment

The authors would like to thank Edmund Clarke for the original idea of combining LTL model checking and quantitative analysis.

References

1. R. Alur, C. Courcourbetis, and D. Dill. Model-checking for real-time systems. In *Symposium 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. ANSI Std. *FDDI Token Ring Media Access Control*, s3t95/83-16 edition, 1986.
4. M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
5. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
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 *Symposium on Logic in Computer Science*, 1990.
7. S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. In *ACM Workshop on Languages Compilers and Tools for Real-Time Systems*, 1995.

8. S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Real-Time Systems Symposium*, 1994.
9. S. V. Campos, E. M. Clarke, W. Marrero, and M. Minea. Timing analysis of industrial real-time systems. In *Workshop on Industrial-strength Formal specification Techniques*, 1995.
10. S. Campos, E. Clarke, W. Marrero, and M. Minea. Verifying the performance of the pci local bus using symbolic techniques. In *International Conference on Computer Design*, 1995.
11. E. Clarke, O. Grumberg, and H. Hamaguchi. Another look at ltl model checking. In D. Dill, editor, *proceedings of the Sixth Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 818, pages 415–427. Springer-Verlag, 1994.
12. 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. *Lecture Notes in Computer Science*, volume 131.
13. 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.
14. P. C. Clements, C. L. Heitmeyer, B. G. Labaw, and A. T. Rose. MT: a toolset for specifying and analyzing real-time systems. In *IEEE Real-Time Systems Symposium*, 1993.
15. E.A. Emerson and Chin Laung Lei. Modalities for model checking: Branching time strikes back. *Twelfth Symposium on Principles of Programming Languages*, January 1985.
16. A. N. Fredette and R. Cleaveland. RTSL: a language for real-time schedulability analysis. In *IEEE Real-Time Systems Symposium*, 1993.
17. T. A. Henzinger, P. H. Ho, and H. Wong-Toi. HyTech: the next generation. In *IEEE Real-Time Systems Symposium*, 1995.
18. IEEE Standard Board and American National Standards Institute. *Standard Backplane Bus Specification for Multiprocessor Architectures: Futurebus+*, ansi/ieee std 896.1 edition, 1990.
19. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12^{th} Conference on Principle of Programming languages*, 1985.
20. O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proc. Conf. Logics of Programs*, Lecture Notes in Computer Science 193, pages 196–218. Springer-Verlag, 1985.
21. Z. Manna and A. Pnueli. The anchored version of the temporal framework. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Lecture Notes in Computer Science 354, pages 201–284. Springer-Verlag, 1989.
22. K. L. McMillan. *Symbolic model checking — an approach to the state explosion problem*. PhD thesis, SCS, Carnegie Mellon University, 1992.
23. 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.
24. A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the eighteenth conference on Foundation of Computer Science*, 1977.
25. Y. Ramakrishna, P. Melliar-Smith, L. Moser, L. Dillon, and G. Kutty. Really visual temporal reasoning. In *IEEE Real-Time Systems Symposium*, 1993.
26. L. Sha, R. Rajkumar, and S. Sathaye. Generalized rate-monotonic scheduling theory: a framework for developing real-time systems. In *Proceedings of the IEEE*, Jan 1994.
27. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, 1986.
28. F. Wang. Timing behavior analysis for real-time systems. In *Proceedings of the Tenth Symposium on Logic in Computer Science*, 1995.
29. J. Yang, A. Mok, and F. Wang. Symbolic model checking for event-driven real-time systems. In *IEEE Real-Time Systems Symposium*, 1993.