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

Sérgio Campos
Carnegie Mellon University*

Edmund M. Clarke
Carnegie Mellon University†

Orna Grumberg
The Technion‡

January 4, 1996

Abstract

In this work we propose a verification methodology consisting of *selective quantitative 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. We use a formula of the linear-time temporal logic LTL in order to select either infinite paths or finite intervals over which the computation is performed. We therefore define two semantics for LTL – over infinite paths and over finite intervals. We show how tableaux for LTL formulas can be used for selecting either paths or intervals and can also be used for model checking formulas interpreted over paths or intervals.

We have implemented a tool based on our techniques. To demonstrate the usefulness of our methods we verified a complex distributed real-time system. 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.

Out tool were 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.

*Address: School of Computer Science, Pittsburgh, PA 15213, USA. Email: `Sergio.Campos@cs.cmu.edu`
†Address: School of Computer Science, Pittsburgh, PA 15213, USA. Email: `Edmund.Clarke@cs.cmu.edu`
‡Address: Department of Computer Science, Haifa 32000, Israel. Email: `orna@cs.technion.ac.il`

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 if the system works correctly, but also in understanding how *well* the system works. The suggested methodology consists of *selective quantitative analysis* and *interval model checking* and it is based on two concepts – *quantitative analysis*, and *tableaux* for linear-time temporal logic.

In [10] it has been proposed how quantitative symbolic algorithms can be used to analyze the model of a system. The technique suggested there 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. The schedulability of a real-time task set can be determined by computing response times for all processes. Reaction time to important events can also be computed in the same manner. In general, performance parameters can be analyzed using this technique.

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. Being able to select the execution paths considered during verification, 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 frequent cases, while maintaining correctness for the infrequent ones. If the designer can restrict system behavior to only the most common cases, he can optimize response time. Later he can remove the restrictions and check the correctness of the system in all cases.

In this work we suggest how to apply selective quantitative analysis. We use a formula of the linear-time temporal logic LTL in order to specify the paths selected to be verified. Quantitative analysis is then applied only to those paths along which the formula holds.

Sometimes a more precise analysis is needed, requiring that the selecting formula is true exactly on the investigated interval and not just anywhere on the path. Thus, if the LTL formula is Ga (meaning "a holds globally") then an interval is selected only if all states on the interval satisfy a. If the formula is Fa (meaning "a holds eventually") then there must be a state on the interval that satisfies a.

To maintain selecting of either infinite paths or finite intervals we will consider two semantics for the logic LTL – over infinite paths and over finite intervals.

To strengthen our verification methodology, we combine the selective quantitative analysis with model checking techniques. Traditionally, LTL model checking procedures [20, 11] accept a structure that models the system, a set of designated states, and an LTL formula. The procedures determine whether the formula holds on all infinite paths of the structure that start from some designated 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. We use *Tableaux* for LTL formulas as the main tool for both selecting and model checking.

Using tableaux for LTL formulas: A tableau for an LTL formula f is a structure that includes all possible paths that satisfy the formula. In order to verify that all paths of a system have some property f, we construct the tableau for the negation of f, $\neg f$, and check that no path of the system is included in the tableau for $\neg f$. This is done by constructing the *intersection* of the system and the tableau and checking that the intersection includes no path.

In order to select the set of all paths that satisfy a formula f, we construct the tableau for f and intersect it with the system. Verification is then applied to the intersection structure.

We show how the same tableau can also represent all finite intervals satisfying a formula. Thus, intersecting with the tableau for $\neg f$ can also be used to check that all intervals in the system satisfy f and intersecting with the tableau for f can be used to select the intervals satisfying f.

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. Also, such methods are usually *ad hoc*; the class of execution sequences that can be analyzed cannot be characterized in a straightforward way. They are also more difficult to implement and error-prone.

An important characteristic of the method proposed is that it counts the number of computation steps between events, or the number of occurrences of events in an interval. Because of this it finds application in synchronous systems in general, such as computer circuits and protocols. Another area in which the method has been successfully used is in the verification of real-time systems, as will be seen in the example verified in this paper. These systems are inherently asynchronous, and would not seem to be appropriate for our method. However, they are subject to tight timing constraints, which are difficult to satisfy in an asynchronous design. For this reason designers of real-time systems often significantly reduce asynchronism in their systems to ensure predictability. In fact, several real-time systems we have analyzed are even more synchronous than traditional circuits, and have been successfully verified using techniques such as the one proposed [10, 9, 8].

Another advantage of this approach is that it is amenable to symbolic implementations using bdds [5]. This makes it possible to verify systems with extremely large state spaces, allowing realistic and interesting problems to be handled.

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, 25, 17]. Those methods provide a very accurate notion of passage of time. However, the state space of dense time models is infinite, and these verification 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.

Discrete time is used by other tools such as [16, 28]. The tool described in [28] also uses symbolic algorithms using BDDs. 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.

Analytical methods for analyzing real-time systems also exist, such as the rate-monotonic scheduling theory [22, 19, 26]. In this method a real-time system is characterized by a set of periodic tasks, each having a period and an execution time. Assumptions about system behavior are made (such as no task preempts itself), and if these assumptions are satisfied, simple formulas determine the schedulability of the system. The rate-monotonic theory algorithms have much simpler complexity than the other verification methods discussed, but they also generate more restricted information.

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.

Linear-time temporal logics interpreted over both infinite paths and finite intervals have been introduced in [21, 23]. However, they use tableau only for satisfiability and did not handle either quantitative analysis or interval model checking.

The closest method to our selection of paths or intervals is the use of fairness constraints in model checking [13, 24, 15]. 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 [27]. 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.

The system consists of three major components, the first being an FDDI network to which are connected audio and video data sources. The network is connected to a multiprocessor, where other data is generated. Finally the third component is one of the processors of the multiprocessor, the control processor, which receives the audio/video signals from the network, as well as data from other processors in the system.

The specification determines deadlines between data sources (audio/video on the network and sensor data on the multiprocessor), and their processing in the control processor. However, verifying that these deadlines are met using standard techniques is made more difficult because of the distributed nature of the problem. Analytical methods such as the rate monotonic scheduling must impose restrictions on the system, for example, intermediate deadlines [27]. The complex interaction between the various components of the system also makes its analysis using continuous time models unmanageable.

Our tools, on the other hand, were 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. The final model uses approximately 5000 bdd nodes, and has about 10^{21} states. Verification time varied from seconds for the simplest properties to several minutes for the most complex ones using a pentium based workstation.

The remainder of this paper is organized as follows. In Section 2 we define the logic LTL, give two semantics for the logic, and construct a tableau for LTL formulas that suits both semantics. Section 3 shortly describes CTL model checking and quantitative analysis. In Section 4 selective quantitative analysis and interval model checking are described. Section 5 concludes with the verification of a complex example using a combination of our new techniques.

2 A tableau for LTL

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

We first give the syntax of LTL. Given a set of atomic propositions AP, the *linear-time temporal logic* LTL is defined inductively as follows. Every atomic proposition is an LTL formula. Moreover, 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, $R \subseteq S \times S$ is the transition relation, and $L : S \rightarrow \mathcal{P}(AP)$ is the labeling function that associates 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 $j \geq 0$, $(s_j, s_{j+1}) \in R$. Let $\pi = s_0, s_1, \ldots$ be a path, we use π^j to denote the *suffix* of π starting at s_j. 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. Let $\sigma = [s_0, \ldots, s_n]$ be an interval, then the size of σ, 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 designated 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 designated sets of states $start$ and $final$, we say that $M, [start, final] \models_{path} f$ iff for every interval σ from some state in $start$ to some state in $final$, $M, \sigma \models_{int} f$. Note that this definition does not require that intervals will be disjoint. Unless otherwise stated, overlapping intervals are allowed.

The relation $\models_{path}$ is defined inductively as follows (the structure M is omitted whenever clear from the context).

1. $\pi \models_{path} p \quad \Leftrightarrow \quad p \in L(s_0)$, for $p \in AP$.
2. $\pi \models_{path} \neg f_1 \quad \Leftrightarrow \quad \pi \not\models_{path} f_1$.
3. $\pi \models_{path} f_1 \vee f_2 \quad \Leftrightarrow \quad \pi \models_{path} f_1$ or $\pi \models_{path} f_2$.
4. $\pi \models_{path} \mathbf{X}\, g_1 \quad \Leftrightarrow \quad \pi^1 \models_{path} g_1$.
5. $\pi \models_{path} g_1\ \mathbf{U}\ g_2 \quad \Leftrightarrow \quad$ there exists a $k \geq 0$ such that $\pi^k \models_{path} g_2$ and for all $0 \leq j < k$, $\pi^j \models_{path} g_1$.

The relation $\models_{int}$ is identical to $\models_{path}$ for atomic propositions and boolean connectives. For temporal operators it is defined by

4. $\sigma \models_{int} \mathbf{X}\, g_1 \quad \Leftrightarrow \quad |\sigma| > 0$ and $\sigma^1 \models_{int} g_1$.
5. $\sigma \models_{int} g_1 \,\mathbf{U}\, g_2 \quad \Leftrightarrow \quad$ there exists a $0 \leq k \leq n$ such that $\sigma^k \models_{int} g_2$ and for all $0 \leq j < k$, $\sigma^j \models_{int} g_1$.

The following abbreviations are used in writing LTL formulas:

• $f \wedge g \equiv \neg(\neg f \vee \neg g)$ • $\mathbf{F}\, f \equiv true \,\mathbf{U}\, f$ • $\mathbf{G}\, f \equiv \neg\, \mathbf{F} \neg f$.

In the sequel, whenever we refer to a path that satisfies a formula, the satisfaction is with respect to $\models_{path}$. Whenever an interval is considered the satisfaction is with respect to $\models_{int}$.

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).

It is also important to notice that LTL formulas might have quite a different meaning when interpreted over paths or over intervals. For instance, a path will satisfy the formula $\mathbf{G}\,\mathbf{F}\, a$ iff a holds infinitely often along the path. On the other hand, an interval will satisfy this formula iff the last state of the interval satisfies a. Furthermore, while the formulas $\neg\, \mathbf{X}\, a$ and $\mathbf{X} \neg a$ are equivalent over paths, these formulas are not equivalent over intervals. To see this, consider an interval $[s_0]$ of size 0. $[s_0] \models_{int} \neg\, \mathbf{X}\, a$ but $[s_0] \not\models_{int} \mathbf{X} \neg a$.

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. The tableau described below is based on the construction given in [11]. There, the tableau was used for checking that the LTL formula is true for all paths of a given Kripke structure. Here we will use the tableau for three purposes:

1. Selecting the set of paths of a structure that satisfy f and computing minimum and maximum delays over those paths;

2. Selecting the set of intervals of a structure that satisfy f and computing minimum and maximum delays over those intervals;

3. Checking that a specified set of intervals of a structure satisfy f.

We first introduce the notion of *fairness constraints*, needed for some of the tableau applications. 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.

We now give an informal description of the tableau. A state of the tableau is a set of formulas, intended to be true along all paths in the tableau that start with that state. The transition relation of the tableau guarantees the satisfaction of all formulas except formulas of the form $f \,\mathbf{U}\, g$. If $f \,\mathbf{U}\, g$ is included in a state, then the tableau construction guarantees that f is true as long as g is not true. In the case of LTL over paths, fairness constraints are required in order to identify those infinite paths along which g will eventually be true. For LTL over finite intervals, it is sufficient to consider those intervals that have a final state that does not contain any formula of the form $\mathbf{X}\, g$. Intuitively, $\mathbf{X}\, g$ formulas can be viewed as transferring to next states the requirements that are necessary for the satisfaction of f and are not yet fulfilled. Thus a state that contains no formula of the form $\mathbf{X}\, g$ indicates that all necessary requirements have already been fulfilled.

We next describe the construction of the tableau $T(f)$ in detail. Let AP_f be the set of atomic propositions in f. The tableau associated with f is a structure $T(f) = (S_T, R_T, L_T)$ with AP_f as its set of atomic propositions. Each state in the tableau is a set of *elementary* formulas obtained from f. The set of elementary subformulas of f is denoted by $el(f)$ and is defined recursively as follows:

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

- $el(g \mathbf{U} h) = \{\mathbf{X}(g \mathbf{U} h)\} \cup el(g) \cup el(h)$.

Thus, the set of states S_T of the tableau is $\mathcal{P}(el(f))$. The labeling function L_T is defined so that each state is labeled by the set of atomic propositions contained in the state.

In order to construct the transition relation R_T, we need an additional function sat that associates with each elementary subformula g of f a set of states in S_T. Intuitively, $sat(g)$ will be the set of states that satisfy g.

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

We want the transition relation to have the property that for every elementary formula $\mathbf{X}\, g$ of f, $\mathbf{X}\, g$ is in a state iff $\mathbf{X}\, g$ is true in that state. Clearly, if $\mathbf{X}\, g$ is in some state s, then all the successors of s should satisfy g. Moreover, if $\mathbf{X}\, g$ is not in s, then no successor of s should satisfy g. Thus, the definition for R_T is

$$R_T(s, s') = \bigwedge_{\mathbf{X} g \in el(f)} s \in sat(\mathbf{X}\, g) \Leftrightarrow s' \in sat(g).$$

Unfortunately, the definition of R_T does not guarantee that *eventuality* properties are fulfilled. Consequently, an additional condition is necessary in order to identify those paths and intervals along which f holds. In order to identify the paths along which f holds we define a set of *fairness constraints*, $Fair \subseteq \mathcal{P}(S_T)$,

$$Fair(f) = \{sat(\neg(g \mathbf{U} h) \vee h) \mid g \mathbf{U} h \text{ occurs in } f\}.$$

Theorem 2.1 *Let $T(f)$ be the tableau for f.*

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 $T(f), \pi \models_{path} f$.*
2. *For every interval $\sigma = [t_0, \ldots, t_n]$ in $T(f)$, if $t_0 \in sat(f)$ and $t_n \in \mathcal{P}(AP)$ then $T(f), \sigma \models_{int} f$.*

The following theorem makes precise the intuitive claim that $T(f)$ includes every path and every interval which satisfies f. In order to state this property, we must introduce some new notation. A path $\pi = t_0, t_1, \ldots$ in $T(f)$ *corresponds* to a path $\pi' = s_0, s_1, \ldots$ in a structure M iff for every $i \geq 0$, $L(s_i) \cap AP_f = L_T(t_i)$. An interval $\sigma = [t_0, \ldots, t_n]$ in $T(f)$ corresponds to an interval $\sigma' = [s_0, \ldots, s_n]$ in M iff for every $0 \leq i \leq n$, $L(s_i) \cap AP_f = L_T(t_i)$.

Theorem 2.2 *Let $T(f)$ be the tableau for the formula f and let M be a Kripke structure.*

1. *If π' is a path of M such that $M, \pi' \models_{path} f$ then there is a path π in $T(f)$ such that (i) π corresponds to π', (ii) π starts with a state in $sat(f)$ and (iii) π is a fair path with respect to $Fair(f)$.*
2. *If σ' is an interval in M such that $M, \sigma' \models_{int} f$ then there is a interval σ in $T(f)$ such that (i) σ corresponds to σ', (ii) σ starts with a state in $sat(f)$ and (iii) the last state of σ is in $\mathcal{P}(AP)$.*

Next, we want to compute the *product* $P = (S, R, L)$ (also called the intersection) of the tableau $T(f) = (S_T, R_T, L_T)$ and the Kripke structure $M = (S_M, R_M, L_M)$.

- $S = \{(s, s') \mid s \in S_T, s' \in S_M \text{ and } L_M(s') \cap AP_f = L_T(s)\}$.
- $R((s, s'), (t, t'))$ iff $R_T(s, t)$ and $R_M(s', t')$.
- $L((s, s')) = L_T(s)$.

We extend the function sat to be defined over the set of states of the product P by $(s, s') \in sat(g)$ iff $s \in sat(g)$.

Lemma 1 *$\tau'' = (s_0, s'_0), (s_1, s'_1), \ldots$ is a path or an interval in P with $L_P((s_i, s'_i)) = L_T(s_i)$ for $i \geq 0$ if and only if there exist $\tau = s_0, s_1, \ldots$ in $T(f)$, and $\tau' = s'_0, s'_1, \ldots$ in M with $L_T(s_i) = L_M(s_i) \cap AP_f$ for $i \geq 0$.*

3 Known Verification Techniques

The variety of verification techniques that we develop in the next section are based on the tableau, as described in the previous section, and in addition on two verification techniques: CTL model checking and quantitative analysis.

3.1 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 $\mathbf{E}$ standing for "there exists a path" or $\mathbf{A}$ standing for "for all paths". CTL is interpreted over a state in a Kripke structure. The path quantifiers are interpreted over the *infinite paths* of the structure that start 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 a *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 [24] that takes a CTL formula f, and the BDD that represents the transition relation. SMV returns 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 together 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 BDD.

3.2 Quantitative Analysis

Several methods have been proposed to verify timed systems, as has been discussed in the introduction. These verifiers assume that timing constraints are given explicitly in some notation like temporal logic and determine if the system satisfies the constraint. In [10] 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 [24]. 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. This requirement is not necessary for the *minimum* algorithm, however, it is essential for the correctness of the *maximum* algorithm described below.

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, 24]. 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.

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*.

Informally, the algorithm computes at stage i the set I' 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 I' 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.

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

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

Figure 1: Minimum and Maximum Delay Algorithms

As before, the algorithm is implemented using BDDs, however, a backward search is required in this case. Both algorithms are proven correct in [10].

4 New Verification Techniques

In the following subsections we present three verification techniques, based on the tableau and the quantitative analysis, presented in previous sections. In Section 5 we combine all three techniques in the verification of a complex example.

4.1 Selective Quantitative Analysis — Over Paths

In this section we describe how to adapt the *minimum* and *maximum* algorithms given in Figure 1 to apply to a set of selected paths of a given structure M.

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.

Let *fair* be a set of states such that $s \in fair$ iff s is the beginning of a path which is fair with respect to *Fair(f)*. The **Path Selective** *minimum* and *maximum* **Algorithms**:

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.

4. Construct P', the restriction of P to the state set *fair*. $P' = (S', R', L')$ is defined as follows. $S' = fair$, $R' = R \cap (S' \times S')$ and for every $s \in fair$, $L'(s) = L(s)$.

5. Apply the algorithms *minimum*(st, fn) and *maximum*(st, fn, not_fn) to P', with $st = (sat(f) \times start) \cap fair$, $fn = final \cap fair$, and $not\_fn = fair - final$.

To see why the algorithms work correctly, note that by Lemma 1 P contains all paths of M that are also paths of $T(f)$. P' is restricted to the fair paths of $T(f)$. Thus, every path in P' from $sat(f) \times start$ satisfies f. Consequently, applying the algorithms to P' from $sat(f) \times start$ to $final$ 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 this method in the analysis of the PCI Local Bus to show how it can be used in the verification of real systems [7]. In this example we have computed the minimum and maximum transaction times in the PCI bus for several different configurations. One of the most important characteristics of the PCI bus is the ability to abort a transaction and restart it later. This significantly affects response time, and therefore must be considered in the implementation. Although in the actual system aborts can occur at any time, during verification they must be restricted. The reason is that if an unlimited number of aborts can occur, the maximum transaction time is infinity. Even though this is a possible behavior, it occurs rarely, and it does not provide any information about the behavior of the system in most cases.

In order to restrict the number of aborts we have implemented an abort counter in the model that is incremented whenever an abort happens. Then we used the LTL formula `G abort_counter <` n to select the paths considered for verification. This method produced information about the protocol for a specific subset of execution sequences without the need to change the model. More information about this analysis can be found in [7].

4.2 Selective Quantitative Analysis — Over Intervals

In this section we adapt the *minimum* and *maximum* algorithms of Figure 1 to apply to a set of selected intervals in a given structure M.

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.

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

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

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

This formula is true of a state s if s is not in *final*. Furthermore, there exists an interval that leads from s to a state in *prop* without going through states in *final*, and this interval has a continuation to a state in *final*.
The **Interval Selective** *minimum* and *maximum* **Algorithms**:

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 $st = sat(f) \times start$ and let $fn = prop \times final$. The algorithm *minimum*(st, fn) applied to P will return the length of the shortest interval between $start$ and $final$ that satisfies f. The algorithm *maximum*(*st, final*, $\mathcal{C}$) applied to P will return the length of a longest interval between $start$ and $final$ that satisfies f.

The correctness of the algorithm relies on the fact (stated in Lemma 1) 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 computed shortest and longest lengths over intervals from $start$ to $final$ that satisfy f.

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.

4.3 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 $st = sat(\neg f) \times start$ and $fn = prop \times final$.
4. If *minimum* results in ∞ 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.

5 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.

The FDDI network is a 100Mb/s local/metropolitan area network that uses a token ring topology [3]. It has gained popularity recently, particularly in real-time applications, since it allows communication time to be bounded. 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. There is a deadline of 100ms between the generation of multimedia data and its processing by the control processor.

The traffic in the network has been modeled as proposed in [27]. Under this protocol, stations choose a *target token rotation time* (TTRT). Each station is then allocated a synchronous capacity such that if all stations use all their synchronous bandwidth, the token returns to a station at most 2*TTRT time units after leaving it [27]. In this example the TTRT is 8. Traffic is modeled such that every 16 units (2*TTRT) the stations utilize the network as follows: *video* station, 6 units; *audio* station, 1 unit; and remainder network traffic, 8 units (in this example we will analyze only the behavior of video and audio. Therefore all the remaining traffic in the network has been grouped together).

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

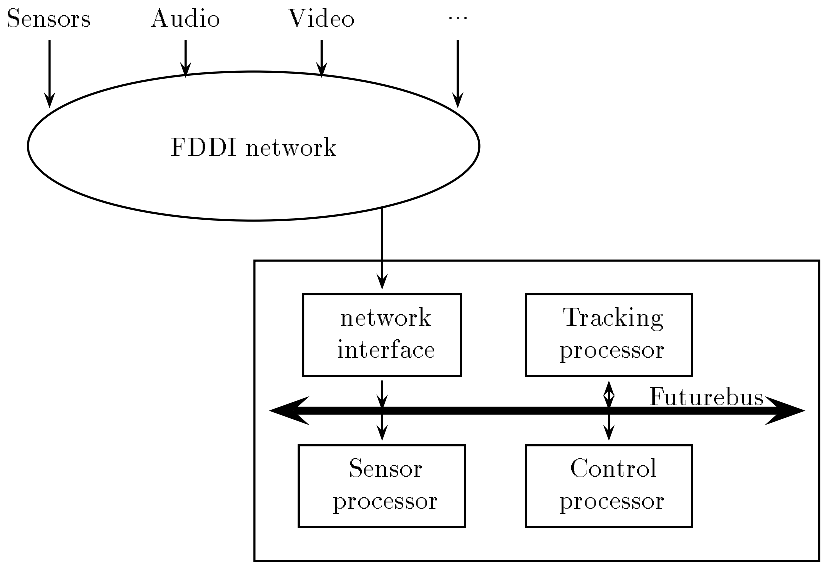

Figure 2: System Architecture

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

Figure 3: Timing requirements for tasks in the control processor

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 rate-monotonic scheduling theory, 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, with priorities being assigned by the rate-monotonic theory. Two of the 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 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 execution sequences that can actually be reached. In an abstract model such dependencies do not exist. In this example, selective quantitative analysis has been used to ensure that only execution sequences that are valid have been considered during verification.

The first deadline to be checked is the deadline of 100ms between the generation of multimedia data (signaled by variable `video.start`) and its processing in the control processor by process τ_5 (signaled by variable `t5.finish`). Ideally, we would like to compute these time bounds using `MIN{MAX}[video.start, t5.finish]`. However, since in our model we have abstracted out synchronization between tasks, this would consider paths in the model in which `t5` finishes executing just after `video`, without going through the network interface. This execution sequence corresponds to `t5` processing data generated by previous instantiations of `video`.

In order to identify the valid paths in the model, we have computed the same time bounds as before, but now considering only paths that satisfy the constraint `F interface.finish`. Unfortunately, this is still not accurate enough, as it allows for execution sequences in which `interface` executes *before* `video` finishes, or *after* `t5` starts. The actual formula used to characterize the correct paths is

```
F (video.finish & F (interface.finish & F t5.start))
```

This formula guarantees that the events `video.finish`, `interface.finish` and `t5.start` must occur, and in that order. Moreover, by using bounded selective quantitative analysis we also guarantee that these events must happen after `video.start` and before `t5.finish`. We are then able to eliminate from consideration all false paths introduced in the model, and determine the correct response times.

Using this formula for computing the time between `video.start` and `t5.finish` resulted in the interval [24, 96] (values are presented in the form [*min*, *max*]), that is, the multimedia traffic is schedulable. The audio traffic has been analyzed in a similar way, and will not be presented here for brevity. The response time for the audio station is [16, 96].

This analysis also uncovered an ambiguity in the system description. Initially, we assumed that process τ_2 processed the multimedia traffic in the control processor. In the original description this point is not clear. However, the same analysis using τ_2 instead of τ_5 produces the interval [100, 148], which is clearly not schedulable. Discussions with the authors of the original paper then clarified the issue, and in the model we introduced process τ_5 to handle multimedia traffic.

Finally, we must check 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. In order to determine how long it takes for data to go from the sensor processor to the control processor we must use a similar approach to the one described. The direct computation of `MIN{MAX}[sensor_observation, t3.finish]` searches through paths in which data does not have time to go through all the steps in the protocol.

We must, therefore, compute this time provided that a LTL formula describing the correct data path is satisfied. The formula that must be satisfied in this case is

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

By using this formula we have obtained 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. Before this time, data was blocked at the tracking processor because of its periodicity. 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.

The rate-monotonic theory was used to assign priorities to bus requests, and it states that shorter periods have higher priorities. In this case however, both processors have the same period, and their relative priority is irrelevant (from the rate-monotonic perspective). From the data transfer pattern, though, it seemed that exchanging the order of these two processors would yield a better result. We modified the design by changing the priorities, and the response time became [37, 403], an improvement of almost 50% in the response time.

Moreover, we have been able to compare the performance of both designs using interval model checking. One of the most important problems with real-time systems is priority-inversion. It occurs when high-priority tasks are blocked by low priority tasks. This can happen even with priority scheduling, in most cases caused by synchronization.

Determining the existence of priority inversion is extremely important in the analysis of real-time systems. In our example we have been able to check this parameter using interval model checking.

We are interested in determining the existence of priority inversion between the time the sensor produces data until the time the tracking processor processes this data. Priority inversion occurs in this interval if the bus is idle or the lower priority process is executing. The lower priority process is either the sensor or tracking processor, depending on the priority order. In both cases the network interface has higher priority, because it has a shorter period.

Using interval model checking we have been able to check the LTL formula `G !(bus_idle | bus_granted = lower_priority)` on the intervals between the sensor processor finishing sending data and the tracking processor 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. It would be easy to adapt this analysis to a different data pattern, and optimize the response time for any application that is considered more important. In this example we considered the data path from the sensor to the control as the most important one.

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.

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. Verifying the performance of the pci local bus using symbolic techniques. In *International Conference on Computer Design*, 1995.

[8] 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.

[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. V. Campos, E. M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *IEEE Real-Time Systems Symposium*, 1994.

[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, New Orleans, La.*, 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 NationalStandards Institute. *IEEE Standard Backplane Bus Specification for Multiprocessor Architectures: Futurebus+*, ansi/ieee std 896.1 edition, 1990.

[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] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the twelfth Conference on Principle of Programming languages*, January 1985.

[21] 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.

[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] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *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.

[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.

[27] 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.

[28] J. Yang, A. Mok, and F. Wang. Symbolic model checking for event-driven real-time systems. In *IEEE Real-Time Systems Symposium*, 1993.