A Quantitative Approach to the Formal Verification of Real-Time Systems

Sérgio Vale Aguiar Campos

September 1996
CMU-CS-96-199

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy*

Thesis Committee:

Edmund M. Clarke, Chair
Rance Cleaveland, North Carolina State University
Daniel Jackson
John P. Lehoczky

© 1996 Sérgio Vale Aguiar Campos

This research was sponsored in part by the National Science Foundation under grant no. CCR-9217549, by the Semiconductor Research Corporation under contract 95-DJ-294, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research projects Agency (ARPA) under grant F33615-93-1-1330.

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

Keywords: real-time systems, formal verification, symbolic model checking, binary decision diagrams, rate-monotonic scheduling, schedulability, quantitative timing analysis, Verus language, CTL, LTL, FDDI, Futurebus, PCI Local Bus, robotics controller, aircraft controller.

Carnegie
Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
Pure and Applied Logic

A Quantitative Approach to the Formal Verification of Real-Time Systems

SÉRGIO VALE AGUIAR CAMPOS

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Edmund M. Clarke THESIS COMMITTEE CHAIR	12/18/96 DATE
DEPARTMENT HEAD	12/18/96 DATE

APPROVED:

DEAN	1/28/97 DATE

Abstract

The task of checking if a computer system satisfies its timing specifications is extremely important. These systems are often used in critical applications where failure to meet a deadline can have serious or even fatal consequences. This work proposes an efficient method for performing this verification task. The method is based on temporal logic model checking, a technique for verifying concurrent reactive systems. In the proposed technique, a real-time system is modeled by a state-transition graph represented by binary decision diagrams. Efficient symbolic algorithms exhaustively explore the state space to determine whether the system satisfies a given specification.

In addition to accepting an explicit timing constraint, and checking if it is satisfied, our approach computes quantitative timing information such as minimum and maximum time delays between given events. These results provide insight into the behavior of the system as well as assist in the determination of its temporal correctness. The technique evaluates how well the system works or how seriously it fails, as opposed to only if it works or not, allowing a much richer analysis than previous methods. Response time to events, schedulability of a task set and system performance are examples of information produced by our algorithms. They also provide insight into how changes in the parameters affect global behavior and allow fine-tuning of the system based on these results.

These techniques have been used in the verification of several industrial real-time systems such as an aircraft controller, a robotics system and the PCI local bus, demonstrating that the method proposed is efficient enough to be used in real-world designs. The examples show how the information produced can assist in designing more efficient and reliable real-time systems.

Abstract

“You must remember this,
a kiss is just a kiss,
A sigh is just a sigh;
The fundamental things apply,
As time goes by.”

— Herman Hupfeld.

Abstract

Acknowledgments

To my wife Alessandra, for providing complete unrestricted support whenever I needed it, even when I didn't deserve it. But even more important, for giving me a reason to go on, for being the light at the end of my tunnel. There would be no reason for writing this thesis without her.

To my advisor Edmund Clarke, for the incredible amount of time he spent with me, from the beginning when he patiently listened to my so wrong ideas about verification to the end when he always found the time to read and comment on the thesis. For teaching me so much not only about formal verification, but also about all aspects of research, from writing papers that can actually be understood to grant proposals that can actually be accepted. If I ever succeed in becoming a good researcher I will be using what he has taught me.

To Marius Minea, for the many long discussions on formal verification and all other topics. For all the help with ideas on how to do things, prove the algorithms, comments and suggestions on papers we've written and on this thesis. It's been a pleasure working with Marius all this time, and I hope we can continue to work together.

To David Long, a former student of Ed, for the patience to explain to me everything I've always wanted to know about model checking but couldn't find anyone with enough time to answer. And for continuing to help with so many other questions after he left CMU by patiently answering my e-mails and phone calls.

Acknowledgments

To Orna Grumberg, for showing me a very special style of doing research. She is clear, concise and very efficient. It is a pleasure to work with Orna, whose style I've been trying to learn. Unfortunately, I still have a long way to go.

To many other researchers that have helped me. To Andreas Kuehlmann from IBM for the original insight behind the optimized condition counting algorithms. To Ken McMillan, Jerry Burch, Xudong Zhao and Somesh Jha for the many important discussions that helped shape this work. To the Real-Time Mach group members, particularly R. Rajkumar for many discussions about real-time systems and how to verify them.

To the members of my committee, Rance Cleaveland, Daniel Jackson and John Lehoczky for trying to understand all my crazy ideas and not only succeeding (in some cases better than me), but also for helping me understand them better as well. Their comments and suggestions have made this thesis significantly better.

To my parents Lia and Daniel and to my sister Daniela, for the help and support throughout my whole life. Even though they have not been present during my Ph.D., without their support I would not have been able to be here.

Table of Contents

Chapter 1 Overview

1.1 Motivation

In many computer applications predictable response times are essential for the correctness of the system. Such systems are called *real-time systems*. They occur in many critical applications in which a late (or sometimes early) response can have severe consequences. Examples of such applications include controllers for aircraft, industrial machinery and robots. Due to the nature of such applications, errors in real-time systems can be extremely dangerous, even fatal. Guaranteeing the correctness of a complex real-time system is an important and non-trivial task. Because of this, only conservative and usually ad hoc approaches to design and implementation are routinely used. This leads to predictable but inefficient designs. The use of modern software engineering techniques can improve the efficiency of these systems without forgoing predictability. Recently, the development of methods such as the *rate monotonic scheduling theory* [53,59,68] has helped increase the popularity of formal approaches by providing designers with powerful tools for analyzing real-time systems. Current real-time designs incorporate these ideas with increasing frequency.

Other factors make the validation of real-time and non real-time systems particularly difficult. The architecture of computer applications is becoming more and more complex each day. The more complex a system, the higher the possibility of errors being introduced in its design. Moreover, performance is also becoming a more important factor in the success of new applications. Due to competition, new products have to fully utilize the available resources. A slow component can compromise the performance of the whole system, and consequently its acceptance by the market. Although not traditionally associated with real-time systems, verifying timing properties of these applications is also critical. The task of verifying that new applications satisfy their timing specifications is more critical and difficult today than ever.

The main objective of this work is to explore the application of formal methods to the validation of real-time systems. Methods such as *temporal logic model checking* [10,19,20] have been very successful in validating realistic industrial designs. Model checking techniques have the ability to verify designs with extremely large state spaces efficiently. Models with up to 10^{20} states can be verified in minutes [10,15]. Methods such as rate monotonic theory offer an elegant way of analyzing the performance of a real-time system. We have developed a tool for the analysis of real-time systems by combining ideas from both techniques. This tool allows the user to define a real-time system using a language especially designed to simplify the description of time related characteristics. Algorithms derived from model checking are used to extract *quantitative* properties of such a model. This information is used to check if the system does indeed satisfy its temporal specifications. Moreover, our algorithms provide an insight into the behavior of the system, helping to understand and optimize its design. We believe that the use of formal methods can increase the efficiency and reliability of systems in general, and particularly of real-time systems. It is our hope that this work and its future extensions can contribute to this goal and open new possibilities in the design of efficient and reliable real-time and non real-time systems.

1.2 Verification Tools for Real-Time Systems

Temporal Logic Model Checking

Temporal logic model checking [19,20] is an approach for the verification of concurrent systems that has achieved significant results recently. In this technique, computer systems are represented by state-transition graphs and specifications are written as formulas in a propositional temporal logic. Verification is accomplished by an efficient search procedure that views the transition system as a model for the logic, and determines if the specifications are satisfied by that model.

There are several benefits to this approach. An important one is that the procedure is completely automatic. The model checker accepts a model description and logic formulas describing the specifications; it then determines if the formulas are true or not for that model. If a formula is not true, the model checker can often provide a *counterexample*. The counterexample is an execution trace that shows why the formula is not true. This is an extremely useful feature because it can help locate the source of the error and speed up the debugging process. Another benefit is the ability to verify partially specified systems using *nondeterminism*. If the behavior of the component that determines the value of a given variable hasn't been specified, the variable can be assigned any possible value nondeterministically. The actual behavior of the variable is a subset of the modelled behavior, and useful information about correctness can be gathered before all the details have been determined. This allows the verification to proceed concurrently with the design.

The concept of *symbolic model checking* has been developed later [10,62]. In this approach the transition relation is represented implicitly by boolean formulas, and implemented by *binary decision diagrams* [6]. This usually results in a much smaller representation for the transition relation and set of reachable states, allowing the size of the models being verified to increase up to more than 10^{20} states. By using such techniques it has become possible to verify realistic industrial systems formally. Significant results have been achieved, such as the verification of the Futurebus^{+} protocol, adopted by the U.S. Navy [23]. That work has uncovered protocol errors that were not previously known.

It is possible to use symbolic model checking to verify real-time systems. However, current tools have limitations that make it difficult to perform this verification. It is difficult, for example, to express timing properties. It is possible to express the property that "event p will happen in the future", but it is not simple to express the property that "event p will happen in at most n time units". Moreover, quantitative information such as response time or the number of occurrences of events cannot be directly obtained using these techniques. Pure symbolic model checking cannot be used in a natural and efficient way to verify many types of real-time systems that occur frequently in practice.

Rate Monotonic Scheduling Theory

Because real-time systems are used in critical applications, until recently only conservative approaches have been commonly used in their design, leading to simple but inefficient designs. One example of such a safe technique is static time-slicing, which divides time equally among all tasks. Each task executes until its time slot has been used and then releases the processor. The resulting program is very simple to analyze, but rather inefficient, since all tasks are given equal resources, regardless of their importance or resource utilization. Recently, more powerful techniques to analyze the behavior of a real-time system have become more popular. The *rate monotonic* scheduling theory (RMS) [53,59,68] is one example. The RMS theory is applicable to systems described by a set of periodic tasks. It consists of two components, the first being an algorithm for assigning priorities to tasks in order to maintain predictability. This algorithm assigns higher priorities to processes with shorter periods. Optimal response time with respect to static priority algorithms is guaranteed by the RMS theory if priorities are assigned according to this rule [59]. The second component of the RMS theory is a schedulability test based on total CPU utilization; a set of processes (which have priorities assigned according to RMS) is schedulable if the total utilization is below a computed threshold. If the utilization is above this threshold, schedulability is not guaranteed. RMS is a powerful tool for analyzing real-time systems. It is simple to use, yet it provides very useful information for designers.

However, this analysis imposes a series of restrictions on the set of processes. Only certain types of processes are considered, with limitations, for example, on periodicity and synchronization. Recent work has extended this theory to more general classes of processes, but limitations still exist [38]. Although suited to the verification of real-time systems, RMS can only handle systems that can be described within the theory. Moreover, the types of properties that can be verified is also restricted to properties that can be modeled as task execution times. Verifying different types of systems such as distributed systems or systems that do not have a regular communication pattern is not a trivial task in general. The task of checking for properties that cannot be easily expressed as task execution times such as the number of occurrences of arbitrary events in the system can also be complex.

Other Methods

Another approach to schedulability analysis uses algorithms for computing the set of reachable states of a finite-state system [18,35,36]. A model for the real-time system is constructed with the added constraint that whenever an exception occurs (e.g. a deadline is missed) the system transitions to a special exception state. Verification consists of computing the set of reachable states and checking whether the exception state is in this set. No restrictions are imposed on the model in this approach, but the algorithm only checks if exceptions can occur or not. Other types of properties cannot be verified, unless encoded in the model as exceptions. Even though most properties can be encoded as exceptions, this can sometimes be difficult and error-prone. Symbolic model checking techniques have also been extended to handle real-time systems [28,29,77]. However, these methods as well as the others mentioned only determine if the system satisfies a given property, and do not provide detailed information on its behavior. Restricted quantitative analysis on discrete-time models can be performed in [27], but only to the extent of computing minimum/maximum delays.

In this work we use a discrete notion of time. In recent years, there has been considerable research on algorithms that use continuous time [1,2,34,39,41,55,63]. Most of these techniques use a transition relation with a finite set of real-valued clocks and constraints on times when transitions may occur. It can be argued that such algorithms lead to more accurate results than discrete time algorithms. However, an uncountable infinite state space is required to handle continuous time, because the time component in the states can take arbitrary real values. Most verification procedures based on this paradigm depend on constructing a finite quotient space called a *region graph* out of the infinite state space. Unfortunately, the region graph construction is very expensive in practice and current implementations of the algorithms can only handle quotient spaces with at most a few thousand states. This makes it impossible to verify large complex systems such as the ones described in chapter 6 using continuous time tools. Dense time models in which restricted quantitative analysis can be performed can be found in [42,75].

1.3 The Proposed Approach

In this work we propose a new method for specifying and verifying real-time systems. The system being verified is specified in the *Verus* language and then compiled into a state-transition graph. Algorithms derived from symbolic model checking are used to compute *quantitative* information about the model. An important benefit of this approach is that the Verus language has been especially designed to allow a straightforward description of the temporal characteristics of programs. Another advantage is that the information produced allows the user to check the temporal correctness of the model: schedulability of the tasks of the system can be determined by computing their response time; reaction times to events and several other parameters of the system can also be analyzed by this method. This information provides insight into the behavior of the system and in many cases it can help identify inefficiencies and suggest optimizations to the design. The same algorithms can then be used to analyze the performance of the modified design. The evaluation of how the optimizations affect the design can be done *before* the actual implementation. This can significantly reduce development costs.

Other advantages of our approach include the fact that the Verus code serves as a precise description for the system, which can uncover subtle ambiguities and can be used for documentation purposes. Also, because we use a discrete notion of time, we are able to take advantage of symbolic techniques in which the transition relation is represented by a binary decision diagram. This enables us to handle systems that are several orders of magnitude larger than can be handled using continuous time techniques.

Our method extends several others that have been mentioned. The model of a real-time system, and the algorithms for exploring the state space are derived from model checking. However, the method proposed allows the natural expression of many types of real-time properties that cannot be easily described in the original method, such as properties that depend on the exact timing behavior of the system. The definition of real-time system, and of its main characteristics are derived from the rate monotonic theory. But unlike RMS, our approach does not impose *a priori* restrictions on the types of systems and properties being verified. Limitations do exist, however, due mostly to the complexity of the verification algorithms. But they do not depend on the structure of the system, only on the amount of time and memory required for verification.

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. Real-time systems usually do not execute in lock-step, 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 real-time system developers often significantly reduce asynchronism in their designs to ensure predictability. In fact, most real-time systems we have analyzed are more synchronous than traditional circuits, and have been successfully verified using the method proposed [13,14,16].

The main limitation of our approach is the inherent complexity of the model checking problem. Constructing the model has an exponential asymptotic complexity in the number of components and there are no guarantees that our algorithms will terminate in any practical sense. However, we have achieved good success in this area; in most cases verification is performed in minutes, even for complex real-world systems. It must be said, however, that these problems are inherent to formal verification of timed systems, and that we know of no approach that has solved them.

Overview

The remainder of the chapter gives an overview of the proposed method. We describe how a real-time system is modeled as a state-transition graph, and present the Verus language, used to describe the system being verified. We then briefly explain how the verification algorithms work, and how these results can be used to analyze real systems. We conclude the chapter by outlining the main contributions of this work.

1.3.1 Modeling a Real-Time System

A model of the system in our algorithms is a labeled state-transition graph M. The labels correspond to the values of the variables in the program, and transitions correspond to the passage of time in the model. The key to the efficiency of the algorithms is to use BDDs to represent the labeled state-transition graph and to verify if the formula is true or not.

In this method transitions are represented by boolean formulas. A formula f represents a transition between states s and s' iff the formula is true when we substitute the variable values in states s and s' for the variables in f (using different variables for current and next state). The formula f is represented by a BDD. Details about this representation can be found in Section 4.1 and [8,62].

Lazy composition

In the vast majority of cases a real-time system is described by a set of processes that execute concurrently. Given the transition relation for each process, a parallel composition algorithm constructs the global transition relation of all processes and their interaction. The composition algorithm is essential to the verification of non-trivial real-time systems. It is, however, extremely expensive. The number of states of a composed model can be exponential in the number of processes. The complexity of the composition algorithm is the main limiting factor on the size of systems being verified. We propose a new approach in process composition called *lazy composition*. The basic idea is to avoid composing processes whenever possible, performing the operation only when necessary.

With the new technique the composition algorithm is applied at each time the verifier computes the image or pre-image of a state set. When computing the image of a state set S, we are only interested in transitions that start in S. At this point the lazy composition algorithm reduces the transition relations of each process by simplifying and possibly eliminating transitions that do not start in S. The resulting transition relations are often much simpler than the original ones, while preserving all transitions that start in S. The simplified relations are then composed and the normal image computation algorithm can be applied. Significant gains in time and space during verification can be accomplished by this method.

1.3.2 The Verus Language

We have designed a new language to be used as the specification language for the real-time systems verified. The main goal of this language is to allow engineers and designers to describe real-time systems easily and efficiently. It is an imperative language with a syntax resembling that of C. Special primitives are provided to express of timing aspects such as deadlines, priorities, and time delays. The available data types are integer and boolean. Nondeterminism is supported, which allows partial specifications to be described. The language constructs have been kept simple in order to make an efficient compilation into a state-transition graph possible. Smaller representations can then be generated, which is critical to the efficiency of the verification and permits larger examples to be handled.

There are several other languages for specifying finite-state real-time systems. However, they are suited for different applications, and usually only allow a natural description of characteristics that are typical of those applications. For example, Lustre [72] and Signal [37] are languages for describing sequential circuits. They are declarative languages, and in this sense they resemble the SMV language [62], used for describing circuits in our original symbolic model checker. In these languages one describes explicitly the relationship between variables, but not the flow of control. Imperative languages take the opposite approach, by explicitly describing the control flow. Declarative languages, however, do not provide a natural way of describing real-time programs, usually implemented on imperative languages. In fact, our experience with SMV was the motivation for developing Verus.

Esterel [5], on the other hand, is an imperative language, better suited for describing programs. Its syntax, however, may be very unfamiliar to most designers of real-time systems, used to program in C or similar languages. Moreover, Esterel constructs, unlike those of Verus, have not been designed to simplify modeling real-time systems. For example, specifying the execution of a periodic process with a deadline is not as straightforward as in Verus. Finally, Esterel is a deterministic language; it does not allow the expression of nondeterminism. Modechart [47] is another example of real-time specification language. It is a graphical language in which nodes represent states, and transitions are explicitly drawn between states. This language, however, is more restrictive than Verus due to its graphical nature. Complex constructs such as `periodic` may be difficult to draw. Moreover, it is an explicit state enumeration language, since individual states are drawn in the program. Many systems are too large to be naturally described using graphical languages.

A different approach is taken in Spin [43] and Murφ [30]. These systems use languages that resemble C, but that have actually a significantly different semantics. Modeling a system originally written in C in one of these systems may cause confusion between a similar syntax, but different semantics. Moreover, both systems are better suited to verify asynchronous designs. Their languages have been designed to allow the straightforward expression of such systems. It is not clear how natural or efficient it is to write a synchronous program in one of these languages.

1.3.3 Verification Algorithms

In the previous section we have described how to model a real-time system in a form amenable to formal analysis. This section will describe the algorithms that perform this analysis. The types of properties that can be expressed are also discussed.

Real-Time CTL Model Checking

Computation Tree Logic, CTL, is the temporal logic used in our verification system [19,20]. In CTL it is possible to express properties such as "p will eventually occur", or "p will never be asserted". However, it is not possible to express bounded properties such as "p will occur in less than 10ms" directly. Properties such as this can only be expressed using nested next state operators. However, the resulting formula can be very complex and cumbersome to work with. The *bounded until operator* overcomes this restriction by allowing bounds on all CTL operators to be specified [33]. It has the form: $\mathbf{U}_{[a,b]}$, where $[a, b]$ defines the time interval in which the property must be true. Informally, $f \, \mathbf{U}_{[a,b]} \, g$ is true of some path if g holds in some future state s' on the path, f is true in all states between state s at the beginning of the path and s', and the distance from s to s' is within a and b. All other CTL temporal operators are defined in terms of the bounded until [11]. The logic obtained by augmenting CTL with bounded operators is called RTCTL.

The new logic allows many important properties of real-time systems to be verified. For example, we have used it to show the existence of priority inversion [66] in a real-time system [11]. In this example, we have modeled a simple real-time system in which processes communicate in a non-regular pattern. The main objective is to determine which problems can arise from this communication and how to avoid them. The bounded until operator has allowed us to determine the existence of priority inversion, and to check that the solution implemented, priority inheritance, avoids the problem.

We have also used RTCTL model checking in several other occasions to verify time bounded properties of real-time and non real-time systems. Some examples include verifying that an industrial communications circuit would *not* meet its timing specification [78] and the verification of the generalized railroad crossing example [40].

Quantitative Algorithms

Most verification algorithms assume that timing constraints are given explicitly in some notation like temporal logic. Typically, the designer provides a constraint on response time for some operation, and the verifier automatically determines if it is satisfied or not. Unfortunately, these techniques do not provide any information about how much a system deviates from its expected performance, although this information can be extremely useful in fine-tuning the behavior of the system.

We present algorithms that determine the minimum and maximum length of all paths leading from a set of starting states to a set of final states. We also present algorithms that calculate the minimum and the maximum number of times a specified condition can hold on a path from a set of starting states to a set of final states. Our algorithms provide insight into *how well* a system works, rather than just determining whether it works at all. They enable a designer to determine the timing characteristics of a complex system given the timing parameters of its components. This information is especially useful in the early phases of system design, when it can be used to establish how changes in a parameter affect the global behavior of the system.

Several types of information can be produced by this method. Response time to events is computed by making the set of starting states correspond to the event, and the set of final states correspond to the response. Schedulability analysis can be done by computing the response time of each process in the system, and comparing it to the process deadline. Performance can be determined in a similar way. The algorithms have been used to verify several real-time and non real-time systems. Several examples of systems verified are discussed in later chapters.

Selective Quantitative Analysis and Interval Model Checking

The algorithms described above compute the minimum and maximum time delays along *every* possible execution sequence of a real-time system. In many situations, however, we may be interested in computing time delays that relate only to a subset of the execution sequences that satisfy a given property. For example, in the aircraft controller example

[13] the time between requesting the activation of the weapons and their actual firing time is computed. The maximum time in that example is infinity. The weapons may never fire because the firing sequence can be aborted. It may be the case, however, that the designers want to compute the maximum response time of the weapon subsystem *provided that no abort occurs.*

We propose a method for specifying and verifying properties such as these. The user can restrict the set of paths that will be considered by specifying a property that must be satisfied in all paths traversed. This property is expressed using *linear-time temporal logic* (LTL). Special model checking techniques [22] are then used to ensure that only paths that satisfy the formula are considered by the algorithms.

1.3.4 Analysis of the results

The power of our method comes mostly from the different types of analysis that can be performed with the results produced by the algorithms. This section explores different ways in which these results can be used to extract the correctness of a design, its performance and insight into its behavior.

The basic algorithms compute minimum and maximum time delays between two state sets *start* and *final.* We can use them to determine response time to events by applying the algorithms to the predicates *start* = *event* and *final* = *response.* Such numbers also give information about the correctness of a design. If the maximum is less than infinity then *event* always implies *response* in the future:

$$\mathrm{MAX}(\mathit{event}, \mathit{response}) < \infty \quad \text{iff} \quad \mathbf{AG}(\mathit{event} \rightarrow \mathbf{AF}\ \mathit{response})$$

Notice that a similar statement can associate a minimum value to the existence of a path. This shows that our algorithms can express the same properties as these specific CTL formulas. It can be argued that CTL can express more complex properties than those described above. However, properties such as the one described are certainly some of the most frequently checked, and this correspondence is extremely useful in practice.

The minimum and maximum algorithms can also be used to perform the schedulability analysis of a real-time task set. We can compute bounds on the execution time of all processes, and check if they are within the corresponding deadlines. That also gives the user information about the load on the system: maximum execution times close to the deadline indicate high load. We have applied this technique to various real-time systems, such as the aircraft controller described in [13]. Non real-time systems can also benefit from this analysis. We can compute response time for any event in the system, and check the performance against the specifications. These results can provide information that may have significant impact in market acceptance when compared to the expected behavior or competitor products. For example, a correct product may have a performance bottleneck that may compromise the performance of the whole system. If the bottleneck cannot be identified and corrected, the product may lose its market share due to poor performance. We have applied this method to the verification of the PCI local bus [15].

The algorithms that count the number of times a condition occurs in a path can be extremely useful in this analysis as well. Given a *condition* to be counted and two events *start* and *final*, these algorithms compute the minimum and maximum number of times *condition* holds on any path from *start* to *final*. They make it possible to determine even more detailed information about the system. For example, in real-time systems it is possible to compute information such as priority inversion time by making *start* and *final* ‘request for execution’ and ‘end of execution’ at a certain priority level respectively, and the *condition* to be counted to be ‘executing at lower priority’. In non real-time systems for example, it is possible to compute the overhead associated with processing of data by making *start* a ‘request for transaction’, *final* the ‘end of transaction’, and *condition* to be ‘data being processed’.

Another way in which designers can benefit from this method is by fine-tuning the system for optimal performance. After computing the response times for important events, the designer can change parameters in the model and check how the response times are affected. In this way it is possible to optimize performance by determining the effect of the parameters on the global behavior. Even finer tuning is possible by using selective quantitative analysis. For example, a very useful practice is to optimize the performance for the most common case, while maintaining the correctness of uncommon cases. Selective quantitative analysis can be used to restrict the model to the common cases. Optimization can then be performed on this model. Finally, the complete system can be checked for correctness by removing the selection condition.

Several real-time systems have been analyzed using the method proposed. One example is the aircraft controller described in [60]. This control system is characterized by a set of real-time tasks, each controlling one subsystem of the aircraft. We have modeled this control system and analyzed it using the algorithms described. We have been able to determine the schedulability of the task set and to determine the response times for specific events of the system such as how long it takes from the moment the pilot presses the firing button until the weapons are actually fired.

Another example that we have analyzed is a robotics system used in nuclear plants to measure the shape of pipes by moving around them with a distance sensor [38]. We have been able not only to determine the schedulability of this task set but also to discover inefficiencies in the design. The results produced by the algorithms also suggested optimizations. The modified design has also been analyzed by the same algorithms. It has a lighter load and data is consumed faster than in the original design.

Other systems that have been analyzed include a medical monitoring system, the PCI Local Bus and a distributed real-time system. In all these cases we have been able to analyze the correctness and performance of the system, and in most cases optimize it using the method proposed.

1.4 Summary and Main Contributions

In this work we propose a new method for the formal verification of real-time systems. The method allows a detailed and accurate analysis of the behavior of the system and is efficient enough to be used in the verification of real systems. We have used it to analyze several complex systems. The analysis performed using this method not only demonstrates the correctness of the design, but in many cases it can uncover ambiguities in the behavior that might be difficult to find otherwise.

Verus extends previous methods in several directions. It allows the natural expression of many types of real-time systems that occur in practice via a language especially designed to simplify the description of timing characteristics such as periods and deadlines. Previous languages cannot in general be used as efficiently because they either do not have the primitives needed to express timing characteristics (e.g. SMV [62]), lack nondeterministic features (e.g. Esterel [5]) or can be more restrictive than the language proposed (e.g. Modechart [47]).

Moreover, many other verification methods such as model checking cannot directly verify several types of properties that can be checked in a straightforward way in Verus. Time bounds and other quantitative information such as the number of occurrences of events in the system are examples of properties that cannot be easily obtained using standard model checking. Analyzing the performance and determining the timing characteristics of a model is very simple in Verus, but it is not possible (except to a very limited extent) with traditional model checking or reachability based systems. For example, model checkers can only check time bounded properties by expressing them using nested next state operators. The resulting formula, however, is often very large and cumbersome, and impractical to work with. Quantitative information can be obtained by adding counters that flag the occurrence of the event of interest, and checking that the value of the counter is within a certain range. However, adding counters is an expensive operation, significantly increasing the complexity of the verification.

Even though it allows the expression of a richer set of timing properties than standard model checking, the rate monotonic theory is also more limited than the Verus approach in many aspects. The description of a system verified by RMS has to fit a very rigid structure. For example, tasks have to be periodic or to be modelled using an sporadic server (see [70] and Section 2.2.2); synchronization has to follow protocols such as priority inheritance (see [66] and Section 2.2.2) which can be very restrictive. Any deviation from this structure has to be reformulated or the system cannot be verified. In some cases it is possible to change the system description to allow the verification, for instance, by describing an aperiodic processes using an aperiodic server, but in many cases this is not possible.

Distributed systems represent an important class of systems in which modifying the system description may be insufficient to correctly analyze its timing behavior. To date, RMS has required the imposition of intermediate deadlines to analyze distributed systems [69]. However, intermediate deadlines significantly change system behavior. In Verus there are no restrictions on the structure of the system; any Verus program can be verified. Moreover, RMS allows the expression of a very limited type of property; basically it computes the maximum execution times of tasks in the system. Even though a large number of interesting properties can be expressed using this paradigm, this may be very difficult in some cases. On the other hand, in Verus it is possible to determine the temporal relation between any two events in the system. For example, counting the number of occurrences of arbitrary events in specific intervals is a property that is simple to express in Verus, but difficult to express in RMS.

In most verification techniques it is possible to extend its expressive power by changing the system to fit the limitations of the algorithms and verify a modified model. For example, a counter can be introduced to represent the number of occurrences of an event, and a CTL formula can be written stating that the counter never overflows. However, modifying the system may introduce additional errors, or hide existing ones. Some properties similar to those verified by Verus can be checked using model checking or RMS by introducing modifications to the system (such as the aperiodic servers or intermediate deadlines described). But even in these cases, one important advantage of Verus is that it allows the verification of these properties *without* changing the model, ultimately performing a more accurate analysis.

One example of a system that has been verified using Verus that might have been difficult to verify using other techniques is the distributed real-time system described in section 6.6. It is a large complex system which has three main components: a network to which audio and video sources are connected, a multi-processor bus transporting this data and the destination processor for it. In this example, we analyze the time it takes for data to traverse the various components of the system. The type of quantitative analysis performed cannot be done directly using model checking or reachability based techniques. The system cannot be easily described in RMS because it is a distributed system (limitations on the ability of RMS to handle distributed systems are discussed in [69]). Finally, because it is a large system, it is not likely that a continuous time method would be able to handle its complexity.

Chapter 2 Related Approaches

This chapter will present the two methods for analyzing real-time systems that are most closely related with the proposed approach: symbolic model checking and rate monotonic scheduling. The algorithms used by Verus have been derived from symbolic model checking algorithms. The analysis performed is, however, derived from the rate monotonic theory. Knowledge about these methods is not a required prerequisite, but it can simplify understanding the Verus approach.

2.1 Temporal Logic Symbolic Model Checking

Extensive simulation is currently the most widely used verification technique for finite-state systems. However, simulation cannot usually cover all possible behaviors of a computing system. Traditional simulation is too expensive, and non-exhaustive simulation can miss important events, especially if the number of states in the system being verified is large. Other approaches for verification include theorem provers, term rewriting systems, and proof checkers. These techniques, however, are usually very time consuming, and require user intervention to a large degree. Such characteristics limit the size of the systems they can verify in practice.

Related Approaches

Temporal logic model checking [19,20] is an alternative approach that has achieved significant results recently. Efficient algorithms are able to verify properties of extremely large systems. In this technique, specifications are written as formulas in a propositional temporal logic and computer systems are represented by state-transition graphs. Verification is accomplished by an efficient search procedure that views the transition system as a model for the logic, and determines if the specifications are satisfied by that model.

There are several advantages to this approach. An important one is that the procedure is completely automatic. The model checker accepts a model description together with specifications written as temporal logic formulas and determines if the formulas are true or not for that model. Another advantage is that, for most formulas of interest (e.g. safety formulas) if the formula is not true, the model checker will provide a counterexample. The counterexample is an execution trace that shows why the formula is not true. This is an extremely useful feature because it can help locate the source of the error and speed up the debugging process. Another benefit is the ability to verify partially specified systems. Useful information about the correctness of the system can be gathered before all the details have been determined. This allows the verification of a system to proceed concurrently with its design. Consequently verification can provide valuable hints that will help designers eliminate errors earlier and define better systems.

Properties to be verified are described as formulas in a propositional temporal logic. The system for which the properties should hold is given as a state transition graph. It defines a model for the temporal logic since the semantics of the logic are given in terms of state transition graphs. The model checker traverses this graph and verifies if the model satisfies the formula. Checking that a single model satisfies a formula is much simpler than proving that a formula is valid for all possible models. Because of this fact model checkers can be more efficiently implemented than theorem provers. The first algorithms [19] use adjacency lists to represent the transition graph and have polynomial complexity in the size of the model and in the length of the formula. These systems are able to handle graphs with up to 10^5 states.

Recently, more efficient algorithms have been developed using *symbolic* techniques. In the new approach the transition relation is represented implicitly by boolean formulas, and implemented by *binary decision diagrams* [6]. This usually results in a much smaller representation for the transition relation, allowing the size of the models being verified to increase to more than 10^{30} states.

2.1.1 Computation Tree Logic

Computation tree logic, CTL, is the logic used by SMV to express properties that will be verified [20]. Computation trees are derived from state transition graphs. The graph structure is unwound into an infinite tree rooted at the initial state, as seen in figure 2. The tree is infinite because no final states are considered, only infinite paths. Paths in this tree represent all possible computations of the program being modelled. Formulas in CTL refer to the computation tree derived from the model. CTL is classified as a branching time logic because it has operators that describe the branching structure of this tree.

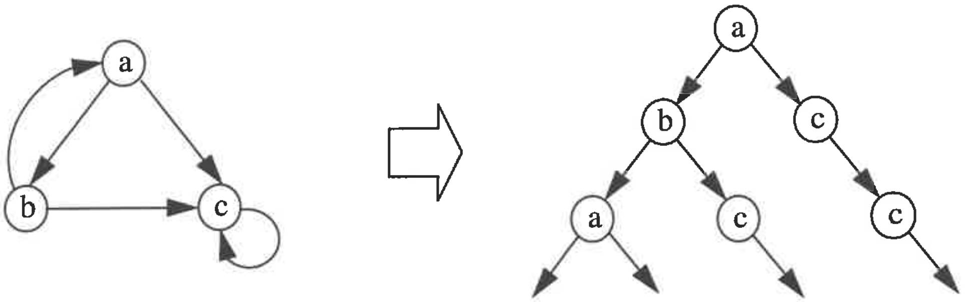


Figure 1. A state transition graph and the corresponding computation tree

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

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

Formally, the syntax for CTL can be defined by:

- Every atomic proposition p is a CTL formula.
- If f and g are CTL formulas, then so are $\neg f$, $f \vee g$, **EX** f, **EG** f and **E**[f **U** g].

The semantics of CTL formulas are defined with respect to a labeled state-transition graph, which is a 5-tuple $M = (P, S, L, N, S_0)$, where P is a set of atomic propositions, S is a finite set of states, L is a function labeling each state with a set of atomic propositions, $N \subseteq S$ x S is a transition relation, and S_0 is the set of initial states. A path is an infinite sequence of states $s_0\ s_1\ s_2 \ldots$, such that $N(s_i, s_{i+1})$ is true for every i.

If f is true in a state s of structure M, we write $M, s \models f$. We write $M \models f$ if $M, s \models f$ for all states s in S_0. The satisfaction relation is defined inductively as follows (Given the model M, we abbreviate $M, s \models f$ by $s \models f$):

1. If f is the atomic proposition $v \in P$, then $s \models f$ if and only if $v \in L(s)$.
2. s $\models \neg f$ iff it is not the case that $s \models f$.
3. $s \models f \vee g$ iff $s \models f$ or $s \models g$.
4. $s \models$ **EX** f iff there exists a path $\pi = s_0\ s_1\ s_2 \ldots$ starting at $s = s_0$, such that $s_1 \models f$.
5. $s \models$ **EG** f iff there exists a path π starting at s such that for every state s' on π, $s' \models f$.
6. $s \models$ **E**[f U g] iff there exists a path $\pi = s_0\ s_1\ s_2 \ldots$ starting at $s = s_0$ and some $i \geq 0$ such that $s_i \models g$ and for all $j < i$, $s_j \models f$.

The following abbreviations are used in CTL formulas:

$$f \wedge g \equiv \neg(\neg f \vee \neg g)$$
$$\mathbf{AX}\, f \equiv \neg\, \mathbf{EX}\, \neg f$$
$$\mathbf{EF}\, f \equiv \mathbf{E}[\mathit{true}\ \mathbf{U}\, f]$$
$$\mathbf{AF}\, f \equiv \neg\, \mathbf{EG}\, \neg f$$
$$\mathbf{AG}\, f \equiv \neg\, \mathbf{EF}\, \neg f$$
$$\mathbf{A}[f\, \mathbf{U}\, g] \equiv \neg\, \mathbf{E}[\neg g\, \mathbf{U}\, \neg f \wedge \neg g] \wedge \neg\, \mathbf{EG}\, \neg g$$

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

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

2.1.2 Symbolic Model Checking

Early model checking algorithms represent the transition graph by adjacency lists [19]. All existing states are explicitly enumerated. However, the number of states in the model can be exponential in the number of concurrent components in the system. This frequently causes state explosion problems. The size of systems that can be verified is severely limited. Symbolic model checking represents states and transitions using boolean formulas. This usually generates smaller representations, because it can automatically eliminate redundancy in the graph. Implementing these boolean formulas as BDDs leads to very efficient algorithms for model checking that are able to verify much larger systems than previous ones. This section explains the symbolic model checking approach.

Binary Decision Diagrams

Binary decision diagrams (BDD) are an efficient way to represent boolean formulas. BDDs often provide a much more concise representation than traditional representations like conjunctive normal form or disjunctive normal form. They can also be manipulated very efficiently [6]. Another advantage offered by BDDs is that they provide a canonical representation for boolean formulas. This means that two boolean formulas are logically equivalent if and only if they have isomorphic representations. It greatly simplifies the execution of operations that are performed frequently like checking equivalence of two formulas or deciding if a given formula is satisfiable or not. Because of all these characteristics, BDDs have found application in the implementation of many computer aided design and verification tools.

BDDs can be better understood by first considering how boolean formulas can be represented by binary decision trees. The nodes in the decision tree correspond to the variables of the formula. Descendants of a node are labelled with *true* or *false*. The value of the formula for a given assignment of values to the variables can be found by traversing the tree from root to leaf. At each node the descendant labelled with the value of that variable is chosen. Each leaf corresponds to a particular assignment to the variables, and contains the truth value of the formula for that assignment.

This representation is not particularly compact, because it may store the same information repeatedly in different places. BDDs are derived from binary decision trees, but their structure is a directed acyclic graph instead of a tree. Redundant information in the structure is avoided by sharing common subtrees. As in decision trees, nodes are visited in sequence, from root to leaf. Note that BDDs impose a total ordering in which the variables occur in this sequence. This order is preserved for all BDDs in use at the same time. For example, the BDD shown in figure 1 represents the formula $f = (a \wedge b) \vee (c \wedge d)$ using the ordering $a < b < c < d$ for the variables.

Given an assignment for the variables in f we can decide if this assignment satisfies the formula by traversing the BDD from root to leaf. At each node we follow the path that corresponds to the value assigned to the variable in the node. The leaf indicates if the formula is satisfied or not for that particular assignment. Notice that redundancy is eliminated in two ways. Common subtrees are not replicated, as can be seen in the figure below on the paths when a is false and when b is false. Also, when all the leaves of a subtree have the same value, the subtree is eliminated, and a leaf of that value is inserted at its place. In the figure, when a and b are both true a subtree containing the variables c and d is eliminated because all of its leaves would have the value 1.

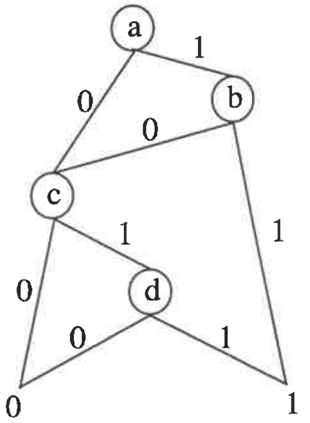


Figure 2. BDD for formula $(a \wedge b) \vee (c \wedge d)$

For any boolean formula there exists a unique BDD for a given variable ordering [6]. The BDD size is critically dependent on the variable ordering. It is exponential in the number of variables in the worst case. Given a good variable ordering, however, the size is linear in many practical cases. Using a good variable ordering is very important, but finding the optimal order is in itself an exponential problem. Nevertheless, there are many heuristics that work quite well in practice.

Efficient algorithms exist to handle boolean formulas represented by BDDs. Given BDD representations for f and g, algorithms for computing $\neg f$ and $f \vee g$ are given in [6]. Algorithms for quantification over boolean variables and substitution of variable names are also required by the model checker. It is simple to compute the restriction of a formula f with a variable v set to 0 or 1. We will denote the restriction of f with v set to 0 by $f|_{v=0}$, and the restriction of f with v set to 1 by $f|_{v=1}$. The formula $\exists v\ [\ f\ ]$ is defined as $f|_{v=0} \vee f|_{v=1}$, and $\forall v[\ f\ ]$ is defined as $\neg\exists v[\ \neg f\ ]$. Variable substitution can be accomplished using the quantification algorithm. $f\langle v \leftarrow w\rangle$ denotes the substitution of variable w for variable v in formula f. It is computed as $f\langle v \leftarrow w\rangle = \exists\ v\ [(v \Leftrightarrow w) \wedge f\ ]$. These operations are performed very frequently in the model checker, and more efficient algorithms are used in the actual system. These algorithms can be found in [8].

Representing the Model

The key to the efficiency of the algorithm is to use BDDs to represent the labeled state-transition graph and to verify if the formula is true or not. The representation used in Verus is the same as the one used by symbolic model checking. Details can be found in Section 4.1 and [8,62].

By definition, time passes by one time unit at each transition. This does not restrict the models that can be verified by the method, because non-unit transitions can be modeled as a sequence of unit transitions. Nondeterministic transition times can also be implemented in the same way, by using stuttering [11].

Frequently a model is described by a set of processes that execute concurrently. Given a set of processes and a state-transition graph for each, a parallel composition algorithm is used to construct a global transition system in which all processes execute concurrently. Two composition models are normally implemented by model checking: synchronous and asynchronous composition. In synchronous composition, all processes transition at the same time, while in asynchronous composition only one transitions. In asynchronous composition the choice of which process executes is non-deterministic and fairness is used to avoid starvation [62]. Both models are implemented using BDDs.

The composition algorithm is extremely expensive; it often generates an exponential number of states in the composed graph. However, the efficiency of symbolic model checking and the fact that our method uses discrete time allows the use of composition in several practical systems without state explosion problems.

Fixpoint characterization

Consider a labeled transition graph M with set of states S. We can denote a lattice of predicates over S by *Pred*, where each predicate is identified with the set of states in S that make it true, and use set inclusion as ordering. A functional F that maps *Pred*(S) to *Pred*(S) is called a *predicate transformer.* Informally, *Pred*(S) is a set of states, and F is a function from sets of states to set of states.

As described in [25], if a predicate transformer F is monotonic, it has a least fixpoint **lfp** $Z[F(Z)] = \cup_i F^i(\mathit{false})$ and a greatest fixpoint **gfp** $Z[F(Z)] = \cap_i F^i(\mathit{true})$. We can compute both fixpoints by iteration. Starting with $Z^0 = \mathit{false}$ (for **lfp**) or $Z^0 = \mathit{true}$ (for **gfp**), we have $Z^{i+1} = Z^i \cup F(Z^i)$ for **lfp** and $Z^{i+1} = Z^i \cap F(Z^i)$ for **gfp**. The fixpoint is found when $Z^i = Z^{i+1}$. If the number of elements in *Pred*(S) is finite, termination is guaranteed, because there can be no infinite sequence of Z^is such that $Z^i \neq Z^{i+1}$.

We can identify each CTL formula f with the predicate $\{s \mid M,s \models f\}$ in *Pred*(S) (this is the set of states that satisfy f). Then, we can characterize each basic CTL temporal operator as a fixpoint of an appropriate predicate transformer. The set of states that satisfy the until operator **E**[f **U** g] is given by the least fixpoint of $Z = g \vee (f \wedge \mathbf{EX}\ Z)$. Informally **E**[$f$ **U** g] is true at state s, if either g is true in s, or f is true in s and there exists a successor state where **E**[f **U** g] is true. The set of states that satisfy the **EG** f operator is given by the greatest fixpoint **EG** f of $Z = f \wedge \mathbf{EX}\ Z$. Informally, this means that **EG** f holds in a state s if f holds in s and **EG** f holds in a successor state of s. Proofs that the characterizations above correspond to the expected semantics are given in [25].

The Model Checking Algorithm

Given a CTL formula f and a model M represented as described above, the model checking problem consists of finding the set of states in M that satisfy f. The model checking algorithm is defined inductively over the structure of CTL formulas. It accepts the formula as an argument (and M as an implicit argument), recurses over the structure of f and returns a BDD that has one boolean variable for every atomic proposition in V. The resulting BDD is true of a state if and only if f is true in that state. The algorithm is:

- If f is an atomic proposition p, return the BDD that is true if and only if p is true. This is simply the BDD for p.
- If f is $\neg g$ or $g \wedge h$, use the standard BDD algorithms for computing boolean connectives.
- If f is **EX** g, then we must verify if g is true in a successor state of the current state. **EX** f is true in a state t if and only if there exists a state s such that g is true in state s, and there exists a transition from t to s:

  $$t \models \mathbf{EX}\ g \text{ iff } \exists\, s\ [\ g\langle s\rangle \wedge N(t, s)\ ]$$

  Where $g\langle s\rangle$ means the value of formula g in state s. This value can be computed using the existential quantification algorithms described previously. $g\langle s\rangle$ is true if and only if $s \models g$. However, this operation occurs frequently, and it is important to compute it in an efficient manner; efficient algorithms for this purpose are discussed in [8].
- If f is **E**[g **U** h], the BDD that represents the states where **E**[g **U** h] is true can be computed by iterating:

  $$\mathbf{E}[g\ \mathbf{U}\ h] = h \vee (g \wedge \mathbf{EX}\ \mathbf{E}[g\ \mathbf{U}\ h])$$
- If f is **EG** g, the algorithm is defined in a similar way. It searches for the greatest fixpoint **EG** g instead, and uses the following formula:

  $$\mathbf{EG}\ g = g \wedge \mathbf{EX}\ \mathbf{EG}\ g$$
- All other CTL operators are written in terms of the ones presented.

2.1.3 Verus and Symbolic Model Checking

Verus shares many important characteristics with symbolic model checking, such as the use of symbolic algorithms implemented by BDDs. The model is represented in a similar way, and the quantitative algorithms have similar fixpoint characterizations. The main differences are in the way the system is specified, and in the types of results produced by the algorithms.

Most model checkers use specification languages that have not been designed to simplify the specification of real-time systems. In some systems, such as SMV, the specification language simplifies the description of synchronous circuits. Writing a real-time program in such languages is difficult and error-prone. Too much time is spent trying to accommodate the system into the language constructs. Other verifiers use languages better suited for describing timing constraints such as Esterel, but they usually have an unfamiliar syntax. Too much time may be spent learning the language and in trying to understand how to represent the desired features of the system. The Verus language, on the other hand, uses a syntax that is familiar to most real-time system designers. It also has constructs that make it easy to express timing characteristics. Because of this, it is better suited to specify the real-time systems that will be verified than most languages used by other model checkers.

The most important differences, however, are the results produced and their analysis. Model checking concentrates on determining if events will or will not happen at some point in the future. This information is essential in asserting the correctness of a model, but it is not sufficient to assure predictability of a real-time system. Verus overcomes this limitation by concentrating on the determination of time bounds between events. This information provides important insight into the behavior of the system. The quantitative information produced by Verus cannot be easily produced by a standard model checker or reachability system, yet it is vital in determining the correctness of a real-time system.

2.2 Rate Monotonic Scheduling Theory

Computers have been used in critical situations for a long time. Their ability to react faster than humans and to perform under dangerous conditions was foreseen early on. However, it was also realized early on that the behavior of a computer system is not always simple to predict. In some cases the interaction between the various components in the system can cause a response to be delayed unexpectedly, making the system unpredictable. However, unpredictable behavior is not acceptable in critical applications. Because of this a very conservative approach was usually taken, by designing the system to support a significantly larger load than expected. The rationale was that if enough computing power is given to the application, the response time would be acceptable even for cases in which unexpected delays occur. The erroneous but still common idea that being real-time is the same as being fast may have originated at this time.

This approach is incorrect, however. In some cases it may work, because sometimes unexpected delays are bounded. However, there may be situations in which unbounded delays can occur caused by problems such as priority inversion [66]. In this case a response may never be produced. But the approach of designing the system to support larger loads is not a good one, even in the cases where it leads to correct results. It is not scalable, and it is very expensive in general. In order to be able to design predictable systems it is necessary to use methods that determine a priori if the system will meet its timing requirements. These methods must guarantee that the system is predictable, even if not necessarily fast.

2.2.1 The Liu and Layland Theory

Liu and Layland presented one of the first techniques that addressed analytically the problem of determining the predictability of a real-time system [59]. The original rate monotonic scheduling theory is a subset of their work. In this approach a real-time system is given by a set of tasks that execute periodically on a single processor. They have shown how to schedule task execution in order to guarantee that the timing requirements will be satisfied, and to optimize resource utilization. Their method assumes that:

- All tasks are periodic, that is, they execute once every *t* time units, where *t* is a parameter of the task. The period of a task is a time interval of length *t* such that the union of all periods partition the time line starting at time 0. Tasks execute once every period, and are ready to execute at the start of each period. They have known, deterministic execution times.
- The deadlines for each task are at the end of the period, that is, every execution *must* finish by the end of the period. If the deadline for *any* period of *any* task is not met, an error condition occurs and the system becomes unschedulable.
- Tasks do not suspend themselves during execution.
- Tasks are independent and can be preempted instantaneously. Preemption overhead is assumed to be zero, that is, the context switch time is assumed to be negligible.
- There is no synchronization between tasks.

Under these assumptions Liu and Layland studied both static and dynamic scheduling algorithms. Static scheduling assigns a fixed priority for each task, this priority does not change. Under dynamic scheduling priorities may change in time.

Dynamic Scheduling

An example of a dynamic priority algorithm is the earliest deadline first algorithm. Under the earliest deadline first algorithm, the process that has the closest deadline is given the highest priority. This algorithm guarantees schedulability of task sets that utilize the processor up to its full capacity. However, some practical problems associated with dynamic scheduling have not been solved such as its behavior under transient overload, scheduling of aperiodic tasks and priority granularity in communication scheduling [49]. For this reason, static scheduling algorithms are more popular than dynamic scheduling algorithms.

Static Scheduling

The rate monotonic scheduling algorithm is an example of a static priority algorithm. It assigns higher priorities to tasks with shorter periods. It is an optimal static priority algorithm in the sense that if a task set can be scheduled using some static priority algorithm, it can be scheduled using the rate monotonic scheduling algorithm.

Liu and Layland derived a sufficient condition for a task set to be schedulable by the rate monotonic algorithm under the assumptions given above. A set of n periodic tasks τ_1, $\tau_2, \ldots, \tau_n$ is characterized by a period t_i and an execution time c_i for each task. The formula $u_i = c_i / t_i$ gives the percentage of the time task τ_i is utilizing the processor. The formula $U = u_0 + u_1 + \ldots + u_n$ gives the total processor utilization for the task set. If a task set with n tasks has total processor utilization of at most $n(2^{1/n}-1)$ then its schedulability is *guaranteed* under rate monotonic scheduling. For large values of n, this bound converges to $\ln 2 \approx 0.693$. This is a sufficient, but not necessary condition; some task sets with utilization higher than this bound *can* be schedulable.

Another important result is shown in [59]. It states that the longest response time for any invocation of task τ_i occurs when all tasks start executing simultaneously. The time when all processes request execution is called critical instant. This result can be used to check schedulability in some cases where the total utilization is higher than the bounds described above. It is possible to guarantee that the task set is schedulable by assuming that all tasks start execution at the same time, and checking if the deadline of the first instantiation of each task is met.

2.2.2 Extensions of the Liu and Layland Theory

The assumptions made by Liu and Layland are rather restrictive. Many interesting real-time systems cannot be described using their method. Moreover, the worst case bound of 0.693 is very pessimistic in general. Much research has been done to extend the theory. This section will briefly overview some of this research. A more detailed presentation can be found in [53].

Deadline not equal to Period

The situation in which the deadline of a task is not equal to its period was first considered by Leung and Whitehead [54] in 1982. They introduced the deadline monotonic algorithm, in which priorities are assigned inversely with respect to task deadline. They have shown that this algorithm is optimal when the deadline is smaller than the period, and that a task set is schedulable by this algorithm if the first instantiation of each task after a critical instant meets its deadline.

In [50,51,64] the case in which the deadline and the period of each task are harmonic has been analyzed. It has been shown that under these conditions the rate monotonic and the deadline monotonic algorithms are the same. A sufficient schedulability condition similar to Liu and Layland's is given in [53].

Exact Schedulability Analysis

The schedulability test previously described is sufficient, but not necessary. There are task sets that are schedulable, but fail the test. An important result is the exact analysis of the schedulability of a task set, presented by Lehoczky, Sha and Ding [52] and by Joseph and Pandya [48]. Let a periodic task set $\tau_1, \tau_2, \ldots, \tau_n$ be given in priority order (τ_1 is the highest priority task). Under the critical instant assumption, let $W_i(t) = \Sigma_{j=1}^{i} C_j \lceil {}^t/_{Tj} \rceil$ be the cumulative demand for processing by tasks τ_j, $1 \leq j \leq i$, during the interval $[0, t]$ (C_j is the execution time of task τ_j, and T_j is its period). In other words, $W_i(t)$ is the demand for processing by all tasks of priority higher than or equal to τ_i. The task τ_i meets all its deadlines if it meets the deadline of its first instantiation under the critical instant. This occurs if $W_i(t) = t$ for some time t before the deadline of τ_i. Therefore, the task set is schedulable if this condition holds for all tasks: $\forall i\ \exists t\ W_i(t) \leq t$. This result can also be used to show that the rate monotonic algorithm can schedule task sets with up to 100% utilization when the periods are harmonic.

The exact schedulability analysis also studies the average case behavior of the rate monotonic algorithm. They have shown that when periods are drawn from a uniform distribution with a sufficiently wide range of values, task sets can be scheduled with total processor utilization as high as 88 to 92% on average. Their analysis, however, is beyond the scope of this presentation, details can be found in [52].

Task Synchronization

Synchronization of real-time tasks suffers from a serious problem, *priority inversion*. This happens when a high priority task is ready to execute, but is blocked by a lower priority task, usually because of synchronization constraints [11,66]. Consequently, the rate monotonic theory assumes that no synchronization between tasks occurs. However, by studying the priority inversion problem, it is possible to include task synchronization in the rate monotonic theory in a consistent way.

A typical scenario in which priority inversion occurs is as follows: A high priority task *A* requests access to a shared resource, which is held by a lower priority task *C*. It is then blocked until *C* releases the resource. However, a task *B*, with priority higher than *C* but lower than *A* might start executing, blocking *C* (and consequently *A*) indefinitely. This is called unbounded priority inversion.

The problem in this case is that *C* is blocking *A*, but it is still executing at its lower priority. A solution to the problem is to make the lower priority task *inherit* the higher priority whenever blocking the corresponding task. This protocol is called priority inheritance, and eliminates the unbounded priority inversion, making it bounded. One can then compute the maximum priority inversion time as the longest possible access of the shared resource by C, and incorporate it into the worst case execution time of A. Notice that priority inversion cannot be completely eliminated in the presence of synchronization, and no inherent inefficiency is introduced by this protocol.

The priority inheritance protocol bounds the maximum priority inversion time, but it does not avoid deadlocks. For this reason, the priority ceiling protocol was introduced. The priority ceiling of a shared resource is defined as the priority of the highest priority task that may access this resource. The priority ceiling protocol blocks a task trying to access a resource S if its priority is not higher than the priority of S^*, the highest priority resource currently being accessed by another task. Whenever accessing a resource, a task inherits the priority of the highest priority task it blocks. The priority ceiling protocol has the same advantages of the priority inheritance protocol, and in addition, it prevents deadlocks. More details on these protocols can be found in [66].

Aperiodic Tasks

In Liu and Layland's work, all tasks are periodic. Scheduling a mixture of periodic and aperiodic tasks can be done, however, using the concept of aperiodic servers. Aperiodic servers are periodic tasks that provide a resource for the exclusive use of aperiodic tasks, which can be used on demand. To provide fast aperiodic response times, the server is given a high priority. Two different types of aperiodic servers have been defined, the deferrable server [71] and the sporadic server [70].

The deferrable server algorithm creates a periodic server task with execution time C, period T and priority defined by the rate monotonic algorithm. This task has its entire period within which it can use up to C units of execution to service aperiodic tasks. At the end of the period any unused portion of C is discarded. The server capacity is renewed at the beginning of the next period. By being assigned high priority, the server provides guaranteed response times for high priority aperiodic tasks, while maintaining predictability for periodic tasks.

The sporadic server differs from the deferrable server in its replenishing policy. It preserves its execution capacity until an aperiodic request occurs. This request then receives immediate service, using part of the server's capacity. The server's capacity is replenished according to the priority it is executing at. Whenever that priority level becomes active, the replenishing time is set to the current time plus the server's period. At that point the server capacity is replenished with the server's execution time consumed since the last time its priority level became active.

The advantages of the sporadic server over the deferrable server are that it can be treated just like an ordinary periodic task for schedulability tests, it can run at any priority, and several servers at different priority levels can be defined to handle different types of aperiodic traffic. The analysis of the task set can be performed in a straightforward way, because a periodic task τ_i can be transparently replaced by an equivalent sporadic server for schedulability purposes [53].

2.2.3 Verus and the Rate Monotonic Theory

Both Verus and the rate monotonic theory are methods designed to determine time bounds between system events. However, they use very different approaches to achieve that end. The RMS theory assumes that the system being verified satisfies certain assumptions about its behavior such as periodicity and synchronization constraints. Systems satisfying these assumptions behave in a more predictable way than generic systems. They can be analyzed using analytical methods, which then provide formulas that can predict the timing behavior of this restricted set of systems. Extensions to the original theory have expanded the class of systems that can be analyzed to include several types of systems that occur in practice.

However, limitations still exist due to the nature of the analysis performed. Many practical existing systems cannot be analyzed, such as systems with aperiodic activity, but in which no aperiodic server has been introduced. In some cases the system can be forced to satisfy certain constraints. For example aperiodic servers can be introduced in systems in which not all tasks are periodic. However, in many situations this is not possible or desirable: an existing implementation may not use aperiodic servers and it may not be possible to change it. Verus does not have such limitations; any Verus program can be verified. For example, an important class of real-time systems that have to be modified in order to be analyzed by RMS is distributed systems. Intermediate deadlines have to be imposed to make the system amenable to RMS [69]. In some cases it is not possible to modify the system in this way; in other cases intermediate deadlines can make the system less efficient. In Verus it is straightforward to analyze distributed systems, one example can be seen in Section 6.6.

Another difference between Verus and RMS is the type of timing properties that can be checked. RMS basically computes the maximum execution times of tasks in the system. Verus allows the determination of the timing relation between any two events in the system. It also allows a richer set of properties that include the maximum execution time as well as minimum execution time and the number of occurrences of events in any time interval. An example of a property that is simple to express in Verus, but not so simple in RMS is the maximum priority inversion time: Given a high priority process P_0, how much time is spent with lower priority processes during the time P_0 is requesting execution? This can be computed in Verus by counting the number of occurrences of the execution of lower priority processes on intervals between P_0 start and P_0 finish.

In spite of all these issues, RMS does have an important advantage over Verus. The algorithms used by RMS have a lower complexity than those used by Verus. This is expected, since the systems analyzed by RMS are well behaved; it is easier to predict their behavior. Some large systems that can be analyzed by RMS may generate Verus models that suffer from state explosion. The best approach seems to be to consider Verus and RMS as complementary tools, each having specific advantages and disadvantages.

Chapter 3 The Verus Language

3.1 Introduction

In order to verify the correctness of a system, this system must be described in a form amenable to the verification algorithms used. Many formal languages and notations exist for this purpose, each one suited to a specific domain[1]. For example, the rate monotonic scheduling algorithm accepts as input tasks execution times and periods. The system description is extremely abstract, but it is expressive enough to allow system analysis. Other languages, such as VHDL or SMV, take the opposite approach, providing a very rich and detailed description.

Most languages simplify the description of certain types of characteristics, while possibly complicating the expression of others. For example, the SMV language allows circuits to be described in a straightforward way. However, the description of sequential execution is not so simple. In fact, this observation led to the development of the *Verus* language, since real-time systems are often described using a sequential programming language.

[1] See section 1.3.2 for a more detailed comparison between several languages used in verification tools.

The main goal of the Verus language is to allow engineers and designers to describe real-time systems easily and efficiently. It is an imperative language with a syntax resembling that of the C language. Using a syntax similar to a well known language simplifies the description of complex programs in Verus, since programmers take advantage of previous knowledge and can master the tool faster. Forcing programmers to learn a new language discourages the use of the tool, and often means that less people will ultimately benefit from it.

The most important characteristic of Verus programs is time. Special primitives are provided for the expression of timing aspects such as deadlines, priorities, and time delays. These primitives make timing assumptions explicit. A different approach is taken by many other languages, such as C, that allow programs where timing assumptions are not clearly stated. As a result, the specification becomes ambiguous and difficult to prove correct. The approach taken in Verus makes the specification clearer and more complete.

The data types allowed are fixed-width integer and boolean. Nondeterminism is supported, which allows partial specifications to be described. This guarantees that even though the model has a finite number of states, a rich set of systems can be described. Language constructs have been kept simple in order to make the compilation into a state-transition graph as efficient as possible. Simple constructs allow the precise expression of the desired features, since complex constructs sometimes force unnecessary details into the specification. Smaller representations can then be generated, which is critical to the efficiency of the verification and permits larger examples to be handled.

3.2 Overview of Verus

This section provides an overview of the language by presenting a simple real-time program. This program implements a solution for the producer-consumer problem by bounding the time delays of its processes. No synchronization is needed if the time delays of producer and consumer are defined properly.

The code for the producer process is shown below. Variable p is a pointer to the buffer in which data is stored. The producer initializes its pointer p to 0 and the produce variable to *false*. It then enters a nonterminating loop in which items are produced at a certain rate. Line 9 introduces a time delay of 3 units, after which an item will be produced. Line 10 marks the production of an item by asserting produce. In line 11 the pointer is updated appropriately. Line 12 makes sure that the event produce is observed. It is needed because the state of a Verus program can only be observed at wait statements. If a wait is not introduced in line 12, line 13 would cancel the effect of the assertion of produce before it can be observed. This behavior is discussed in detail later.

```
 1  producer(p)
 2  int p;
 3  {
 4    boolean produce;
 5
 6    p = 0;
 7    produce = false;
 8    while(!stop) {
 9      wait(3);
10      produce = true;
11      p = p+1;
12      wait(1);
13      produce = false;
14    };
15  }
```

Figure 3. Producer code

Wait Statements

An important feature of Verus is illustrated in line 9. In Verus time passes only on `wait` statements. Lines 6, 7 and 8 execute in time zero and time elapses only after the loop condition has been tested. This feature allows a more accurate control of time, and eliminates the possibility of implicit delays influencing the results of the verification. It also generates models with fewer states, since contiguous statements are collapsed into one transition. Notice that this feature affects the behavior of the program significantly. For example, a block of code not containing the `wait` statement executes atomically.

Nondeterminism

To illustrate another characteristic of Verus, let's assume that the `producer` is not required to actually produce an item after 3 time units, but may instead leave the value of `p` unchanged. This can be modelled in Verus by changing line 11 to:

```
11      p = select{p, p+1};
```

The `select` statement introduces a nondeterministic choice in the program. The value of `p` after executing `select` can be either `p` or `p+1` (addition in Verus is defined modulo the maximum value for the variable). These choices can characterize the fact that the producer *may* produce an item, but it may also *not* produce it. This way we can model both possibilities without having to specify all the details that are actually needed to decide between these two options. Besides hiding unnecessary details, nondeterminism can be used to verify partial specifications. Whenever the value of a variable hasn't been determined by the design, nondeterministic constructs can specify all possible values the variable could take. This approximates the behavior of the actual system by exploring all possibilities. As the design process evolves, the values can be restricted until the correct behavior is determined. Nondeterminism encourages the use of automated verification in earlier phases of the design. Components of the system can be validated before all modules have been specified. In this way errors can be uncovered before propagating to components added later in the design.

```
consumer(p, c)
int p, c;
{
  boolean consume;

  c = 0;
  consume = false;
  while (!stop) {
    wait(1);
    if (p != c) {
      consume = true;
      c = c + 1;
      wait(1);
      consume = false;
    };
  };
}
```

Figure 4. Consumer code

The `consumer` process is very similar to the `producer`. The basic differences are that it waits for less time before consuming, and that it only consumes if `p` and `c` have different values (`p == c` signals an empty buffer). Notice that the `producer` does not check if the buffer is full before inserting another item. The time delays of both processes guarantee that an overflow will never occur.

The `main` function

As in the C language, `main` has a special function in Verus. In this function all processes are instantiated, and global variables can be declared. The variables `p` and `c` (used as pointers in the buffer) are declared and the `producer` and `consumer` processes are instantiated in the `main` function of the example code.

Process instantiation in Verus follows a synchronous model. All processes execute in lock step, with one step in any process corresponding to one step in the other processes. Asynchronous behavior can be modeled by using *stuttering*, as described in section 6.1. An implicit instantiation of the `main` module is assumed, where the code in `main` executes as another synchronous module.

Specifications can also follow the code as can be seen. The specifications below compute the minimum and maximum time between producing an item and consuming it, as well as checking that a `produce` is always followed by a `consume`.

```
main()
{
  int p, c;

  process prod producer(p, c),
          cons consumer(p, c);

  spec MIN[prod.produce, cons.consume]
       MAX[prod.produce, cons.consume]
       AG(prod.produce -> AF cons.consume)
}
```

Figure 5. Producer/consumer main function

Periodic Execution.

To illustrate different features of Verus some extensions to the program above are considered. The first comes from realizing that both processes will always execute, even when no data exists. For example, even if the `producer` does not generate items, the `consumer` will execute. This can be avoided using *periodic execution*, where execution is scheduled at specific points in time. It can be specified in Verus very easily. The `producer` process can be made into a periodic process executing once every 10 time units as seen in figure 6.

The periodic statement has four parameters, the last being the code that will be executed periodically. The first parameter is the *start_time*, which specifies how many time units the periodic code will idle *before* starting its execution for the first time. In this example it will start immediately. The second parameter is the period. In this case the statements following periodic will execute once every 10 time units. The third parameter defines a deadline. It states that the execution must finish in less than 10 time units or an exception will be raised (exception handling is discussed below). Execution may take longer than the sum of the waits because of synchronization with other processes.

```
producer(p, c)
int p, c;
{
  boolean produce;

  p = 0;
  produce = false;
  periodic(0, 10, 10) {
    wait(3);
    produce = true;
    p = p+1;
    wait(1);
    produce = false;
  };
}
```

Figure 6. Periodic producer

Deadlines

Verus also allows the definition of deadlines independent of periodicity. For example, we can specify that producer will be an aperiodic process, but that it must finish each iteration in less than 10 time units:

```
producer(p, c)
int p, c;
{
  boolean produce;

  p = 0;
  produce = false;
  while(!stop) {
    deadline(10) {
      wait(3);
      produce = true;
      p = p+1;
      wait(1);
      produce = false;
    };
  };
}
```

Figure 7. Aperiodic producer

Exceptions

Exception handling is used to control abnormal situations. The only exception currently defined in Verus is a *missed deadline*. It occurs when the code inside a `deadline` or a `periodic` statement does not finish within the specified time. An exception handler must be specified for the exception to take effect. If no exception handlers are defined, the exception is ignored. Whenever a deadline is missed the code designated as handler is executed. After the execution of the exception handler the rest of the code inside the deadline scope is ignored. Control is then passed to the statement following the deadline statement. This could be the next instantiation of a periodic process when the exception occurs inside a `periodic` statement or the code after a `deadline` statement.

```
producer(p, c)
int p, c;
{
  boolean produce;

  handler {
    error = 1;
  } for {
    p = 0;
    produce = false;
    periodic(0, 10, 10) {
      wait(3);
      produce = true;
      p = p+1;
      wait(1);
      produce = false;
    };
  };
}
```

Figure 8. Exception handling

Figure 8 shows the typical exception handling mechanism. Whenever a deadline is missed an error flag is asserted. The verification procedure can then check to see if the error condition is reachable. In some other applications, however, a different behavior may be the desired one. For example, a multimedia program might choose to ignore some image frame that hasn't arrived, provided the last *n* arrived. An exception handler to model this behavior can be easily written: whenever an exception occurs, the handler would record the number of the frame missed. If the current missed frame is at least *n* frames apart from the previous one, no error would be issued. Otherwise an error condition would be asserted.

Priorities

Priorities can also be described in Verus in a straightforward manner. In the same way that the constructs shown above encapsulate the code they apply to,

```
priority(3) {
              ...
          }
```

can be used to make the `producer` process run at priority level 3. If, for example, the `consumer` runs at priority 2, then `the producer` will be given priority over it during execution.

Internal and External Variables

There are two types of variables in Verus, *internal* and *external*. In both cases, there is no default value for variables in Verus. Unless assigned a specific value, the value of a variable is chosen nondeterministically from all possible values (*true* or *false* for booleans and $0..2^{\text{width}}$-1 for integers). The two types differ, however, regarding the rules that control when their value can change. The value of an internal variable changes only when assignments are executed. External variables on the other hand model the interaction of the model with the environment. They correspond to inputs from the outside world, and the program has no control over their value. Assignments to external variables are not allowed and their value can change nondeterministically at any transition of the model. The declaration of external variables is preceded by the `extern` keyword. Internal variables are declared without this keyword.

3.3 Verus Syntax

This section describes the syntax of Verus in more detail. Initially the basic statements and how they combine to form a function are described. A function, much like its counterpart in C, is a block of statements executed sequentially. Finally, it is explained how to instantiate processes using functions, and how to compose several processes in parallel. The syntax presented is abstract and unnecessary details have been omitted for clarity. In particular, lists are not formally defined. The name of an entity followed by the keyword *_list* is used to specify a list of entities of the type given. For example a list of identifiers is denoted simply by *identifier_list*. Unless otherwise specified, all lists are separated by spaces, tabs or newlines.

The Core Language

Verus has a core subset, which defines the main characteristics of the language. The remaining constructs are defined in terms of the core language.

- Functions

  function_definition ::=
  identifier `(` *identifier_list* `)` *declaration_list compound_statement*

Identifier lists are separated by commas.

- Declarations

  declaration ::= *type_specifier decl_identifier_list* `;` |
  `extern` *type_specifier decl_identifier_list* `;`

  type_specifier ::= `boolean` | `int`

  decl_identifier ::= *identifier* | *identifier* `:` *constant.*

Variables can be boolean and fixed-width integers. This guarantees that the model will be finite-state. The default integer has 8 bits. In a declaration, an identifier can be followed by the number of bits to override the default, e.g., `int c;` (8 bits) or `int c : 4;` (4 bits).

- Statements:

  compound_statement ::= `{` *statement_list specification_list* `}`

```
statement ::= compound_statement |
              expression_statement |
              selection_statement |
              iteration_statement |
              time_statement |
              null

expression_statement ::= ; | assignment_expression ;

selection_statement ::= if ( expression ) statement else statement

iteration_statement ::= while ( expression ) statement

time_statement ::= wait ( constant )
```

- Expressions

```
assignment_expression ::=  identifier = expression |
                           identifier = select { expression_list }

expression ::= expression || expression |
               expression && expression |
               ! expression |
               primary_expression

primary_expression ::= ( expression ) |
                       identifier |
                       constant
```

Only boolean expressions are defined in the core language. Integers variables and operations are defined in terms of booleans using binary encoding. The `select` expression allows for a nondeterministic choice of values. The value returned is one of the possible values listed, chosen nondeterministically. Expression lists are separated by commas.

- Specifications

specification ::= `spec ( `*ctl_formula*` ) ;` |
`min ( `*expression*`, `*expression*`) ;` |
`max ( `*expression*`, `*expression*`) ;` |
`mincount ( `*expression*`, `*expression*`, `*expression*`) ;` |
`maxcount ( `*expression*`, `*expression*`, `*expression*`) ;`

Extensions to the Core Language

The constructs described below can be defined in terms of the previous ones. They simplify Verus programs, but do not add to the expressive power of the language. We only present the additions to the definitions given previously.

- Statements:

  statement ::= *nondeterministic_statement* | *schedule_statement*

  selection_statement ::= `if ( `*expression*` ) `*statement*

  nondeterministic_statement ::= `select` *compound_statement*

  schedule_statement ::=
  `periodic ( `*constant*`, `*constant*`, `*constant*` ) `*compound_statement* |
  `deadline ( `*constant*` ) `*compound_statement* |
  `handler` *compound_statement* `for` *compound_statement*

- Expressions:

  expression ::= *boolean_expression* | *relation_expression* | *int_expression*

  relation_expression ::= *expression* = = *expression* | *expression* `!=` *expression* |
  expression < *expression* | *expression* > *expression* |
  expression <= *expression* | *expression* >= *expression*

int_expression ::= *expression* + *expression* | *expression* – *expression* |
expression * *expression* | *expression* / *expression*

Process instantiation

Process instantiation occurs in the `main` function, using the `process` keyword.

statement ::= `process` *f_instantiation_list*

f_instantiation ::= *identifier function_name* (*identifier_list*)

Function instantiation lists and identifier lists are separated by commas.

Chapter 4 The Semantics of Verus

This chapter describes the meaning of Verus programs, or in other words, their expected behavior. It shows how each construct in Verus relates to the state-transition model used and how this graph is constructed from a Verus program. Understanding how Verus programs are translated into state-transition graphs is essential in order to be able to model the system and to interpret the results of the verification correctly.

The meaning of a Verus program is a state-transition graph. Section 4.1 explains how state-transition graphs are represented in Verus. In section 4.2 the concept of *wait graphs* is introduced. Wait graphs are an abstraction used to keep track of the control flow of the program. The formal semantics of the core language is described in section 4.3, while section 4.4 presents the semantics of the extensions to the core language. Finally, section 4.5 discusses the semantics of concurrent processes in Verus.

4.1 State-Transition Graphs in Verus

The state-transition graph constructed from a program P is $G_P = (S_P, I_P, T_P)$, where S_P is the set of states, I_P is the set of initial states and T_P is the transition relation. The set of states is defined by the variables in the program. Each possible assignment to the variables is a state. I_P and T_P are defined by the program as will be seen shortly.

Symbolic Representation

States are defined by the assignment of values to program variables (we assume that different states have different values for the variables, as described in [62]). Each possible assignment to the program variables is a state. For example, if the program has three boolean variables a, b and c, examples of states are $(\bar{a},\bar{b},c)$, $(\bar{a},b,c)$ and (a,b,c), where, for variable v, v means the variable is true in the state, and $\bar{v}$ means the variable is false. Boolean formulas over program variables can be true or not in a given state. The value of a boolean formula in a state is obtained by substituting into the formula the values of the variables in that state. For example, the formula $(a \vee c)$ is true in all states shown above. The graph representation used by Verus is a direct consequence of this observation. Sets of states are represented by boolean formulas, where each formula represents the set of states in which the formula is true. For example, the formula *true* represents the set of all states, the formula *false* represents the empty set of states, and the formula $(a \vee c)$ represents the set of states in which a or c are true. Because symbols are used to represent states, algorithms that use this method are called *symbolic algorithms*.

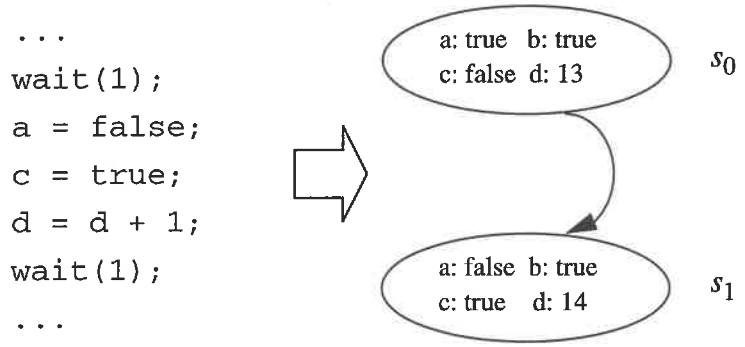
Transitions can also be represented by boolean formulas. A transition $T(s, s')$ is represented using two sets of variables, one for the current state and another for the next state. Each variable in the next state set corresponds to one variable in the current state set. If state s is represented by the formula f_s over the current state variables, and state s' is represented by formula $f_{s'}$ over the next state variables, then the transition $T(s, s')$ is represented by the formula $f_s \wedge f_{s'}$. For example, a transition from state (a,b,c) to state (a,b,c) is represented by the formula $\neg a \wedge \neg b \wedge \neg c \wedge \neg a' \wedge b' \wedge \neg c'$. The transition relation of a graph is the disjunction of all transitions in the graph. The meaning of the formula representing the transition relation is the following: there exists a transition from s to s' iff the substitution of the variable values for s in the current state variables and s' in the next state variables of the transition relation yields *true*.

In the same way that boolean formulas represent *sets* of states, they also represent *sets* of transitions. In general, a formula can represent many transitions, making the symbolic representation usually much smaller than an explicit enumeration of all transitions. This technique, also used in [62], is one of the reasons for the efficiency of symbolic algorithms.

The symbolic representation relies on the fact that states are represented by the values of the atomic propositions in those states. In order to guarantee that states can be identified uniquely, we must make the assumption that different states have different labeling of propositions. More formally, we assume that for any two states s_1 and s_2 in S, if $L(s_1) = L(s_2)$ then $s_1 = s_2$. This assumption does not, however, impose any restrictions on the model, since extra atomic propositions can be added in order to make $L(s_1) \neq L(s_2)$ for distinct states s_1 and s_2 [62].

4.2 Tracking the Control Flow — Wait Graphs

The execution of a Verus statement may change the value of one or more program variables. In general, it changes a given state s into another state s'. Executing a sequence of statements in a given state then generates a sequence of states. However, in Verus not all of those states are observable. The state of the program can only be observed at `wait` statements. When a `wait` is executed all changes caused by the execution of the block of statements since the previous `wait` take effect at the same time. Transitions in the graph occur only when `wait` statements are executed.



```
...
wait(1);
a = false;
c = true;
d = d + 1;
wait(1);
...
```

Figure 9. Wait statement example: if s_0 is the current state at the first `wait`, s_1 will be the current state at the second.

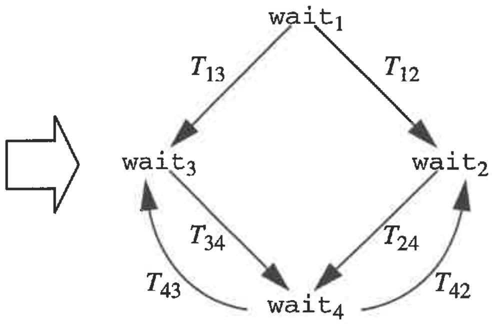
Each transition in the graph corresponds to time elapsing by one unit. The statement `wait(1)` corresponds to one transition in the graph. Longer waits are modeled by a sequence of unit transitions. This allows the programmer to specify exactly when time passes, and permits a more accurate model of time than possible if each statement takes one time unit to execute.

It is easier to understand the behavior of a Verus program by concentrating on its `wait` statements. This can be accomplished by translating the program into a *wait graph*. The wait graph corresponding to a Verus program is a graph in which the states are the `wait` statements in the program. It corresponds to an intermediate representation between the Verus program and the corresponding state-transition graph. It is used only to illustrate how this translation occurs and is not actually constructed. In the discussion below, to differentiate between distinct waits, $\texttt{wait}_i$ represents the i^{th} occurrence of a `wait` statement in the program. Subscripts have been added to the sample program below to aid the presentation, but no subscript exists in actual programs.

As discussed, each `wait` in the program is a state in the wait graph. Transitions between `waits` are defined as follows. A transition between $\texttt{wait}_i$ and $\texttt{wait}_j$ exists iff $\texttt{wait}_j$ can be reached from $\texttt{wait}_i$ in the control flow of the program without going through intermediate `waits`. The edges of the wait graph are labelled by a relation T_{ij} between any two states in the state-transition graph. This relation describes the state changes caused by the execution of the statements between the corresponding `waits`. The construction of this relation is described in the next section. Intuitively, given two states s and s', $T_{ij}(s, s')$ means that if the execution of the program is in $\texttt{wait}_i$ and the current state is s, then there exists a path in the control flow leading to $\texttt{wait}_j$ (without intermediate waits) and the execution of the statements on this path will change state s into state s'.

Notice that T_{ij} represents exactly all transitions from s to s' in the state graph such that s and s' are respectively the current state of the program before and after control is transferred from $\texttt{wait}_i$ to $\texttt{wait}_j$. This makes it possible to construct the state transition graph that corresponds to a given Verus program from its wait graph. The set of all relations between `wait` statements represents all transitions in the program. Consequently, the disjunction of all such transitions constitutes the transition relation of the state-transition graph of the program.

```
wait1(1);
S1;
while cond1 {
  if cond2 {
    S2;
    wait2(1);
    S3;
  } else {
    S4;
    wait3(1);
    S5;
  };
  S6;
  wait4(1);
};
```



(S_is correspond to blocks of assignments)

Figure 10. A Verus program and its corresponding wait graph

Wait Counters

Since each relation T_{ij} corresponds to a set of transitions, their disjunction should correspond to the transition relation of the program. However, this is not true because T_{ij} does not contain information about where it came from ($\texttt{wait}_i$) and where it leads to ($\texttt{wait}_j$). The disjunction of all relations would not maintain consistency of the values of variables after the execution of a sequence of `wait`s.

This problem is solved by creating an extra variable in the program to record this information, the wait counter `wc`. Each `wait` statement is preceded by an assignment `wc` = i, where i is the occurrence number of the `wait` statement (this assignment is introduced by the compiler; it is not part of the source code). The relation T_{ij} now contains information about where it leads to, since the assignment wc $= j$ is introduced before wait$_j$. As detailed in the next section, the previous value of the wait counter indicates where this transition came from. Now T_{ij} has all information needed to maintain consistency across sequences of wait statements, and the disjunction of all relations between waits is the transition relation of the program.

Determining the Initial State Set

The initial state set of a Verus program is the state it reaches just after executing the first wait. In order to compute the initial state set, Verus programs start with an initial wait, with the wait counter of 0 (introduced by the compiler). The state of the program at this point, S_0, is represented by the formula $(wc = 0)$. The initial state set is defined as the set of states reached from S_0 in one transition.

In S_0 all variables (with the exception of wc) have nondeterministic values. In the initial state only those that have been assigned before the first wait in the program will have fixed values. The difference between defining the initial state set as S_0 or as the states reached from S_0 in one step is a subtle consequence of this fact. The difference can be seen in the program below:

```
main()
{
  boolean a;

  a = true;
  while(true) {
    wait(1);
  };
}
```

Figure 11. Initial state set example

The specification (**AG** a) is false if the initial state set is S_0, because the state set represented by $(wc = 0 \wedge \neg a)$ is part of S_0, that is, a could be false in the initial state. However, this behavior can be confusing to programmers, since it is not intuitive from the program source. Defining the initial state set as the set of successors of S_0 solves this problem. For the program above the initial state set is $(wc = 1 \wedge a)$, which models the expected behavior.

4.3 Core Language Semantics

This section formally defines the meaning of a Verus program. It explains how the relations between `waits` are constructed, and formalizes the construction of the state graph that models a Verus program.

The state space of a Verus program is defined by a set of boolean variables. A state in the model is an assignment of values to the variables. The set of all states is *ST*. A relation between any two states belongs to $Relation \equiv Powerset(ST \text{ x } ST)$.

The semantics is defined as a function of the program text. The meaning of a program is a transition relation between states such that there is a transition from state s to state s' iff there exists an execution sequence leading from $\texttt{wait}_i$ to $\texttt{wait}_j$ without intermediate `wait` statements that changes state s into state s'.

The function R given below constructs the relations between `wait` statements. Intuitively, given a relation r describing the program until the execution of statement *Stmt*, the function R will produce the relation r' describing the program after executing *Stmt*. The function R also constructs another relation t by accumulating the relations constructed for all `wait` statements. Function R is defined by

$$R\text{: STMT} \rightarrow Relation \text{ x } Relation \rightarrow Relation \text{ x } Relation$$

where pairs of relations are $\langle r, t\rangle$, r being the relation containing changes to the program state since the last `wait` statement, and t being the transition relation of the program, that is, the disjunction of the relations between all pairs of `wait` statements.

The state-transition graph corresponding to a program P is constructed as follows. Given program P, function R constructs $\langle r, t\rangle = R[\![P]\!]\langle wc = 0, \varnothing\rangle$, where t is the transition relation of the state-transition graph corresponding to P, and the initial state set is constructed from t as discussed above.

Additional definitions are needed before presenting the semantic functions:

- There are only boolean variables in the program. Integer variables are encoded in binary and substituted for the corresponding boolean variables.
- V and V' are two sets of boolean variables such that for each variable `v` in the program there are corresponding variables $v \in V$ and $v' \in V'$. The variable $v \in V$ represents the value of the program variable `v` in the current state, and the variable $v' \in V'$ represents its value in the next state. A transition is a relation between variables in V and V'.
- A variable wc is introduced in the model. It is used to keep information about the previous and next `wait` statements in the control flow. An assignment `wc = i;` is assumed to exist just before statement $\texttt{wait}_i$.
- An initial `wait` (with wait counter value of 0) is the second statement of the program, preceded by an assignment `wc = 0`.
- All programs are assumed to have as the last statement:

  ```
  while (true) wait(1);
  ```

  There are two reasons for this requirement. Transitions are only generated at `wait` statements (see $R[\![$ `wait` $]\!]$) and the existence of a final `wait` guarantees that transitions will be generated for all programs. Moreover, this loop also guarantees that even when the program terminates and no more statements are executed, the state of the program can still be observed. Intuitively the loop means that after the program terminates its state will remain unchanged.

4.3.1 Expressions

The meaning of a Verus expression is a boolean formula corresponding to the syntactic expression. Since the core language only allows boolean expressions, the translation is straightforward; it is described below by the function E:

Primary Expressions

$E[\![\texttt{true}]\!] = \mathit{true}$

$E[\![\texttt{false}]\!] = \mathit{false}$

$E[\![\texttt{v}]\!] = v'$, where $\texttt{v}$ is an internal variable and $v' \in V'$.

$E[\![\texttt{v}]\!] = v$, where $\texttt{v}$ is an external variable and $v \in V$.

Internal variables are represented by their *next* state value, while external variables are represented by their *current* state value. This choice of representation significantly affects the behavior of each type of variable, as described below.

First, lets consider how internal variables behave. All references to an internal variable will be denoted by its next state variable. For example, a reference to variable `v` on the left-hand side of an assignment (as in `v = false`) will be denoted by the next state variable v', and therefore the assignment will change the value of v' in the current relation (see semantics of assignments). This is the expected behavior, since an assignment determines the value of the variable in the next state.

However, other references to `v` (as in `x = !v`) also refer to v'. In the assignment `x = !v` the value of x' in the current relation will be assigned the negation of the value of v'. Two cases must be considered. If variable `v` has been assigned a value previously, this assignment has updated the value of v' in the current relation. Consequently, the assignment to `x` uses the most recent value assigned to `v`. To illustrate how this affects the behavior of the program, consider the program fragment below. The value assigned to `x` in line 4 is *false*, because in the current relation of the program at line 4 the value of variable v' is *true*.

```
1   v = false;
2   wait(1);
3   v = true;
4   x = !v;
```

Figure 12. Example of the behavior of internal variables

In the case where variable v has not been assigned any value, the current relation enforces that the value of internal variables do not change via the clause ($\bigwedge_{v \in \text{internal variables}} v = v'$) introduced in the current relation at wait statements (see R[[wait]]). This clause guarantees that the current and next state variables of internal variables have the same value (the clause is automatically overridden if an assignment is made). This has the effect that the value of an internal variable does not change if no assignments are made. This is true even across waits. For example, if line 3 did not exist in the fragment above, the value of v' would be *false* in the current relation because of this clause, and therefore the value assigned to x would be *true*.

External variables, on the other hand, are not included in the wait statement clause introduced in the current relation. This is because their value is not maintained across wait statements. External variables may change value nondeterministically at wait statements and they cannot be assigned to. The value an external variable has at any point in the program is the value it had in the previous wait statement, since no assignments exist. This value is represented by its current state variable.

Boolean Expressions

$E[\![\ expr_1 \ \texttt{||}\ expr_2\ ]\!] = expr_1 \vee expr_2$

$E[\![\ expr_1 \ \texttt{\&\&}\ expr_2\ ]\!] = expr_1 \wedge expr_2$

$E[\![\ \texttt{!}\ expr\ ]\!] = \neg\, expr$

4.3.2 Statements

Assignments

$$R[\![\ \texttt{v = } expr\ ]\!]\langle r, t\rangle = \langle(\exists\, y\ [\ v = Expr\ {}^{y}/_{v} \wedge r\ {}^{y}/_{v}]),\ t\rangle,$$

where $v = E[\![\ \texttt{v}\ ]\!]$, $Expr = E[\![\ expr\ ]\!]$ and y is a new variable.

This expression computes the strongest post-condition for the assignment `v = ` *expr* given r as a pre-condition. If r is the set of valid transitions in the graph since the last `wait` statement, the expression above determines the largest set of transitions that satisfy the assignment and that satisfy r for variables other than v. Intuitively, this expression substitutes the previous value of v in r for *Expr*, while maintaining the values of other variables.

$$\begin{aligned} R[\![\ \texttt{v = select\{}expr_1\texttt{, }expr_2\texttt{\}}\ ]\!]\langle r, t\rangle = \textbf{let}\ &\langle r', t\rangle = R[\![\ \texttt{v = } expr_1\ ]\!]\langle r, t\rangle, \\ &\langle r'', t\rangle = R[\![\ \texttt{v = } expr_2\ ]\!]\langle r, t\rangle\ \textbf{in} \\ &\langle r' \vee r'', t\rangle \end{aligned}$$

The relation for a nondeterministic assignment is the disjunction of the expression for each possible assignment. In other words, a nondeterministic assignment is true if any possible value is assigned. The extension of R for the case in which more than two expressions exist is a simple extension of the disjunction shown, and is omitted for brevity.

Sequential Execution

$$R[\![\ \mathrm{S}_1\ ;\ \mathrm{S}_2\ ]\!]\langle r, t\rangle = R[\![\ \mathrm{S}_2\ ]\!](R[\![\ \mathrm{S}_1\ ]\!]\langle r, t\rangle)$$

Wait Statements

$$R[\![\ \texttt{wait}_i\texttt{(1)}\ ]\!]\langle r, t\rangle = \langle((wc = i) \wedge \textstyle\bigwedge_{v \in IV} v = v'),\ (t \vee r)\rangle,$$

where IV is the set of internal variables in the program.

Function R for the `wait` statement changes the previous relation in two ways. At this point in the program transitions that lead to $\texttt{wait}_i$ are generated. These transitions are represented by relation r before the `wait` is executed, which is then disjointed with the previous transition relation t. This is the only statement that changes the value of t.

Moreover, the current relation after the execution of $\texttt{wait}_i$ must reflect the fact that a new set of transitions will be computed. The new relation specifies that transitions start in $\texttt{wait}_i$ with the formula $(wc = i)$, that is, the wait counter value of the current state variable in the transition will be i. The destination of the new set of transitions will be established when the next `wait` statement is found. At that point the assignment $\texttt{wc} = j$ before $\texttt{wait}_j$ introduces the formula $(wc' = j)$ in the current relation, specifying where the transition leads to. Because of these two conditions, all transitions specify a value for both the current and next state wait counters.

Finally, it is necessary to introduce the expression $\bigwedge_{v \in IV} v = v'$ into the current relation. For internal variables this expression guarantees that unless assigned a new value, internal variables maintain their previous value across transitions. External variables may change value during transitions, and therefore are not included in this expression. This allows the use of the next state variable as the semantic value of internal program variables. Whenever an internal variable is referenced, its next state variable will have its previous value (via the clause $v = v'$ above) or its new value (via the assignment expression described previously) in the current relation.

The expression above handles only unit waits. Longer waits are modeled by a sequence of unit wait statements.

Conditionals

$$
\begin{aligned}
R[\![\ \texttt{if}\ cond\ S_1\ \texttt{else}\ S_2\ ]\!]\langle r, t\rangle = \ & \textbf{let}\ \langle r', t'\rangle = R[\![\ S_1\ ]\!]\langle (r \wedge cond), t\rangle, \\
& \qquad \langle r'', t''\rangle = R[\![\ S_2\ ]\!]\langle (r \wedge \neg cond), t\rangle\ \textbf{in} \\
& \langle r' \vee r'', t' \vee t''\rangle
\end{aligned}
$$

Each branch in the `if` statement is executed by restricting its parameter to the set of transitions that satisfy the appropriate conditional — S_1 receives those transitions satisfying *cond*, and S_2 receives transitions not satisfying *cond*. In this way, if control reaches the `if` statement through a state that satisfies the condition, control will proceed to S_1. If the state does not satisfy the condition, control proceeds to S_2. The representation of a conditional is the disjunction of the representation of its branches.

Loops

$$\begin{aligned} R[\![\, \texttt{while}\ cond\ S_1 \,]\!]\langle r, t\rangle = \ &\textbf{let}\ \langle r', t'\rangle = R[\![\, \texttt{while}\ cond\ S_1 \,]\!](R[\![\, S_1 \,]\!]\langle (r \wedge cond), t\rangle), \\ &\qquad \langle r'', t''\rangle = \langle (r\ \wedge \neg cond), t\rangle\ \textbf{in} \\ &\langle r' \vee r'', t' \vee t''\rangle \end{aligned}$$

The representation of a `while` loop can be seen as unrolling the loop into nested `if` statements: `if` *cond* `{`S_1`;  if` *cond* `{`S_1; ...`}};`. However, even though simple to understand, function R for a `while` statement cannot be computed as shown above, because it is circular. A more accurate definition can be seen below. It uses the *fix* operator, which returns the least fixpoint of the functional given as its argument.

$$\begin{aligned} R[\![\, \texttt{while}\ cond\ S_1 \,]\!] = fix(\lambda f\, \lambda\langle r,t\rangle.\ &\textbf{let}\ \langle r', t'\rangle = f(R[\![\, S_1 \,]\!]\langle (r \wedge cond), t\rangle), \\ &\qquad \langle r'', t''\rangle = \langle (r\ \wedge \neg cond), t\rangle\ \textbf{in} \\ &\langle r' \vee r'', t' \vee t''\rangle) \end{aligned}$$

The operations performed by the functional above are projection (from the result of the application of f into r' and t'), disjunction (of r', r'' and t', t'') and pairing (of the results of the disjunctions). Since these operations are continuous [76], any functional constructed from them is also continuous. By being continuous, the functional is also monotonic, and therefore it has a fixpoint.

However, not all programs with while statements have well behaved semantics. For example, a fixpoint characterization for the program below is the relation *false*, which corresponds to non-termination, as is expected from the program. But since there are no `waits` in the program, time does not pass. Non-termination in this case means that if the program is in state *s* when the code below is executed, *there will be no outgoing transition from s*, that is, the non-terminating behavior is not observable. In order to avoid this anomalous behavior, we impose the rule that all execution sequences inside all `whiles` in the program must execute at least one `wait` statement. This ensures that even non-terminating `while` programs are always observable and that no states without outgoing transitions will be created.

```
while(true) {
  a = !a;
}
```

Figure 13. A while program with a trivial fixpoint

Null Statement

$R[\![\, \texttt{null} \,]\!]\langle r, t\rangle = \langle r, t\rangle$

4.4 Verus Extension Semantics

If Statement

selection_statement ::= `if` (*expression*) *statement*

The if statement is a simple extension of the if-then-else statement, it is simply translated as: `if` (*expression*) *statement* `else null`

Non Deterministic Statement

nondeterministic_statement ::= `select` *compound_statement*

Verus Extension Semantics

The statement `select` {S_1; S_2;... ; S_n} has the intuitive meaning of a non deterministic choice of execution between the statements in the compound statement. It corresponds to:

```
extern int s;
...
if (s == 1) $S_1$ else
  if (s == 2) $S_2$ else
    ...
```

Schedule Statements

schedule_statement ::=
        `deadline` (*constant*) *compound_statement*

The deadline statement is translated into the Verus core language by creating an integer variable `timer`. At the `deadline` keyword an assignment `timer = 0` is inserted. Within the scope of the deadline, each `wait(`*n*`)` statement is preceded by `timer = timer +` *n*`;` and by a check `if (timer >= deadline)` *error_code*, where the exception handler defines *error_code*.

schedule_statement ::=
        `periodic` (*constant*, *constant*, *constant*) *compound_statement*

The periodic statement is handled in a similar way. The difference is that an infinite loop is inserted enclosing the periodic statement, and once the periodic statement has finished executing, a loop is inserted to enforce the periodicity:

```
while (timer < period) {
  timer = timer + 1;
  wait(1);
};
```

A similar loop is inserted before the main loop at the beginning of the periodic statement to account for the initial offset.

Exception Handling

> *schedule_statement* ::=
> `handler` *compound_statement* `for` *compound_statement*

The first compound statement is the exception handler, and the second is the scope of the handler. The exception handling statement `handler` S_1 `for` S_2 is translated by substituting the *error_code* created by deadline statements in S_2 for: S_1 `else {`. The compound statement S_1 is executed in case of a missed deadline, and the `else` clause guarantees that the rest of the `deadline` statement is skipped in case of a missed deadline. The `{` after the `else` is closed at the end of the deadline statement.

For example, the periodic producer described in the previous chapter is translated into the core language as seen below:

```
producer(p, c)
int p, c;
{
  boolean produce;

  handler {
    error = 1;
  } for {
    p = 0;
    produce = false;
    periodic(0, 10, 10) {
      wait(3);
      produce = true;
      p = p+1;
      wait(1);
      produce = false;
    };
  };
}
```

Figure 14. Original periodic producer

```
producer(p, c)
int p, c;
{
  boolean produce;
  int timer;

  p = 0;
  produce = false;
  while (true) {
    timer = 0;
    timer = timer + 3;
    if (timer > 10) {
      error = 1;
    } else {
      wait(3);
      produce = true;
      p = p+1;
      timer = timer + 1;
      if (timer > 10) {
        error = 1;
      } else {
        wait(1);
        produce = false;
      };
    };
    while (timer < 10) {
      timer = timer + 1;
      wait(1);
    };
  };
}
```

Figure 15. Periodic producer in the core language

Integer expressions and operations

These operations are translated into boolean equivalents by using a binary encoding of integers. Since all integers have fixed-width, this is a straightforward translation.

4.5 The Semantics of Concurrency in Verus

Up to this point we have seen how a single process in Verus is translated into the corresponding state transition graph. But more frequently than not real-time systems are described by a set of concurrent processes instead of a single one. It is possible to specify concurrent processes in Verus using the `process` keyword. This section describes how the behavior of parallel processes is defined in Verus.

Synchronous Composition

Given a set of processes defined by their state transition graphs, it is possible to construct a global state transition graph corresponding to the environment in which all processes execute concurrently. The concurrency model implemented in Verus is synchronous, that is, one transition in the global model corresponds to exactly one transition in each process.

Given two processes defined by their state transition graphs $G_1 = (S_1, I_1, T_1)$ and $G_2 = (S_2, I_2, T_2)$ we can construct a global state transition graph $G = (S, I, T)$ by:

- $S = \{(s_1, s_2) \mid s_1 \in S_1, s_2 \in S_2$ and $s_1\langle v\rangle = s_2\langle v\rangle$, for all shared variables $v\}$
  where $s\langle v\rangle$ denotes the truth value of variable v in state s.

  Each state in the global model contains one component in each process. However, one constraint must be satisfied. If a variable is referenced in more than one process, its value in each component of the global state space must the same. This model guarantees consistency of the values of shared variables.

- $I = \{(i_1, i_2) \mid i_1 \in I_1, i_2 \in I_2$ and $(i_1, i_2) \in S\}$

  An initial state in G is a state in the global model that is an initial state in all processes.

- $T((s_1, s_2), (t_1, t_2))$ iff $T_1(s_1, t_1)$ and $T_2(s_2, t_2)$

  A transition in the global model exists iff it corresponds to existing transitions in each component. Symbolically T is constructed by conjuncting T_1 and T_2. The meaning of the formula representing the global transition relation is that a transition exists if transitions exist in *all* components.

This construction allows the specification and verification of systems in which several processes execute concurrently. The synchronous model can be relaxed using stuttering, as discussed in section 6.1. This model is expressive enough to allow the description of most types of practical systems.

Prioritized Composition

Real-time systems frequently use priorities to ensure that critical processes are not delayed by less important ones. Priorities are needed whenever there is contention for resources in the system, such as the processor. A scheduler is used to decide which process accesses the resource at any time, and a real-time scheduler uses priority information to decide the access order for the resource.

In Verus the scheduler can be implemented in the core language to model this behavior. Both static and dynamic priority schedulers can be implemented. A static priority scheduler can be seen below. The scheduler receives requests from each process (via the req*i* variables), and asserts the variable granted, which signals its decision about which process executes next.

```
scheduler(req1, req2, req3, granted)
boolean req1, req2, req3;
int granted;
{
  while (true) {
    if (req1)  granted = 1;  else
    if (req2)  granted = 2;  else
    if (req3)  granted = 3;  else
    granted = 0;
    wait(1);
  };
```

The scheduler chooses which process executes next by following the nested if structure. The order in which the requests are tested define the priority order used.

Whenever requesting execution, process p*i* sets variable req*i* to true. When it finishes executing it resets the variable. During execution it must wait until the variable granted has its index before proceeding:

```
    /* Beginning of execution */
    req1 = true;
    while (granted != 1) wait(1);
    wait(1);  /* execute for one time unit */
    ...
    while (granted != 1) wait(1);
    wait(1);
    /* End of execution */
    req1 = false;
    wait(1);
```

Dynamic scheduling can also be implemented. In this case the `req`*i* variables are integers and contain the priority level at which the process is requesting execution. One way of defining the values for the `req`*i* variables is to use the `priority` statement. The statement `priority(`*n*`)  {`S_1` };` is translated to:

```
reqi = n;
S1;
reqi = 0;
```

This can be easily generalized. For example the earliest deadline scheduling algorithm can be implemented by assigning to the request variable the difference between the current time and the deadline. The scheduler now must choose the process with the highest requested priority:

```
scheduler(req1, req2, req3, granted)
int req1, req2, req3, granted;
{
  while (true) {
    if (req1 >= req2) {
      if (req1 >= req3)  granted = 1; else
                         granted = 3;
    } else {
      if (req2 >= req3)  granted = 2; else
                         granted = 3;
    };
    wait(1);
  };
```

Many different schedulers can be written to suit specific applications. For example, the scheduler above always favors the process with the lower index in case of equal priority levels. In some cases this might not be desirable, and the scheduler has to be refined to reflect this feature of the system.

Notice that issues such as fairness, absence of deadlocks and starvation depend on the specific scheduler being used. For example, it can be easily seen that fairness is not guaranteed by the schedulers described above. They always favor higher priority processes, and may starve lower priority ones. However, this is allowed in real-time resource management, and the schedulers are correct. Different schedulers may have different requirements, and these issues have to be considered again if new schedulers are introduced.

Prioritized composition allows the specification of many common types of real-time systems in a straightforward way. In fact, most of the examples described in the next chapter have been implemented using this paradigm.

Chapter 5 Verification Algorithms

Previous chapters have described how to specify a real-time system, and how to generate, from the specification, a model that is amenable to a formal analysis. This chapter will describe in detail the algorithms used to verify real-time systems in Verus. The simplest method consists of introducing time bounds on CTL operators. Later more powerful algorithms will be explored that introduce quantitative and path selective analysis methods into the Verus tool.

5.1 RTCTL Model Checking

Symbolic model checking algorithms are able to verify a large and important class of properties of computer systems in general. Properties such as liveness and safety can be easily expressed and verified. However, there is an important class of properties that cannot be adequately handled using this method. This class consists of the properties which place bounds on response time. In CTL it is possible to state that some event will happen in the future, but the property that some event will happen at most x time units in the future cannot be expressed directly.

A simple and effective way to allow the expression and verification of time bounded properties is to introduce bounds in the CTL temporal operators. The extended logic is called RTCTL [33]. The expressive power of RTCTL is the same as CTL, since the bounded operators can be translated into nested applications of the **EX** (or **AX**) operators. However, this translation is often impractical, and RTCTL provides a much more compact and convenient way of expressing such properties.

The basic RTCTL temporal operator is the *bounded until* operator which has the form: $\mathbf{U}_{[a,b]}$, where $[a,b]$ defines the time interval in which the property must be true. We say that $f\,\mathbf{U}_{[a,b]}\,g$ is true of some path if g holds in some future state s on the path, f is true in all states between the beginning of the path and s, and the distance from this state to s is within the interval $[a,b]$. The bounded **EG** operator can be defined similarly. Other temporal operators are defined in terms of these.

More formally, we extend the CTL semantics to include the *bounded until* operator by adding the following clauses to the formal semantics given in section 2.1.1:

7. $s \models \mathbf{E}[f\,\mathbf{U}_{[a,b]}\,g]$ iff there exists a path $\pi = s_0\,s_1\,s_2 \ldots$ starting at $s = s_0$ and some i such that $a \le i \le b$ and $s_i \models g$ and for all $j < i$, $s_j \models f$.
8. $s \models \mathbf{EG}_{[a,b]}f$ iff there exists a path $\pi = s_0\,s_1\,s_2 \ldots$ starting at $s = s_0$ and for all i such that $a \le i \le b$, $s_i \models f$.

As an example of the use of the bounded until consider the property “It is always true that p may be followed by q within 3 time units”. This property can be expressed in RTCTL as $\mathbf{AG}(p \rightarrow \mathbf{EF}_{[0,3]}\,q)$. The bounded **F** operator is derived from the bounded until just as in the unbounded case, i.e. $\mathbf{EF}_{[a,b]}\,f \equiv \mathbf{E}[\text{true}\ \mathbf{U}_{[a,b]}\,f]$.

In order to implement this operator, we will use a fixpoint computation that is similar to the one implemented in CTL model checkers. It is easy to see that the formula $\mathbf{E}[f\,\mathbf{U}_{[a,b]}\,g]$ can be expressed in the form:

if $a > 0$ and $b > 0$: $\mathbf{E}[f\ \mathbf{U}_{[a,b]}\ g] = f \wedge \mathbf{EX}\ \mathbf{E}[f\ \mathbf{U}_{[a\text{-}1,b\text{-}1]}\ g]$

else, if $b > 0$: $\mathbf{E}[f\ \mathbf{U}_{[0,b]}\ g] = g \vee (f \wedge \mathbf{EX}\ \mathbf{E}[f\ \mathbf{U}_{[0,b\text{-}1]}\ g])$

else: $\mathbf{E}[f\ \mathbf{U}_{[0,0]}\ g] = g$

Other operators are defined similarly.

Consider the first of these cases. We compute the sets of states where *f* is true for *a* steps. During this computation, a fixpoint may be reached before *a* iterations have passed. When this happens, we can skip to the second case. By using this optimization, the number of required iterations may be reduced when the time interval is large, but a fixpoint is reached quickly. The same optimization can also applied in the second case. If a fixpoint is reached before *b* - *a* iterations, with *b* and *a* being respectively the upper and lower bounds of the operator, we can immediately proceed to the third case.

5.2 Quantitative Analysis: Minimum/Maximum Delay

Traditional formal verification algorithms assume that timing constraints are given explicitly in some notation like temporal logic. Typically, the designer provides a constraint on response time for some operation, and the verifier automatically determines if it is satisfied or not. These techniques do not provide any information about how much a system deviates from its expected performance, although this information can be extremely useful in fine-tuning the behavior of the system.

This section describes algorithms to compute quantitative timing information, such as exact minimum and maximum delays on the time between a request and the corresponding response. We also present algorithms that compute the minimum and maximum number of times a certain condition is satisfied on all paths between two given events. For example, we can use these algorithms to bound the time between asserting a bus request and the corresponding bus grant. In addition, we may need to compute the number of times a third event occurs within such an interval, such as the number of times other transactions are issued between the bus request and the corresponding grant.

5.2.1 Minimum Delay Algorithm

The algorithm takes two sets of states as input, *start* and *final*. It returns the length of (i.e. number of edges in) a shortest path from a state in *start* to a state in *final*. If no such path exists, the algorithm returns infinity. In the algorithm, the function $T(S)$ gives the set of states that are successors of some state in S. In other words, $T(S) = \{s' \mid N(s, s')$ holds for some $s \in S\}$. In addition, the variables R and R' represent sets of states in the algorithm.

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

Figure 16. Minimum Delay Algorithm

The first algorithm is relatively straightforward. Intuitively, the loop in the algorithm computes the set of states that are reachable from *start*. If at any point, we encounter a state satisfying *final*, we return the number of steps taken to reach the state. Figure 17 shows how the algorithm works by computing the successors of the current frontier (the shaded area) at each iteration.

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

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

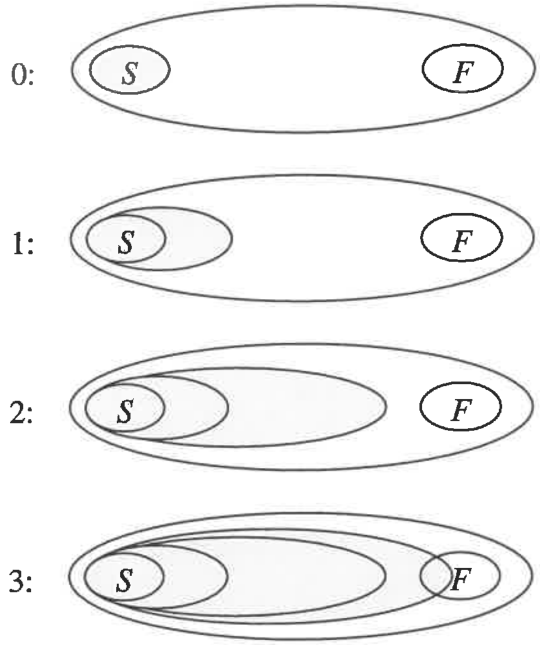


Figure 17. Minimum algorithm: it searches forward from S.

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

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

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

- The loop invariant on R, $R = S_i$, and the second half of the loop test, $R \cap \mathit{final} = \varnothing$ imply that $i < L$, otherwise some state in S_i would belong to final. This inequality implies $i + 1 \leq \mathrm{L}$ so we can safely increment i without violating the invariant on i.

- The second statement sets R to the value of R', so now $R = S_{i+1}$. This means that R now satisfies its invariant with the new value of i.
- The last statement sets R' to the union of R' and the image of R'. So by construction, we know that $R' = S_{i+2}$ Therefore, R' satisfies its invariant with the new value for i.

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

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

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

5.2.2 Maximum Delay Algorithm

This algorithm also takes *start* and *final* as input. It returns the length of a longest path from a state in *start* to a state in *final*. If there exists an infinite path beginning in a state in *start* that never reaches a state in *final*, the algorithm returns infinity. The function $T^{-1}(S')$

gives the set of states that are predecessors of some state in S' (i.e. $T^{-1}(S') = \{s \mid N(s, s')$ holds for some $s' \in S'\}$). R and R' will once more be sets of states. Finally, we denote by *not_final* the set of all states that are not in *final*.

```
proc $max$ ($start, final$)
$i = 0$;
$R = true$;
$R' = not\_final$;
while $((R' \neq R) \wedge (R' \cap start \neq \varnothing))$ do
    $i = i + 1$;
    $R = R'$;
    $R' = T^{-1}(R') \cap not\_final$;
if $(R = R')$
    then return $\infty$;
    else return $i$;
```

Figure 18. Maximum Delay Algorithm

The upper bound algorithm is more subtle than the previous algorithm. In particular, we must return infinity if there exists a path beginning in *start* that remains within *not_final*. A backward search from the states in *not_final* is more convenient for this purpose than a forward search. Figure 19 shows the maximum delay algorithm. At each iteration it finds the set of states which are the beginning of intervals with i states, none satisfying *final*. Initially, i is 0, and the frontier is *not_final*. At the i^{th} iteration the current frontier is the set of states that are the beginning of paths with i states completely in *not_final*. We then compute the set of predecessors (in *not_final*) of the current frontier. Those states are the beginning of paths with i+1 states completely in *not_final.*

We use the following two definitions in proving the algorithm correct:

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

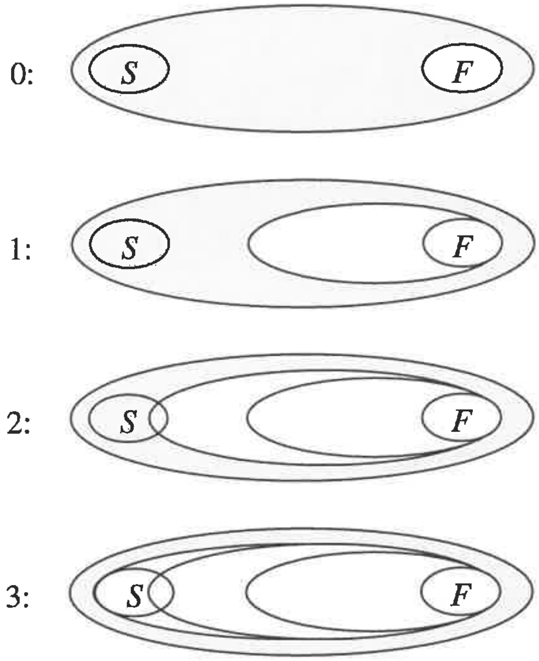


Figure 19. Maximum algorithm: it searches backwards from $\neg F$

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

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

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

- The invariant on R', $R' = S_{i+1}$, and the second half of the loop test, $R' \cap start \neq \varnothing$ imply that $i + 1 \leq$ M. Therefore, we can increment i without violating the invariant on i.
- The second statement sets R to the value of R' so we know that now $R = S_{i+1}$. This means that R now satisfies its invariant with the new value for i.
- The third statement sets R' to the inverse image of R' intersected with *not_final*. The invariant gave us that $R' = S_{i+1}$ before the assignment. By construction, after the assignment we have that $R' \subseteq S_{i+2}$. For the inclusion in the other direction, we observe that any path of $i + 2$ states contained in *not_final* can be thought of as beginning in a state labeled with *not_final* that has an edge to a state that is the beginning of a path of $i + 1$ states labeled with *not_final*. In other words, the states in S_{i+2} are states in *not_final* with an edge to a state in S_{i+1}. But these are precisely the states just computed for the new value of R' so we get that $S_{i+2} \subseteq R'$. This means that R′ also satisfies its invariant with the new value for i.

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

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

We complete the proof by analyzing what happens at the final conditional. We first consider the case where we exit the loop because R = R′. In this case, we have reached a fixpoint. By the invariant, $R = S_i$ and $R' = S_{i+1}$; therefore we have $S_i = S_{i+1}$. We argued previously that the states in S_{i+1} have edges to states in S_i. Since $S_{i+1} = S_i$, we know that every state in S_{i+1} has an edge to another state in S_{i+1}. So every state in S_{i+1} is the beginning of an infinite path of states remaining in $S_{i+1} \subseteq not\_final$. The previous observation tells us that $R' \cap start \neq \varnothing$, therefore some state $s \in S_{i+1}$ belongs to start. This state then is the beginning of an infinite path starting at a state in *start*, which never reaches a state in *final*, so we return infinity.

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

5.3 Quantitative Analysis: Condition Counting

In many situations we are interested not only in the length of a path leading from a set of starting states to a set of final states, but also in measures that depend on the number of states on the path that satisfy a given condition. For example, we may wish to determine the minimum or maximum number of times a condition *cond* holds on any path from *start* to *final*. The algorithm in this section compute this information.

To simplify the algorithms, we assume that any path beginning in *start* must reach a state in *final* in a finite number of steps. This requirement is necessary to ensure that the minimum (maximum) is well-defined. It can be checked using the upper bound algorithm described in the previous section. Finally, we assume that all computations involve only reachable states. This can be achieved by intersecting *start* with the set of reachable states computed *a priori*.

To keep track at each step of the number of states in *cond* that have been traversed, we define a new state-transition system, in which the states are pairs consisting of a state in the original system and a positive integer. Thus, if the original state-transition graph has state set S, then the augmented state set will be $S_a = S \times \mathbb{N}$.

If $N \subseteq S \times S$ is the transition relation for the original state-transition graph, we define the augmented transition relation $N_a \subseteq S_a \times S_a$ as

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

In other words, there will be a transition from $\langle s,k \rangle$ to $\langle s',k' \rangle$ in the augmented transition relation N_a iff there is a transition from s to s' in the original transition relation N and either $s' \in cond$ and $k' = k + 1$ or $s' \notin cond$ and $k' = k$. We also define T_a to be the function that for a given set $U \subseteq S_a$ returns the set of successors of all states in U. More formally, $T_a(U) = \{u' \mid N_a(u, u') \text{ holds for some } u \in U\}$. In the actual BDD-based implementation, an initial bound $k_{\max}$ can be selected to achieve a finite representation for k, and new BDD variables can be added dynamically if this bound is exceeded. The system is still finite-state because all paths we consider are finite and k is bounded by their maximum length.

5.3.1 Minimum Condition Counting

The algorithm for computing the minimum count is given in figure 20. In the algorithm text, *final* and *not_final* denote the sets of states in *final* and S - *final*, paired with all possible values of k. More formally:

$$\mathit{final} = \{\langle s,k\rangle \mid s \in \mathit{final},\ k \in \mathbb{N}\} \quad \text{and} \quad \mathit{not\_final} = \{\langle s,k\rangle \mid s \notin \mathit{final},\ k \in \mathbb{N}\}$$

The algorithm uses R to represent the state set in S_a reached at the current iteration, while *reached_final* and R' are its intersections with *final* and *not_final* respectively. Variable *current_min* denotes the minimum count for all previous iterations. The minimum computation over the set of values of k can be done by quantifying out the state variables and following the leftmost nonzero branch in the resulting BDD, provided it uses an appropriate variable ordering. An efficient algorithm that does not depend on the variable ordering is given in [58].

```
proc mincount (start, cond, final)
current_min = $\infty$;
$R = \{\langle s,1\rangle \mid s \in start \cap cond\} \cup \{\langle s,0\rangle \mid s \in start \cap \neg cond\}$;
while true do
    $reached\_final = R \cap final$;
    if $reached\_final \neq \emptyset$ then
        $m = \mathbf{min}\{k \mid \langle s,k\rangle \in reached\_final\}$;
        if $m < current\_min$ then $current\_min = m$;
    $R' = R \cap not\_final$;
    if $R' = \emptyset$ then return $current\_min$;
    $R = T(R')$;
```

Figure 20. Minimum Condition Count Algorithm

At iteration i, the algorithm considers the endpoints of paths with i states. The reached states that belong to *final* are terminal states on paths that we need to consider. The minimum count for these paths is computed, using the counter component of the path endpoints, and the current value of the minimum is updated if necessary. For the reached states that do not belong to *final*, we continue the loop after computing their successors. If all reached states are in *final*, there are no further paths to consider and the algorithm returns the computed minimum. Figure 21 shows how both the minimum and the maximum algorithm work. In the figure, black states satisfy *cond.* The current frontier (the shaded area) is expanded forward, while the counter component maintains the number of occurrences of *cond* on paths from *S*. In the figure, darker areas have higher counts, and the last iteration shows a minimum count and a maximum count paths.

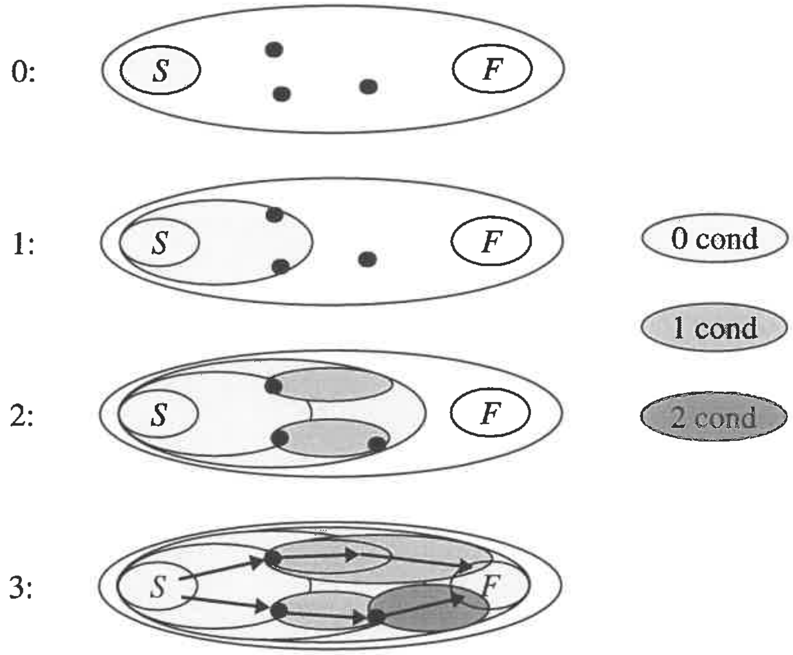


Figure 21. Condition counting: the condition counter is updated during traversal.

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

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

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

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

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

Next, we argue that the algorithm terminates. The precondition ensures that all paths from *start* reach *final* in a finite number of steps. Thus, we will eventually have $R' = R \cap \mathit{not\_final} = \emptyset$, and the algorithm correctly returns the value *current_min*.

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

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

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

5.3.2 Maximum Condition Counting

The algorithm for the maximum count, given below, has the same structure as the minimum count and can be obtained by replacing **min** with **max** and reversing the inequalities. Variants of both algorithms can be used to compute other measures that are a function of the number of states on a path that satisfy a given condition. For example, we can determine the minimum and the maximum number of states belonging to a given set *cond* over all paths of a certain length *l* in the state space.

```
proc maxcount (start, cond, final)
current_max = -∞;
$R = \{\langle s,1\rangle \mid s \in start \cap cond\} \cup \{\langle s,0\rangle \mid s \in start \cap \neg cond\}$;
loop
    $reached\_final = R \cap final$;
    if $reached\_final \neq \emptyset$ then
        $m = \mathbf{max}\{k \mid \langle s,k\rangle \in reached\_final\}$;
        if $m > current\_max$ then $current\_max = m$;
    $R' = R \cap not\_final$;
    if $R' = \emptyset$ then return current_max;
    $R = T(R')$;
endloop;
```

Figure 22. Maximum Condition Count Algorithm

5.4 Quantitative Analysis: Optimized Condition Counting

The condition counting algorithms presented in the previous section augment the state space with a counter *k*. This counter maintains information about the number of times the condition *cond* has been encountered on paths from *start*. The algorithm actually computes all possible values of *k* for all paths beginning in *start*. Algorithms requiring the exact number of times *cond* occurs can be constructed from the basic condition counting algorithms. However, for the computation of the minimum and maximum the exact number of occurrences is not needed. The algorithms presented in this section can be used to count minimum and maximum occurrence times without augmenting the state space.

5.4.1 Optimized Minimum Condition Counting

The minimum condition count algorithm computes the minimum number of states satisfying a given condition *cond* over all paths that start in a state in *start* and end in a state in *final*. Any paths starting in *start*, but which do not reach *final* in a finite number of steps are excluded from this computation. In particular, if no path from *start* ever reaches *final*, the algorithm will return the special value NOPATH.

The algorithm searches forward beginning in *start*. It looks for paths with an increasing number of occurrences of *cond*. Each iteration consists of two phases: The first is a forward traversal through states that do not satisfy *cond*. This traversal is performed until all states (not satisfying *cond*) reachable from the current frontier are found (steps 1 and 3 in figure 24). If *final* has not been reached yet, the frontier is expanded by one step to states that satisfy *cond* and the condition counter is incremented (steps 2 and 4 below). The algorithm iterates until *final* is found, or all reachable states are visited.

The algorithm must differentiate between states that do not satisfy *cond* and those that do, and similarly, between transitions leading to these states. We use subscripts 0 and 1 respectively for the two types of states and transitions. For example, $start_0$ is the set of initial states that do not satisfy *cond*, and $start_1$ is the set of initial states that satisfy *cond*.

$$start_0 = start \cap \neg cond \qquad\qquad start_1 = start \cap cond$$

Furthermore, if $N(s, s')$ is the transition relation, we denote by $T_0(S)$ and $T_1(S)$ the set of transitions from a state in S that lead to states not satisfying *cond* and to states satisfying *cond*, respectively:

$$T_0(S) = \{s' \mid \exists\, s \in S \,.\, N(s,s') \wedge s' \notin cond\}$$
$$T_1(S) = \{s' \mid \exists\, s \in S \,.\, N(s,s') \wedge s' \in cond\}$$

```
**proc** $mincount$(start, $cond$, $final$)
$i = 0$;
$R = \emptyset$;
$R' = start_0$;
**do**
    **do**
        **if** ($R' \cap final \neq \emptyset$) **return** $i$;
        $R = R'$;
        $R' = T_0(R') \cup R'$;
    **while** ($R' \neq R$);
    $R' = T_1(R') \cup R'$;
    **if** ($i = 0$) $R' = R' \cup start_1$;
    $i = i + 1$;
**while** ($R' \neq R$)
**return** NOPATH;
```

Figure 23. Minimum Condition Count Algorithm

We will prove the correctness of the algorithm by showing that certain loop invariants are satisfied and that the algorithm terminates. We will use the following notations:

- *endpoints*(i) is the set of all states that can be reached as endpoints of finite paths starting in *start* and having i or less states in which *cond* holds.
- *mincount* is the minimum condition count as described above.

Two invariants are defined. The first holds at the start of iteration $i + 1$ of the outer loop:

$$I_1(i): ((i = 0 \wedge R' = start_0) \vee (i > 0 \wedge R' = endpoints(i) \cap cond)) \wedge mincount \geq i$$

The second holds, in the same iteration, after the inner loop:

$$I_2(i): R' = endpoints(i) \wedge R' \cap final = \emptyset \wedge mincount \geq i$$

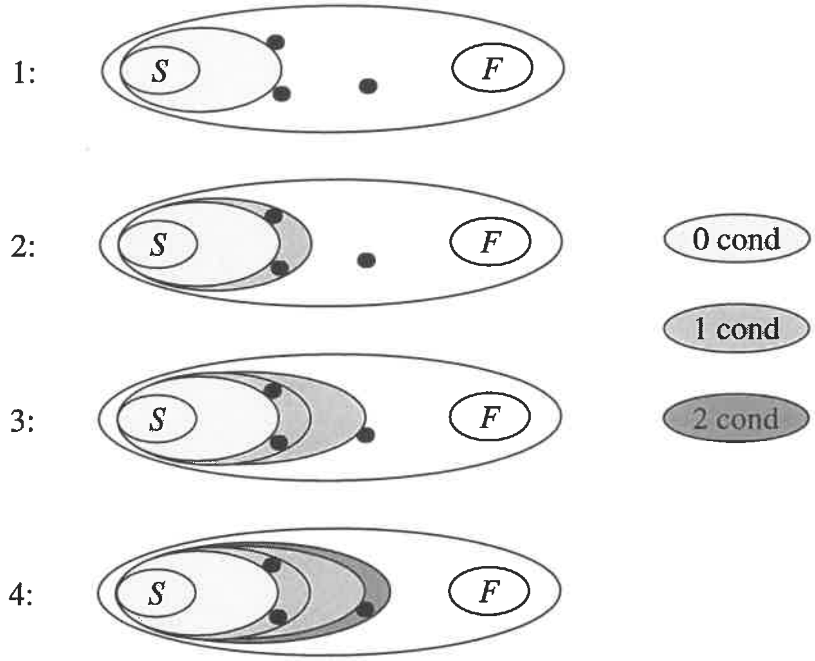


Figure 24. Optimized minimum count: each iteration has two steps: 1 and 2; 3 and 4.

Lemma 1 The algorithm terminates.

Proof If a state satisfying *final* is reached, the algorithm clearly terminates, returning the current value of i. If not, the exit condition of both loops is $R' = R$. By construction $R' \supseteq R$, and since there is only a finite number of states, there can be no infinite sequence of distinct values for R'.

Lemma 2 I_1 and I_2 are invariants of the algorithm.

Proof The correctness of the invariants will be proven by induction on i. Invariant $I_1(0)$ clearly holds at the beginning of the first outer loop, since $i = 0$, $R' = start_0$, and $mincount \geq 0$. We will now prove that $I_1(i) \rightarrow I_2(i)$ and $I_2(i) \rightarrow I_1(i+1)$.

Assume I_1 holds at the beginning of iteration i. If $i = 0$, we have $R' = start_0$ before the inner loop. If the control flow exits the inner loop, a fixpoint has been reached. R' will contain all states reachable from *start* on paths where *cond* does not hold (because of the restriction on T_0). But this is exactly *endpoints*(0), by definition, so $I_2(0)$ holds. If $i > 0$, then by I_1 we have $R' = endpoints(i) \cap cond$ before the inner loop. A state s is in $endpoints(i) \cap \neg cond$ iff there is a state $t \in$ endpoints(i) $\cap$ *cond* from which s is reachable by a path that does not pass again through *cond*. The inner loop adds precisely the set of all such states to R'. Therefore, after the inner loop $R' = endpoints(i)$. Moreover, $R' \cap final = \varnothing$, otherwise the algorithm would have terminated in the inner loop. We conclude that $I_1(i) \rightarrow I_2(i)$.

Now assume $I_2(i)$ is true at the end of the inner loop, and let us prove that $I_1(i+1)$ holds at the beginning of the next outer loop iteration. A state $s \in endpoints(i+1) \cap cond$ satisfies one of two cases: It is either on a nondegenerate path from start, and thus reachable by a transition from a state in *endpoints*(i), or it is on a degenerate path from *start* ($i = 0$ and $s \in start_1$). In the first case, it is added to R' in the first statement after the inner loop, and in the second case, in the second statement after the inner loop. Therefore, after line 12, $R' = endpoints(i+1) \cap cond$. Furthermore, since the inner loop was exited with $R' = endpoints(i) \cap final$ by $I_2(i)$, there is no path with count $\leq i$ that reaches final, so $mincount > i$ or, equivalently $mincount \geq i + 1$. Taking into account that i is incremented in line 13, both conjuncts of $I_1(i+1)$ are satisfied, which shows that $I_2(i) \rightarrow I_1(i+1)$. The induction proof is completed.

Lemma 3 The algorithm returns the minimum number of occurrences of *cond* in any path from *start* to *final.*

Proof The correctness of the algorithm can be seen by analyzing its return value. There are two possible return conditions, the first being when $R' \cap final \neq \varnothing$ and the algorithm returns i. At this point $R' \subset endpoints(i)$ by I_1 and the restriction on T_0. Consider a state $s \in R' \cap final$. Then $s \notin endpoints(i\text{-}1)$ since $I_2(i\text{-}1)$ ensured that $endpoints(i\text{-}1) \cap final = \varnothing$.

Therefore s must be reached by a path containing exactly i states satisfying *cond* and thus $mincount \leq i$. Since by I_1 $mincount \geq i$, we conclude that $mincount = i$.

If the algorithm returns NOPATH, then R' doesn't intersect *final* in any iteration. The algorithm exits the outer loop when $R = R'$, which means that a fixpoint has been found. Every state reachable from start can be found by alternating (possibly empty) sequences of transitions in T_0 with transitions in T_1. Therefore that fixpoint is precisely the set of states reachable from *start*. We can conclude that all paths originating in *start* will be completely contained in $\neg final$, since $R' \cap final = \varnothing$. Hence, the algorithm returns the correct value NOPATH.

5.4.2 Optimized Maximum Condition Counting

The maximum condition count algorithm computes the maximum number of states satisfying a given condition *cond* over all paths that begin in a state in *start* and end in a state in *final* without previously traversing a state in *final*. If there is a path beginning in *start* that goes through *cond* infinitely often without reaching *final*, the algorithm returns infinity. The basic idea behind the algorithm is to find paths with increasing condition count whose states are all within $\neg final$. The condition count of the longest path satisfying this condition and starting in *start* is the desired maximum.

The algorithm assumes that all states are reachable from *start*. This can be enforced by performing a reachability computation from *start* and restricting the state space to reachable states. Moreover, we require that every state has at least one outgoing transition.

Similarly to the mincount algorithm, we will denote transitions into states that satisfy *cond* and that do not satisfy *cond* separately. This algorithm, however, performs a backward search, and we must define the reverse image of the transition relation. In this case $B_0(S')$ is the set of states satisfying neither *cond* nor *final* that lead to a state in S' in one step. Similarly $B_1(S')$ is the set of states satisfying *cond* but $\neg final$ that lead to a state in S' in one step. Note that *final* only appears implicitly in the algorithm, in the definitions of B_0 and B_1.

$$B_0(S') = \{s \mid \exists\, s' \in S' \,.\, N(s, s') \wedge s \notin \mathit{final} \wedge s \notin \mathit{cond}\}$$
$$B_1(S') = \{s \mid \exists\, s' \in S' \,.\, N(s, s') \wedge s \notin \mathit{final} \wedge s \in \mathit{cond}\}$$

We use the following notations:

- *startpoints*(i) is the set of all states that are the start of a finite path in which has no states in *final* (except possibly the last one), and which has i states that belong to *cond*.
- *maxcount* is the maximum condition count, for a path starting in *start* and ending in *cond*, in which no states belong to *final*, except possibly for the last one belong to *final*.

```
proc maxcount(start, cond, final)
i = 1;
R' = cond;
do
    R_1 = R';
    do
        R = R';
        R' = R' ∪ B_0(R');
    while (R' ≠ R);
    if (R' ∩ start = ∅) return i - 1;
    R' = B_1(R');
    i = i + 1;
while (R' ≠ R_1);
return ∞;
```

Figure 25. Maximum Condition Count Algorithm

We prove the correctness of the algorithm using two invariants. The first one holds at the beginning of the outer loop:

$$I_1(i)\text{: } R' = \mathit{startpoints}(i) \cap \mathit{cond} \wedge i - 1 \leq \text{maxcount}$$

The second invariant holds at the end of the inner loop:

$$I_2(i)\text{: } R' = \mathit{startpoints}(i) \wedge i - 1 \leq \mathit{maxcount}$$

Lemma 4 I_1 and I_2 are invariants of the algorithm.

Proof We prove by induction on i that the invariants hold at the corresponding points in the algorithm and argue separately about termination. Invariant $I_1(0)$ trivially holds at the beginning of the first iteration, since paths with a condition count of 1 that have both endpoints in *cond* are exactly the degenerate paths consisting of a state in *cond*. Furthermore, clearly $\mathit{maxcount} \geq 0$. Next, we prove that $I_1(i) \rightarrow I_2(i)$ and that $I_2(i) \rightarrow I_1(i+1)$.

Assume that $I_1(i)$ holds. The inner loop adds to R' all states that lead to states in R_1, without being in *final* or *cond*. Since all states in *startpoints*(i) can be found by a backward traversal from $\mathit{startpoints}(i) \wedge \mathit{cond}$ and i does not change, this establishes $I_2(i)$.

Now assume $I_2(i)$ holds after the inner loop, and that $R' \cap \mathit{start} \neq \varnothing$ (otherwise the algorithm terminates and we have no further invariants to prove). Then there is at least one path from *start* to *cond* with i occurrences of *cond*, and therefore $\mathit{maxcond} \geq i$. A state p is in $\mathit{startpoints}(i+1) \cap \mathit{cond}$ exactly if it belongs to *cond* and it has some successor in *startpoints*(i). Therefore, since $R' = \mathit{startpoints}(i)$ by $I_2(i)$, we will have $B_1(R') = \mathit{startpoints}(i+1) \cap \mathit{cond}$, thus $i - 1 \leq \mathit{maxcond}$ after i is incremented, and $I_1(i+1)$ holds, which completes our induction proof.

Lemma 5 The algorithm terminates.

Proof The inner loop of the algorithm performs a backward reachability computation. It is executed only a finite number of times, because the value of R' is monotonically increasing, and the state space is finite, so a fixpoint has to be reached. Next, we argue that the outer loop finishes as well. This clearly happens if at some point, $R' \cap start = \emptyset$ after the inner loop. Otherwise, let us show that the sequence of values R' at the end of each outer loop iteration is monotonically decreasing. By I_1, $R' = startpoints(i) \cap cond$. But any state in *startpoints*(i) is certainly in *startpoints*(i-1), since we can restrict the path with i occurrences of *cond* to some prefix containing only i - 1 states in *cond*. Since the state space is finite, the monotonically decreasing sequence of sets R' will eventually reach a fixpoint, the loop terminates and the execution of the algorithm as well.

Lemma 6 The algorithm returns the maximum number of occurrences of *cond* in any path from *start* to *final*.

Proof If $R' \cap start = \emptyset$ at the end of the inner loop, this means that there are no paths with count i leading from *start* to *cond*, completely in $\neg final$, and consequently, no paths with count greater than i (since they would have a prefix with count i). Therefore, $maxcount < i$. Since by I_2, $maxcount \geq i - 1$, it follows that $maxcount = i - 1$ is the correct return value. If the outer loop is exited due to the fixpoint, $startpoints(i) = startpoints(i+j)$ for all $j \geq 0$. Moreover, $startpoints(i) \cap start \neq \emptyset$, therefore, there exists an infinite path beginning in *start*, completely contained in $\neg final$ and in which *cond* holds infinitely often.

5.5 Selective Quantitative Analysis and Interval Model Checking

Typically, 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. This section presents algorithms that allow the designer to perform quantitative analysis very accurately by selecting execution sequences of interest and analyzing them separately.

Formulas of the linear-time temporal logic LTL are used 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 [22,56,74] 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 [22] for *interval model checking*, that is, checking a formula with respect to finite intervals.

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.

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.

Linear-time temporal logics interpreted over both infinite paths and finite intervals have been introduced in [57,61]. However, they only check the satisfiability of a formula, and do not handle either quantitative analysis or interval model checking. Interval logics are also used in [67], but in a theorem proving context. However, what differentiates our method from related ones 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 [20,32,62]. 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.

5.5.1 A tableau for LTL

Our specification language is a *linear-time temporal logic* called LTL [65]. 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* [56,74,22] 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 $\neg f$, $\mathrm{f} \vee \mathrm{g}$, $\mathbf{X} f$ and $f\, \mathbf{U}\, g$ are also LTL formulas.

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

An infinite sequence s_0, s_1, ... 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 $[s_0, ..., s_n]$ 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 interval or an infinite path. Thus, s_n may or may not have successors in M. The size of interval $\sigma = [s_0, ..., s_n]$, 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, the meaning of $M, \pi \models_{\text{path}} f$ is that f holds along path π in the graph M. $M, \sigma \models_{\text{int}} f$ means that f holds along interval σ in M. Given a designated set of initial states S_0, we say that $M, S_0 \models_{\text{path}} f$ iff for every path π from every state in S_0, $M, \pi \models_{\text{path}} f$. Given two designated sets of states *start* and *final*, we say that M, $[\mathit{start}, \mathit{final}] \models_{\text{int}} f$ iff for every interval σ from some state in *start* to some state in *final*, M, s $\models_{\text{int}} f$. Note that this definition does not require that intervals be disjoint. Unless otherwise stated, overlapping intervals are allowed.

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

1. $\pi \models_{\text{path}} p$ iff $p \in L(s_0)$, for $p \in AP$.

2. $\pi \models_{\text{path}} \neg f$ iff $\pi \not\models_{\text{path}} f$.

3. $\pi \models_{\text{path}} f_1 \vee f_2$ iff $\pi \models_{\text{path}} f_1$ or $\pi \models_{\text{path}} f_2$.

4. $\pi \models_{\text{path}} \mathbf{X} f_1$ iff $\pi^1 \models_{\text{path}} f_1$.

5. $\pi \models_{\text{path}} f_1 \mathbf{U} f_2$ iff there exists a $k \geq 0$ such that $\pi^k \models_{\text{path}} f_2$ and for all $0 \leq j < k$, $\pi^j \models_{\text{path}} f_1$.

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

6. $\sigma \models_{\text{int}} \mathbf{X} f_1$ iff $| \sigma | > 0$ and $\sigma^1 \models_{\text{int}} f_1$.

7. $\sigma \models_{\text{int}} f_1 \mathbf{U} f_2$ iff there exists a $0 \leq k \leq n$ such that $\sigma^k \models_{\text{int}} f_2$ and for all $0 \leq j < k$, $\sigma^j \models_{\text{int}} f_1$.

The following abbreviations are used in writing LTL formulas:

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

In the following, whenever we refer to a path that satisfies a formula, the satisfaction is with respect to $\models_{\text{path}}$. Whenever an interval is considered the satisfaction is with respect to $\models_{\text{int}}$. Finally, whenever states are considered, satisfaction is with respect to $\models$ for CTL, as defined in section 2.1.1.

Note that, in the definition of $[s_0, \ldots s_n] \models f$ we do not consider successors of s_n (whether they 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 may have quite a different meaning when interpreted over paths or over intervals. For instance, a path will satisfy the formula **G 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. $[s_0] \models_{\text{int}} \neg\, \mathbf{X}\, a$ but $[\mathrm{s0}] \not\models_{\text{int}} \mathbf{X} \neg a$.

Let *f* be an LTL formula. We construct a Kripke structure *T(f)*, called the *tableau* for *f*, containing all paths and intervals satisfying *f*. The tableau described below is based on the construction given in [22]. There, the tableau is used to check the truth of a LTL formula for all paths of a given Kripke structure. Here it will be used for three purposes:

- Selecting the set of paths of a structure that satisfy f and computing minimum and maximum delays over those paths;
- Selecting the set of intervals of a structure that satisfy f and computing minimum and maximum delays over those intervals;
- Checking that a specified set of intervals of a structure satisfy f.

We first introduce the notion of *fairness constraint*, 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.

The tableau $T(f)$ is constructed as follows. 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 by:

- $el(p) = \{p\}$ if $p \in AP_f$.
- $el(\neg f) = el(f)$.

- $el(f \vee g) = el(f) \cup el(g)$.
- $el(\mathbf{X} f) = \{\mathbf{X} f\} \cup el(f)$.
- $el(f\, \mathbf{U}\, g) = \{\mathbf{X}(f\, \mathbf{U}\, g)\} \cup el(f) \cup el(g)$.

Thus, the set of states S_T of the tableau is *Powerset*($el(f)$). 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 $g \in el(f)$.
- $sat(\neg g) = \{s \mid s \notin sat(g)\}$.
- $sat(g \vee h) = sat(g) \cup sat(h)$.
- $sat(g\, \mathbf{U}\, h) = sat(h) \cup (sat(g) \cap sat(\mathbf{X}(g\, \mathbf{U}\, h)))$.

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

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

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*, $Fair \subseteq Powerset(S_T)$,

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

The constructed tableau $T(f)$ includes every path and every interval which satisfies f. The following theorem characterizes those paths and intervals. Notice that a state is in *Powerset*(AP_f) iff it does not contain any formulas of type $\mathbf{X}\ g$.

Theorem 7 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 $T(f)$, p $\models_{\text{path}} f$.
2. For every interval $\sigma = [t_0, ..., t_n]$ in $T(f)$, if $t_0 \in sat(f)$ and $t_n \in Powerset(AP_f)$ then $T(f), \sigma \models_{\text{int}} f$.

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 AP_f to be a subset of AP:

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

5.5.2 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 model checking algorithms 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 $T(f)$ (the proof is presented in section 5.5.5). 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* produces 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.

5.5.3 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.

Modified Quantitative Algorithm

Before proceeding, a minor modification needed in the maximum delay algorithm is presented below. This change does not affect the correctness of the algorithm, but is necessary in order to allow selective quantitative analysis to be performed over intervals.

```
proc $max$ ($start, final, not\_final$)
if ($start \cap (final \cup not\_final) = \emptyset$) then return $\infty$;
$i = 0$;
$R$ = true;
$R' = not\_final$;
while ($R' \neq R \wedge R' \cap start \neq \emptyset$) do
    $i = i + 1$;
    $R = R'$;
    $R' = T^{-1}(R') \cap not\_final$;
if ($R = R'$)
    then return $\infty$;
    else return $i$;
```

Figure 26. Modified Maximum Delay Algorithm

The only changes are that *not_final* is now a parameter of the algorithm, and an initial conditional has been introduced. Notice that if *not_final* = $\neg$ *final* the modified algorithm behaves exactly as the original one. The only case in which *not_final* is not the same as $\neg$*final* is 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*. The initial conditional states that if no starting state is in *final*, or leads to *final*, the algorithm returns infinity, as expected.

We will use a special formula *prop* to identify the set of tableau states that contain only atomic propositions.

$$prop = \{\ s \in S_T \mid s \in Powerset(AP_f)\ \}$$

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 χ to identify the set of states over which the *maximum* algorithm is computed.

$$\chi = \neg final_p \wedge \mathbf{E}[\neg final_p \ \mathbf{U}\ (prop_p \wedge final_p)\ ]$$

States in χ lead to states that are endpoints of intervals satisfying f (states in $prop_p$, see theorem 7), and that are also 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 model checking algorithms on P to identify the set of states that satisfy the CTL formula χ.
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, χ) 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 is presented in section 5.5.5.

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 χ. This guarantees that the intervals considered terminate at *final* without the need to require that the transition relation is total.

Selecting Intervals using Formula Translation

There is another method that can be used to perform selective quantitative analysis over intervals. Given a formula f it is possible to translate it into formula f' such that f holds on an interval $\pi = [s_0, s_1, ..., s_n]$ iff f' holds on paths which have π as prefix. For example, if $f = \mathbf{F}\ cond$, then $f' = start \rightarrow (\neg final\ \mathbf{U}\ cond)$. It is possible then to perform selective quantitative analysis over paths on formula f'.

However, performing the analysis over paths is significantly more expensive than performing it over intervals. The reason is that checking tableau properties over infinite paths require verification on fair paths, and computing the fairness constraints for all temporal operators in the formula is very expensive. Tableau properties over intervals, on the other hand, require no fairness, since intervals are finite.

Intuitively we can see that it is easier to determine interval satisfaction because it does not depend on the interval suffix, and as soon as the end of the interval is identified, verification stops. Path satisfaction, on the other hand must be guaranteed for infinite paths, and the early stop condition does not apply. In our practical experiments using formula translation instead of verification over intervals resulted in a slow down of about 5 to 6 times.

5.5.4 Interval Model Checking

For a given a structure M and two set of states *start* and *final*, an interval $\sigma = [s_0, ..., s_n]$ 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 a 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. An interval $\sigma = [s_0, ..., s_n]$ 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*.

Interval model checking is useful in verifying *periodic* behavior of a system. A typical example is a behavior that 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 during 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 *minimum* delay algorithm.

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 infinity then there is no pure interval from *start* to *final* that satisfies $\neg f$. Thus, every such interval satisfies f.

5.5.5 Correctness of the Algorithms

Correctness of the Tableau Construction

In this section we prove the properties of the tableau, as stated in theorem 7. There are two cases to consider, for infinite paths and for finite intervals. The properties of the tableau with respect to infinite paths can be found in [22] and will not be repeated here. In this section we prove only the properties related to finite intervals.

Given a structure M and a tableau $T(f)$ for $f \in$ LTL, the *product* P of M and $T(f)$ is a structure in which the intervals from *sat(f)* to *prop* correspond to the intervals of M that satisfy f. In fact, intervals in the product have a closer relation with intervals in M and T_f:

Lemma 8 $\tau'' = (s_0,t_0),(s_1,t_1), \ldots$ is a path or an interval in P with $L_P((s_i,t_i)) = L_T(t_i)$ for $i \geq 0$ iff there exist $\tau = s_0, s_1, \ldots$ in M, and $\tau' = t_0, t_1, \ldots$ in $T(f)$ with $L_T(t_i) = L_M(s_i) \cap AP_f$ for $i \geq 0$

Proof Immediate from the definition of the product.

The product can be used in two ways. If we wish to *restrict* our attention only to intervals in M from *start* to *final* that satisfy f, we can consider instead intervals from (*start* × *sat(f)*) to (*final* × *prop*) in the product structure. On the other hand, in order to prove that *all* intervals from *start* to *final* in M satisfy f, we construct the product of M with the tableau $T(\neg f)$ of $\neg f$, and show that it contains no interval from (*start* × *sat*($\neg f$)) to (*final* × *prop*).

Below, we state more precisely the properties of the tableau ensuring that the product has the required correspondence, and prove their correctness.

In order to identify the intervals of the tableau that satisfy f, we would like to have a lemma of the form:

$$[t_i, \ldots t_n] \models_{\text{int}} f \text{ iff } t_i \in sat(f) \wedge t_n \in prop$$

Unfortunately, only one direction of this statement holds, i.e., $t_i \in sat(f) \wedge t_n \in prop$ implies that $[t_i, \ldots t_n] \models_{\text{int}} f$, but there are other intervals that also satisfy f.

It turns out, however, that the intervals from *sat(f)* to *prop are* sufficient in the sense that for every structure M and every interval in M that satisfies f, there is a corresponding interval in $T(f)$ that starts at *sat(f)* and ends in *prop*. Hence, for our purposes it is sufficient to focus solely on these intervals.

Our proof is structured as follows. Theorem 11 proves that if an interval in the tableau starts in a state that satisfies *sat(f)* and ends in a state in *prop* then it satisfies *f*. Theorem 16 proves that every interval in *M* that satisfies *f* corresponds to an interval in *T(f)* that starts in *sat(f)* and ends in *prop*. The proof uses the following steps. For an interval $[s_i, ..., s_n]$ in *M* we define a sequence $[s^*_i, ...\ s^*_n]$ and show that each s^*_j is a state in the tableau (Lemma 12). We notice that by the definition of s^*_j, the last element in the sequence, s^*_n, contains only atomic propositions (Lemma 13). Next, we show that $[s_i, ..., s_n] \models_{\text{int}} f$ iff $s^*_i \in sat(f)$ and $s^*_n \in prop$ (Lemma 14). Finally, we prove that there is a transition between s^*_j and s^*_{j+1} (Lemma 15), thus $[s^*_i, ..., s^*_n]$ is an interval in *T(f)*. Altogether this implies Theorem 16.

We start with Lemma 9 and Lemma 10 that prove two technical results, needed in later proofs. These results help to relate the definition of satisfaction $\models$ with the definition of the set *sat* for formulas of the form $f\ \mathbf{U}\ g$.

Lemma 9 $[s_i, ...,s_n] \models f\ \mathbf{U}\ g$ iff either $[s_i, ...,s_n] \models g$ or $[s_i, ...,s_n] \models f$ and $[s_i, ...,s_n] \models \mathbf{X}(f\ \mathbf{U}\ g)$.

Proof We first show that if either $[s_i, ..., s_n] \models g$ or $[s_i, ..., s_n] \models f$ and $[s_i, ..., s_n] \models \mathbf{X}(f\ \mathbf{U}\ g)$, then $[s_i, ...,s_n] \models f\ \mathbf{U}\ g$, i.e., there exists $i \leq k \leq n$ such that $[s_k, ...,s_n] \models g$ and for all $i \leq j < k$, $[s_j, ..., s_n] \models f$.

If $[s_i, ..., s_n] \models g$ the result immediately holds for $k = i$. Otherwise, assume $[s_i, ..., s_n] \models f$ and $[s_i, ..., s_n] \models \mathbf{X}(f\ \mathbf{U}\ g)$. The latter implies that $i < n$ and that $[s_{i+1}, ..., s_n] \models f\ \mathbf{U}\ g$, i.e., there exists $i+1 \leq k \leq n$ such that $[s_k, ..., s_n] \models g$ and for all $i+1 \leq j < k$, $[s_j, ..., s_n] \models f$. In addition, we have $[s_i, ..., s_n] \models f$, thus we get the required result.

For the other direction, let $[s_i, ..., s_n] \models f\ \mathbf{U}\ g$. If $[s_k, ..., s_n] \models g$ for $k = i$ then the result immediately holds. Otherwise, assume that there is $i+1 \leq k \leq n$ such that $[s_k, ..., s_n] \models g$ and for all $i \leq j < k$, $[s_j, ..., s_n] \models f$.

This implies in particular that, $[s_i, ..., s_n] \models f$. It also implies that $[s_{i+1}, ..., s_n] \models f\,\mathbf{U}\,g$. Thus, $[s_i, ..., s_n] \models \mathbf{X}(f\,\mathbf{U}\,g)$, as required.

Lemma 10 Let $[t_i, ..., t_n]$ be an interval in $T(f)$ such that $t_n \in prop$. Let $g_1\,\mathbf{U}\,g_2$ be a subformula of f. Then, $t_i \in sat(g_1\,\mathbf{U}\,g_2)$ iff there is a $i \le k \le n$ such that $t_k \in sat(g_2)$ and for every $i \le j < k$, $t_j \in sat(g_1)$.

Proof For the first direction assume that $t_i \in sat(g_1\,\mathbf{U}\,g_2)$. We prove the required result by induction on the number of states in the interval $[t_i, ..., t_n]$.

Basis: Let $n{=}i$, i.e., $t_n \in sat(g_1\,\mathbf{U}\,g_2)$. If $t_n \notin sat(g_2)$ then by the definition of $sat(g_1\,\mathbf{U}\,g_2)$, $t_n \in sat(\mathbf{X}(g_1\,\mathbf{U}\,g_2))$. However, $t_n \in prop$, thus $t_n \in sat(g_2)$ and the claim holds for $k = i$.

Induction step: Assume the claim holds for intervals of length r. Further assume that $t_i \in sat(g_1\,\mathbf{U}\,g_2)$ for an interval $[t_i, ..., t_n]$ of length r+1. If $t_i \in sat(g_2)$ then we are done. Otherwise, if $t_i \notin sat(g_2)$ then by the definition of $sat(g_1\,\mathbf{U}\,g_2)$, $t_i \in sat(g_1)$ and $t_i \in sat(\mathbf{X}(g_1\,\mathbf{U}\,g_2))$. By the definition of R_T we have that $t_{i+1} \in sat(g_1\,\mathbf{U}\,g_2)$. We can apply the induction hypothesis on the interval $[t_{i+1}, ..., t_n]$ and conclude that there is a $i{+}1 \le k \le n$ such that $t_k \in sat(g_2)$ and for every $i{+}1 \le j < k$, $t_j \in sat(g_1)$. Together with $t_i \in sat(g_1)$ this implies the required result.

For the other direction assume that there is a $i \le k \le n$ such that $t_k \in sat(g_2)$ and for every $i \le j < k$, $t_j \in sat(g_1)$. We prove that $t_i \in sat(g_1\,\mathbf{U}\,g_2)$ by induction on the number of states in the interval $[t_i, ..., t_n]$.

Basis: Let $n = i$. Then, $k = i$ and $t_i \in sat(g_2)$. By definition, $t_i \in sat(g_1\,\mathbf{U}\,g_2)$.

Induction step: Assume the claim holds for intervals of length r or less. Let $[t_i, ..., t_k]$ be an interval of length r+1. We consider two cases. If $k = i$ then $t_i \in sat(g_2)$, and consequently $t_i \in sat(g_1\,\mathbf{U}\,g_2)$. Otherwise, if $k > i$ then $t_k \in sat(g_2)$ and for every $i{+}1 \le j < k$, $t_j \in$

$sat(g_1)$. Thus, the induction hypothesis applied to the interval $[t_{i+1}, ..., t_n]$ implies that $t_{i+1} \in sat(g_1 \mathbf{U}\ g_2)$. Since $(t_i, t_{i+1}) \in R_T$, $t_i \in sat(\mathbf{X}(g_1 \mathbf{U}\ g_2))$. But also $t_i \in sat(g_1)$, which implies that $t_i \in sat(g_1 \mathbf{U}\ g_2)$, as required.

The following theorem gives a characterization of the intervals in $T(f)$ that satisfy a given subformula g of f.

Theorem 11 Let $[t_i, ..., t_n]$ be an interval in $T(f)$ and let $t_n \in prop$, then for every subformula g of f,

$$t_i \in sat(g) \text{ iff } [t_i, ..., t_n] \models g.$$

Proof We prove the following by induction on the structure of g:
for every interval $[t_i, ..., t_n]$ such that $t_n \in prop$,

$$t_i \in sat(g) \text{ iff } [t_i, ..., t_n] \models g.$$

1. g is an atomic proposition.

   Then $t_i \in sat(g)$ iff $g \in L_T(t_i)$. By definition of satisfaction, this holds iff $[t_i, ..., t_n] \models g$.

2. $g = \neg g_1$.

   $t_i \in sat(g)$ iff $t_i \notin sat(g_1)$. By the induction hypothesis, this holds iff $[t_i, ..., t_n] \not\models g_1$ iff $[t_i, ..., t_n] \models g$.

3. $g = \mathbf{X}\ g_1$.

   If $t_i \in sat(\mathbf{X}\ g_1)$ then $\mathbf{X}\ g_1 \in t_i$. Thus $n > i$ (otherwise $t_n \notin prop$) and t_i has a successor t_{i+1} in the interval. Since $(t_i, t_{i+1}) \in R_T$, $t_{i+1} \in sat(g_1)$ and by the inductive hypothesis, $[t_{i+1}, ..., t_n] \models g_1$. Thus, $[t_i, ..., t_n] \models g$.

   For the other direction, let $[t_i, ..., t_n] \models \mathbf{X}\ g_1$, then $i < n$ and $[t_{i+1}, ..., t_n] \models g_1$. By the inductive hypothesis, $t_{i+1} \in sat(g_1)$. Since $(t_i, t_{i+1}) \in R_T$, $t_i \in sat(\mathbf{X}\ g_1)$.

4. $g = g_1 \vee g_2$ - immediate.

5. $g = g_1 \mathbf{U} g_2$.

Let $t_i \in sat(g_1 \mathbf{U} g_2)$. Since $t_n \in prop$, then by Lemma 10, there is a $i \leq k \leq n$ such that $t_k \in sat(g_2)$ and for every $i \leq j < k$, $t_j \in sat(g_1)$. By the induction hypothesis we have that $[t_k, ..., t_n] \models g_2$ and for all $i \leq j < k$, $[t_j, ..., t_n] \models g_1$. Thus, $[t_i, ..., t_n] \models g_1 \mathbf{U} g_2$.

For the other direction, assume $[t_i, ..., t_n] \models g_1 \mathbf{U} g_2$. Then there exists $i \leq k \leq n$ such that $[t_k, ..., t_n] \models g_2$ and for all $i \leq j < k$, $[t_j, ..., t_n] \models g_1$. By the induction hypothesis, $t_k \in sat(g_2)$ and for all $i \leq j < k$, $t_j \in sat(g_1)$.

Since we also have that $t_n \in prop$, Lemma 10 implies that $t_i \in sat(g_1 \mathbf{U} g_2)$.

We now give the definitions and lemmas needed to show that for every structure M and for every interval in M that satisfies f, the tableau $T(f)$ contains a corresponding interval from $sat(f)$ to $prop$ that agrees with the given one on all subformulas of f. To do so, we fix an interval $[s_0, ..., s_n]$ in a structure M and define, for every $0 \leq i \leq n$,

$$s_i^* = \{ g \mid g \in el(f) \text{ and } [s_i, ..., s_n] \models g \}$$

Lemma 12 For every $0 \leq i \leq n$, s_i^* is a state in $T(f)$.

Proof This is clearly the case, since a state of the tableau is in the powerset of $el(f)$.

Lemma 13 $s_n^* \in prop$.

Proof Note that $\mathbf{X}\, g \in s_n^*$ iff (by the definition of s_n^*) $[s_n^*] \models \mathbf{X}\, g$. Since an interval of size zero does not satisfy a formula of type $\mathbf{X}\, g$, $\mathbf{X}\, g \notin s_n^*$ for every $\mathbf{X}$ g $\in el(f)$.

Lemma 14 For every $g \in el(f) \cup sub(f)$, where $sub(f)$ is the set of all subformulas of f, and for every $0 \leq i \in n$,

$$[s_i, ..., s_n] \models g \text{ iff } s_i^* \in sat(g) \text{ and } s_n^* \in prop.$$

Proof We prove the lemma by induction on the structure of g, where all elementary formulas are considered to be the basis for induction.

Basis: $g \in el(f)$.

For the first direction, assume $[s_i, ..., s_n] \models g$. Then, by the definition of s_i^*, $g \in s_i^*$ and by the definition of $sat(g)$, $s_i^* \in sat(g)$. Since $s_n^* \in prop$ always holds (by lemma 13), the proof of this direction is completed.

For the other direction assume $s_i^* \in sat(g)$ and $s_n^* \in prop$. Then by the definition of $sat(g)$, $g \in s_i^*$ and by the definition of s_i^* we have, $[s_i, ..., s_n] \models g$.

Induction step:

1. $g = \neg g_1$.

   Assume $[s_i, ..., s_n] \models \neg g_1$. Then, $[s_i, ..., s_n] \not\models g_1$. By the induction hypothesis this implies that either $s_i^* \notin sat(g_1)$ or $s_n^* \notin prop$. But by Lemma 13, $s_n^* \in prop$ always holds. Thus, $s_i^* \notin sat(g_1)$ is true, and therefore $s_i^* \in sat(\neg g_1)$.

   For the other direction, assume $s_i^* \in sat(\neg g_1)$ and $s_n^* \in prop$. Then, $s_i^* \notin sat(g_1)$ and therefore, by the induction hypothesis, $[s_i, ..., s_n] \not\models g_1$. Hence, we conclude $[s_i, ..., s_n] \models \neg g_1$.

2. $g = g_1 \vee g_2$ — straightforward.

3. $g = g_1 \mathbf{U} g_2$.

   Assume $[s_i, ..., s_n] \models g_1 \mathbf{U} g_2$. Then, by Lemma 9, either $[s_i, ..., s_n] \models g_2$ or $[s_i, ..., s_n] \models g_1$, $[s_i, ..., s_n] \models \mathbf{X}(g_1 \mathbf{U} g_2)$ and $i < n$. By the induction hypothesis (the induction hypothesis also applies to $\mathbf{X}(g_1 \mathbf{U} g_2)$, since it is an elementary formula, and therefore simpler in structure than $(g_1 \mathbf{U} g_2)$), one of the following holds:

   - $s_i^* \in sat(g_2)$ and $s_n^* \in prop$, or

- $s^*_i \in sat(g_1) \cap sat(\mathbf{X}(g_1\ \mathbf{U}\ g_2))$, $s^*_n \in prop$.

By the definition of $sat(g_1\ \mathbf{U}\ g_2)$, both items imply that $s^*_i \in sat(g_1\ \mathbf{U}\ g_2)$, and in addition that $s^*_n \in prop$.

For the other direction, assume $s^*_i \in sat(g_1\ \mathbf{U}\ g_2)$ and $s^*_n \in prop$. By the definition of $sat(g_1\ \mathbf{U}\ g_2)$ one of the following holds:

- $s^*_i \in sat(g_2)$ and $s^*_n \in prop$, or
- $s^*_i \in sat(g_1)$ and $s^*_i \in sat(\mathbf{X}(g_1\ \mathbf{U}\ g_2))$ and $s^*_n \in prop$.

By the induction hypothesis, the first item implies: $[s_i, ..., s_n] \models g_2$. The second item implies: $[s_i, ..., s_n] \models g_1$ and $[s_i, ..., s_n] \models \mathbf{X}(g_1\ \mathbf{U}\ g_2)$. By lemma 9 we conclude that $[s_i, ..., s_n] \models g_1\ \mathbf{U}\ g_2$.

Corollary $[s_0, ..., s_n] \models f$ iff $s^*_0 \in sat(f)$ and $s^*_n \in prop$.

It is also important to prove that the sequence of states $[s^*_i, ..., s^*_n]$ is indeed an interval in the tableau.

Lemma 15 For every $0 \le i \le n\text{-}1$, $(s^*_i, s^*_{i+1}) \in R_T$.

Proof Let $\mathbf{X}\ g \in el(f)$.

- $s^*_i \in sat(\mathbf{X}\ g)$ iff $\mathbf{X}\ g \in s^*_i$, by the definition of *sat*.
- iff $[s_i, ..., s_n] \models \mathbf{X}\ g$ by the definition of s^*_i.
- iff $[s_{i+1}, ..., s_n] \models g$ by satisfiability.
- iff $s^*_{i+1} \in sat(g)$, by lemma 14, since $s^*_n \in prop$.

An interval $[s_0, ..., s_n]$ in M and $[t_0, ..., t_n]$ in $T(f)$ correspond iff for every $0 \leq i \leq n$, $L(s_i) \cap AP_f = L_T(t_i)$.

Theorem 16 Let M be a structure and let $[s_0, ..., s_n]$ be an interval in M such that $[s_0, ..., s_n] \models f$. Then, there is a corresponding interval $[t_0, ..., t_n]$ such that $t_0 \in sat(f)$, $t_n \in prop$ and $[t_0, ... t_n] \models f$.

Proof Given an interval $[s_0, ..., s_n]$ in a structure M, we choose $t_i = s^*_i$. By Lemma 12 and Lemma 15 we know that $[s^*_0, ..., s^*_n]$ is an interval in $T(f)$. Since $[s_0, ..., s_n] \models f$, Lemma 14 implies that $t_0 \in sat(f)$ and $t_n \in prop$. Clearly, for every i, $L(s_i) \cap AP_f = L_T(t_i)$. Thus, since $[s_0, ..., s_n] \models f$, $[t_0, ..., t_n] \models f$ as well.

This concludes the proof of correctness of the tableau construction.

Correctness of the Algorithms with Path Selection

We now show that the *minimum* and *maximum* algorithms respectively compute the minimum and maximum lengths of all intervals in M from *start* to *final* on paths that satisfy f.

Lemma 17 For every path $\pi = s_0, s_1, ...$ in M such that $\pi \models f$ there is a path $(s_0, t_0), (s_1, t_1), ...$ in P' such that $t_0 \in sat(f)$.

Proof This lemma is a consequence of theorem 1 in [22].

Lemma 18 For every path $\pi'' = (s_0, t_0), (s_1, t_1), ...$ in π' with $t_0 \in sat(f)$, the path $\pi = s_0, s_1, ...$ in M satisfies f.

Proof Let $\pi'' = (s_0, t_0), (s_1, t_1), ...$ be a path in π'. Then, every state on π'' is in *fair*. Thus, $\pi' = t_0, t_1, ...$ is a fair path in $T(f)$. Since it also starts in $sat(f)$, $\pi' \models f$.

The path $\pi = s_0, s_1, \ldots$ in M agrees with π' on the atomic propositions in f. Therefore, $\pi \models f$ as well.

Lemma 19 [Correctness of the minimum algorithm] Let $st = (start \times sat(f)) \cap fair$, $fn = (final \times S_T) \cap fair$ and k be the value returned by *minimum*(st, fn) applied to P'.

- If $k < \infty$ then k is the size of a shortest interval from *start* to *final* in M, along a path from *start* that satisfies f.
- If $k = \infty$ then there is no interval from *start* to *final* in M along such a path.

Proof

- Assume that *minimum*(st, fn) returns k when applied to P'. Then there is a shortest interval $[(s_0, t_0), \ldots, (s_k, t_k)]$ in P' from st to fn.

  Since $(s_k, t_k) \in fn$, it is in *fair* as well and therefore it is the start of a path $(s_k, t_k), (s_{k+1}, t_{k+1}), \ldots$ such that $t_k, t_{k+1}, \ldots$ is a fair path in $T(f)$. Adding a prefix to a fair path results in a fair path. Thus, $\pi' = t_0, \ldots, t_k, t_{k+1}, \ldots$ is also fair. Moreover, it starts in $sat(f)$. Thus, by Theorem 7, π' satisfies f.

  The path $\pi = s_0, \ldots, s_k, s_{k+1}, \ldots$ in M agrees with π' on the atomic propositions of f. Therefore, if π' satisfies f so does π. Thus, $[s_0, \ldots, s_k]$ is an interval of size k from *start* to *final* in M on a path that satisfies f.
- Now assume there is a shorter interval $[u_0, \ldots, u_l]$ with $l < k$, from *start* to *final* in M along a path $U = u_0, u_1, \ldots$ that satisfies f. By lemma 17, there is a path $U'' = (u_0, v_0), (u_1, v_1), \ldots$ in π' from $sat(f)$. The interval $[(u_0, v_0), \ldots, (u_l, v_l)]$ starts at st, ends in fn and is shorter than k. This contradicts the correctness of the minimum algorithm. We therefore conclude that k is a shortest interval in M.
- Finally, we must show that if the algorithm returns infinity there are no intervals from *start* to *final* on paths that satisfy f. Assume there is one interval $[s_0, \ldots, s_n]$ from *start* to *final* along a path $s_0, s_1, \ldots$ that satisfies f. By lemma 17, there is a path $(s_0, t_0), (s_1, t_1),$

... in π' from *sat(f)*. Consequently, there is an interval $[(s_0, t_0), \ldots, (s_n, t_n)]$ in π' from (*start* × *sat(f)*) to (*final* × S_T). However, this contradicts the correctness of the minimum algorithm. We therefore conclude that no interval could exist in this case.

Lemma 20 [Correctness of the maximum algorithm] Let *st* = (*start* × *sat(f)*) ∩ *fair*, *fn* = (*final* × S_T) ∩ *fair* and *not_fair* = *fair* - *fn*. Let k be the value returned by *maximum*(*st*, *fn*, *not_final*) applied to P'.

- If $k < \infty$ then k is the size of a longest interval from *start* to *final* in M, along a path from *start* that satisfies f.
- If $k = \infty$ then there is no bound on the size of the interval from *start* to *final* in M along such paths.

Proof The proof follows the same reasoning as in the minimum algorithm and will not be repeated here for brevity.

Correctness of the Algorithms with Interval Selection

Below we show that the *minimum* and *maximum* algorithms compute the minimum and maximum lengths respectively of all intervals in M from *start* to *final* that satisfy f.

Lemma 21 An interval $[s_0, \ldots, s_n]$ in the model M satisfies f iff there is a corresponding interval $[p_0, \ldots, p_n]$ in the product P that starts in $(\{s_0\} \times sat(f))$ and ends in $(\{s_n\} \times prop)$.

Proof For the first direction, note that the existence of an interval corresponding to $[s_0, \ldots, s_n]$ in the tableau is a direct consequence of theorem 16. Lemma 8 guarantees the existence of the corresponding interval in the product which starts in $(\{s_0\} \times sat(f))$ and ends in $(\{s_n\} \times prop)$.

For the second direction, lemma 8 demonstrates the existence of intervals corresponding to $[p_0, \ldots, p_n]$ in the tableau and in the original model M. Theorem 11 shows that since the interval in the tableau starts in *sat(f)* and ends in *prop*, then it satisfies f. Therefore, because their labels agree on the propositional variables in *f*, the corresponding interval in the original model also satisfies *f*.

Lemma 22 For every interval $[(t_0,s_0), ..., (t_n, s_n)]$ in *P* with $t_0 \in sat(f)$ and $t_n \in prop$, the corresponding interval $[s_0, ..., s_n]$ in *M* satisfies *f*.

Proof Given an interval $[(t_0, s_0), ..., (t_n, s_n)]$ in *P*, lemma 8 guarantees the existence of the corresponding intervals $[s_0, ..., s_n]$ in *M* and $[t_0, ..., t_n]$ in the tableau. Because the interval in the tableau starts in *sat(f)* and ends in *prop*, theorem 11 shows that it satisfies *f*. But since the intervals in the tableau and in *M* agree on the propositional variables in *f*, if $[t_0, ..., t_n]$ satisfies *f*, so does $[s_0, ..., s_n]$.

These lemmas imply that by computing the minimum and maximum lengths of intervals in *P* from $(start \times sat(f))$ to $(final \times prop)$ the quantitative algorithms consider all the intervals in *M* from *start* to *final* that satisfy *f*, and only those. This is made precise by the following lemmas.

Lemma 23 [Correctness of the minimum algorithm] Let $st = (start \times sat(f))$, $fn = (final \times prop)$ and *k* be the value returned by *minimum(st, fn)* applied to *P*.

- $k < \infty$ is the size of a shortest interval from *start* to *final* in *M*, that satisfies *f*.
- If $k = \infty$ then there is no interval from *start* to *final* in *M* that satisfies *f*.

Proof Assume *minimum(st, fn)* is applied to *P*. By the correctness proof of the minimum algorithm (see section 5.2.1), the minimum algorithm returns the length of a shortest interval from $(start \times sat(f))$ to $(final \times prop)$ in *P*, if such an interval exists and returns infinity otherwise.

- Assume *minimum*(*st*, *fn*) returns k. By Lemma 22, there is an interval of size k in M from *start* to *final* that satisfies f. Suppose there is a shorter such interval σ. By Lemma 21 there is a corresponding interval in P, from (*start* × *sat*(f)) to (*final* × *prop*) of the same size as σ. This contradicts the correctness of the minimum algorithm. Hence, k is the size of a required interval in M.
- Assume *minimum*(*st*, *fn*) returns ∞. Further assume that there is an interval of size k from *start* to *final* in M, that satisfies f. By Lemma 21, there is an interval of the same size in P from (*start* × *sat*(f)) to (*final* × *prop*), contradicting the correctness of the minimum algorithm. We therefore conclude that no such interval exists in M.

The correctness proof of the selective maximum algorithm is based on the properties of the maximum algorithm (see section 5.2.2). The following lemma states the required property for the non-selective algorithm.

Lemma 24 If the *maximum*(*st*, *fn*, *not_final*) = k then the size of a longest interval from *st*, that lies entirely within *not_final* is k-1. If it returns ∞ then there is no bound on the size of such intervals.

Proof The proof of the theorem follows very closely the proof of the maximum algorithm in section 5.2.2, and it is only outlined here. The following two definitions are used in the proof:

- S_i is the set of states at the start of an interval with i states, all contained in *not_final*.
- M is size of a longest interval beginning in *start* and contained within *not_final*.

The correctness of the algorithm follows from the loop invariants:

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

Termination is guaranteed because there can be no infinite sequence of distinct S_i given that the state space is finite and that $S_i \supseteq S_{i+1}$.

The correctness of the return value follows from the last conditional in the algorithm. If the loop is exited because $R = R'$, then $S_i = S_{i+1}$. Since $S_{i+1} \subseteq T^{-1}(S_i)$, every state in S_{i+1} has an edge to another state in S_{i+1}. So every state in S_{i+1} is the beginning of an infinite path of states remaining in $S_{i+1} \subseteq$ *not_final*. Moreover, $R' \cap$ *start* $\neq \varnothing$ (otherwise the algorithm would have exited inside the loop). Therefore some state $s \in S_{i+1}$ belongs to *start*. This state then is the beginning of an infinite path starting at a state in *start*, which never reaches a state in *final*. Infinity is then correct return value.

If $R' \cap$ *start* $= \varnothing$, then by the invariant $R' = S_{i+1}$, we know that there is no interval of i+1 states contained in *not_final* beginning in a state in *start*. No longer interval can exist since this would contradict the absence of an interval of $i + 1$ states, so we have $M \leq i$. But we also have the invariant $i \leq M$, so it must be the case that $M = i$, which is the correct return value.

Lemma 25 [Correctness of the maximum algorithm] Let *st* = (*start* × *sat*(*f*)), *fn* = (*final* × *prop*),

$$not\_final = \neg final_p \wedge \mathbf{E}[\neg final_p \ \mathbf{U}\ (prop_p \wedge final_p)]$$

and k be the value returned by *maximum*(*st*, *fn*, *not_final*) when applied to P.

- $k < \infty$ is the size of a longest interval from *start* to *final* in M, that satisfies f.
- If $k = \infty$ then there is no bound on the sizes of the intervals from *start* to *final* in M, that satisfy f.

Proof If *maximum*(*st*, *fn*, *not_final*) returns k, then by Lemma 24 the size of a longest interval from *st* that lies entirely within *not_final* is k-1. Let $[p_0, ..., p_{k-1}]$ be such an interval. We first show that p_{k-1} has a successor p_k, and that $p_k \models prop_p \wedge final_p$. In other words, $p_k \in$ (*final* × *prop*).

The state $p_{k\text{-}1}$ is in *not_final*, thus $p_{k\text{-}1} \models \neg final_p \wedge \mathbf{E}[\neg final_p\ \mathbf{U}\ (prop_p \wedge final_p)]$. Since $p_{k\text{-}1} \models \neg final_p$, the only way for it to satisfy also $\mathbf{E}[\neg final_p\ \mathbf{U}\ (prop_p \wedge final_p)]$ is by having a successor p_k such that $p_k \models \mathbf{E}[\neg final_p\ \mathbf{U}\ (prop_p \wedge final_p)]$.

Assume that $p_k \not\models final_p$. Then, $p_k \models \neg final_p \wedge \mathbf{E}[\neg final_p\ \mathbf{U}\ (prop_p \wedge final_p)]$, i.e. $p_k \in$ *not_final*. But this contradicts the fact that $p_{k\text{-}1}$ is the last state of a longest interval. Thus, $p_k \models final$ and since it satisfies $\mathbf{E}[\neg final_p\ \mathbf{U}\ (prop_p \wedge final_p)]$, it must also satisfy $prop_p$.

Thus, if *maximum(st, fn, not_final)* returns k, k is the size of a longest interval from $st = (start \times sat(f))$ to $fn = (final \times prop)$ in P. Using arguments similar to those for the minimum algorithm we conclude that the size of the longest interval from *start* to *final* in M that satisfies f is k.

The proof in case *maximum(st, fn, not_final)* returns ∞ also follows the same reasoning as in the *minimum* case.

Correctness of the Interval Model Checking Algorithm

Lemma 26 If *minimum(st, fn)* applied to P as described in the interval model checking algorithm returns infinity, then all pure intervals in M satisfy formula f.

Proof Lemma 23 guarantees that if $minimum((start \times sat(g)), (final \times prop))$ returns ∞, then there is no interval from *start* to *final* in M that satisfies g. If $g = \neg f$, this means that all intervals from *start* to *final* in M satisfy f. We conclude that all pure intervals from *start* to *final* in M satisfy f.

5.6 Lazy Composition

The high complexity of verifying real-time and other concurrent systems arises mostly from the number of parallel components in the system. The number of states of the state-transition graph representing the model can grow exponentially with the number of concurrent components in the system. Even symbolic algorithms that do not explicitly represent individual states suffer from this exponential blowup. However, even though extremely expensive, the parallel composition algorithm is vital to verification tools, because the large majority of real systems is described by a set of concurrent processes.

Given a set of processes, each modeled by a state-transition graph, a *parallel composition algorithm* is used to create a global state-transition graph which models all processes and their interaction. There are different ways to model the interaction between processes. One common way is the *synchronized* composition model. Under this model one step in the global model corresponds to exactly one step in each process. In other words, all processes execute synchronously.

Besides synchronous composition, another common composition model is *interleaving* composition. In this model only one process executes at each step of the global model. The choice of which process executes is nondeterministic and fairness assumptions are used to avoid starvation. Interleaving composition is not well suited for modeling real-time systems. The reason is that since the choice of which process executes next is nondeterministic, there is no way to bound the execution time of a process. While fairness can be used to avoid starvation, it cannot be used to bound response time. Fairness assumptions state that some property of the model will *eventually* be satisfied, but there are no restrictions on when this must happen. As a consequence, any process can always be delayed for one extra step, effectively making all response times potentially unbounded.

In Verus the synchronous composition model is used. The symbolic parallel composition algorithm used is extremely simple: Given a set of boolean formulas R_0, R_1, ..., R_n

describing the transition relation of each process, the global transition relation R is constructed by conjuncting all R_is:

$$R = \wedge_{i = 0..n} R_i.$$

Each R_i is a formula such that $R_i(v, v')$ is true for states v and v' iff when process P_i is in state v it can transition to state v'. Consequently, the global transition relation R is a formula such that $R(v, v')$ is true for states v and v' iff *all* processes can transition from state v to state v', that is, all processes transition at the same time.

Unfortunately, the size of R can be several orders of magnitude larger than the sum of the sizes of all R_i. Some techniques exist to handle this blowup, such as *partitioned transition relations* [8]. The partitioned transition relation algorithm modifies the algorithm to compute the set of successors of a state set S. In the original algorithm S is conjuncted with the transition relation R and then the current state variables are quantified out of the result (See "The Model Checking Algorithm" on page 40.). The partitioned transition relation algorithm changes the order in which the conjunction and quantification are performed.

Partitioning the transition relation consists of dividing it into separate partitions, in much the same way as the transition relation of each process of the global system can be separated from the others. It is then possible to quantify out variables *before* conjuncting all components when computing the set of successors of a state set, provided that those variables are not used by the unprocessed partitions. In some cases significant gains can be obtained by this method, since the global transition relation is never constructed.

The problem with partitioned transition relations is that they are very sensitive to the order in which processes are considered for early quantification. As a consequence, in order to use the method efficiently the user must understand how variables interact in the model, and must choose the right order to apply early quantification.

An alternative approach is used in Verus, *lazy composition*. In the same way as partitioned transition relations the global transition relation is never constructed. However, different than the previous method a *restricted* transition relation of all processes is created at each step. The restricted transition relation agrees with the global transition relation for states in the state set of interest, but it may behave in a different way for other states. The advantage comes from the fact that in many cases it is possible to construct a restricted transition relation that is significantly smaller than the global transition relation.

There are many possible ways of constructing a restricted transition relation that would produce correct results. Given an original global transition relation R and a state set f, the computation of the set of successors of f can use any restricted transition relation R' that satisfies the following condition:

$$R'|_f = R|_f$$

The formula above means that R and R' agree on transitions that start in states in f. It is possible to represent R' with significantly fewer nodes that R in some cases by using the *constrain* operator from [26]. For two boolean formulas f and g, $f' = constrain(f, \text{g})$ is a formula that has the same truth value as f for variable assignments that satisfy g. If the variable assignment does not satisfy g, the value of f' is not determined. In many cases the size of f' is significantly smaller than the size of f.

The lazy composition algorithm uses the *constrain* operator to simplify the transition relation of each process before generating the global restricted transition relation. When computing the set of successors of a state set S (represented by a boolean formula) the algorithm computes:

$$R' = \wedge_{i=0..n}\, constrain(R_i, S)$$

Each transition $R'_i = constrain(R_i, \text{S})$ agrees with R_i on transitions that start in S by the definition of the *constrain* operator. As a consequence, the transition relation R' agrees with the global transition relation R on transitions that start S as well. Therefore, computing the set of successors of S using R' produces the same result as using R. The same method can be applied when computing the set of predecessors of a state set.

Lazy Composition

The *constrain* operator has been initially used to simplify the set of states visited during verification in a technique called frontier set simplification. This work is described in [9,26]. The same operator has also been applied to simplify the transition relation in a more restricted way as described in [62,73]. An important difference between these methods and ours is that the structure of the Verus language makes it natural to partition the transition relation into processes. In a well designed program the variables that interact more closely are usually in the same process, consequently sharing the same partition. This makes the method very effective.

We have implemented the lazy composition algorithm and obtained significant gains in space and time during verification. In one example the verification was previously performed in 40 seconds using 12 megs of memory, which dropped to 18 seconds and 1 meg of memory using the proposed method. The same example verified with partitioned transition relations used about the same time, but twice the memory used by the lazy composition algorithm.

A significant part of the savings come from not constructing the global transition relation. These savings were also present in the partitioned transition relation case. However the new method used much less memory. The reason seems to be that partitioned transition relations are heavily influenced by the order in which partitions are processed, because this order determines which variables can or cannot be quantified out early. In the proposed method this does not happen, all variables are quantified out at the same time. This makes it less susceptible to the order in which partitions are processed, and more suitable to be used in the cases in which determining the processing order can be difficult. It also makes the new technique easier to automate.

Chapter 6 Analyzing Real Systems

The Verus approach is a practical one. It can be (and has been) applied to real problems. This chapter describes several examples that have been verified and the results produced by the technique. The presentation not only demonstrates the usefulness of the method proposed, but also explains how the tool can be used and how results can be interpreted. In many cases our analysis has been able not only to determine correctness, but also to uncover subtleties in the behavior of the system being verified and to suggest optimizations. These examples can be used as a guide to verifying other systems. The method is by no means restricted to produce the types of information described, but hopefully the examples presented in this chapter can serve as a starting point to perform similar and even more complete analyses in the future.

6.1 A Priority Inversion Example

Priorities are essential in real-time systems. The correct ordering of task execution is a fundamental problem that must be solved if the system is to be predictable. Many scheduling policies have been developed to define what constitutes a correct ordering and to enforce this ordering during the execution of the system. Most scheduling policies require that higher priority tasks execute before lower priority tasks. However, even in this case, it is sometimes possible for a low priority process to be executing while a higher priority one is blocked. This situation is called *priority inversion* [66]. Unbounded priority inversions occur when high priority processes are blocked indefinitely by low priority processes. When this happens, the system becomes unpredictable. The correct ordering of task execution will be compromised, and the system may fail to satisfy its specification.

In order to present the problem in a more concrete framework, we will introduce a hypothetical air-traffic control system. This example is not associated with a real system, but illustrates a problem that can affect virtually any real-time system and cause it to become unschedulable. We will concentrate our analysis in two of the processes in the system. The first, called *sensor*, reads airplane position data from radars, sets alarms on catastrophic conditions (conditions that cannot wait for a detailed analysis), and puts the data into shared memory. The other process is the *reporter*. It reads the data collected by the *sensor*, and updates the traffic controller screens. The *sensor* is a high priority process since it processes urgent events, and must not be blocked by other processes. The *reporter* on the other hand, is a low priority process. Since it doesn't process urgent events, it may be delayed by other more important tasks.

The *sensor* and the *reporter* processes share data. To access shared data appropriately, synchronization is necessary. In our system, synchronization is implemented by a mutex variable which guarantees mutual exclusion among the processes accessing the data. The mutex variable is locked every time shared data is accessed. However, this may result in priority inversion, as shown in the following scenario. Suppose the *reporter* is inside the critical section, and the *sensor* tries to insert new data into the buffer area. The *sensor* can't access the data and blocks, since it is waiting until the *reporter* unlocks the mutex. At this point a high priority process is waiting for a low priority one, and priority inversion occurs. This situation is shown in figure 27. This figure shows which process is executing at any time. The shaded areas indicate that the process is accessing the mutual exclusion area, and the arrows indicate requests for and releases of mutexes.

This priority inversion scenario is bounded, the *sensor* cannot be blocked indefinitely. The *reporter* will delay the *sensor* only while it is inside the critical section. After the *reporter* releases the lock, the *sensor* will start executing, and the priority inversion will disappear. We can calculate the maximum duration of the priority inversion as the time to execute the largest critical section, and incorporate it in our calculations for the execution times. The system will still be predictable, although there may be a little loss in accuracy in execution time predictions. Consequently, if the system is well designed, and the critical sections are small, bounded priority inversions can be tolerated without sacrificing predictability.

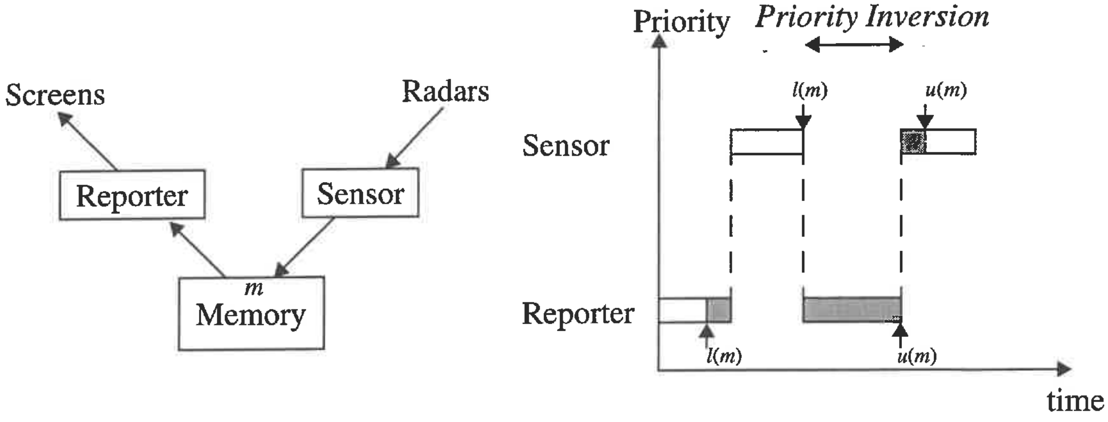


Figure 27. Bounded priority inversion

In certain cases it is possible to have unbounded priority inversions that cannot be solved by this simple method. Suppose a third process, called the *analyzer* is added to the system. This process reads data generated by other components of the air-traffic controller and processes it. The *analyzer* is less important than the *sensor* and has a lower priority. But it is more important than the *reporter,* since urgent conditions may arise as the result of the analysis and handling them is more important than updating the screens. Consider now the same scenario as above, with the *reporter* inside the critical section, and the *sensor* waiting on the mutex. At this point, the *analyzer* starts executing. It will block the *reporter,* since it has higher priority. However, the *sensor* is waiting for the *reporter* (and therefore also for the *analyzer*). Since the *analyzer* doesn't know the relation between the *reporter* and the *sensor*, it may execute for an unbounded amount of time and delay the *sensor* indefinitely. If a catastrophic event occurs, it will go unnoticed, because the *sensor* is blocked. The behavior of the system becomes unpredictable. This unbounded priority inversion can be seen in figure 28.

Priority inheritance protocols are one way of preventing unbounded priority inversions [66]. A typical protocol might work in the following manner. As soon as a high priority process is blocked by a low priority one, the low priority process is temporarily given the priority of the blocked process. In our example, while inside the critical section the *sensor* is trying to access, the *reporter* will execute at high priority. When the *reporter* exits the critical section, it will be restored to its original priority. In this way, the *analyzer* will not be able to interrupt the *reporter* when the *sensor* is waiting. We will show that this protocol avoids the unbounded priority inversion problem (except possibly for deadlocks in accessing synchronization variables). This allows the designer of the system to predict the maximum priority inversion time, as in the bounded case.

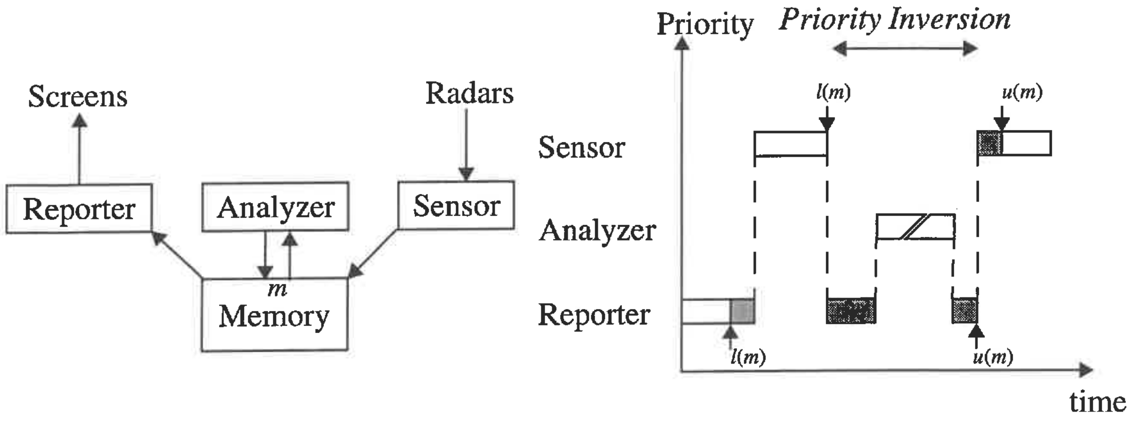


Figure 28. Unbounded priority inversion

Priority inversion occurred in this example because the *analyzer* preempted the *reporter.* Another cause of priority inversion is queueing. Communication protocols may experience priority inversion for this reason. For example, packets to be sent to the network may have priorities. Low priority packets may be enqueued ahead of high priority ones in some protocol queue. In a prioritized network a high priority packet may have to wait for a low priority one to be sent. If medium priority packets start arriving in another processor's queue, they may monopolize the network, preventing high priority packets from being sent. Again, we have unbounded priority inversion. This type of priority inversion could also happen in our system, if the different components were distributed over a network. For example, *sensor* packets could be queued after some low priority packets in a queue, while *analyzer* packets were being transmitted.

The inheritance mechanism that we have described to avoid unbounded inversions is called basic priority inheritance protocol. There are other priority inheritance protocols. Some protocols are designed to avoid deadlocks caused when critical sections are accessed in the wrong order. Other protocols are designed to handle *chained bounded priority inversions.* A chained inversion occurs when a high priority process wants to lock *n* mutexes that are already locked by low priority processes. In this case, the high priority process has to wait for all low priority processes to finish their critical sections. While this wait is bounded, it may be too expensive to wait for the duration of all critical sections. One possible solution to this problem is to assign priorities to critical sections, based on the priorities of the processes that may access it. A process is allowed to access a critical section only if its priority is higher than the priority of all critical sections currently being accessed. A more complete study of these various algorithms and their characteristics can be found in [66].

We have used Verus to implement and analyze a real-time system that is susceptible to priority inversion. The example follows the *sensor—analyzer—reporter* paradigm presented above, but with some important differences. There is no scheduler choosing which process runs next, all processes run concurrently. However, a mutex control module chooses which process will lock the variable next, and when there is contention, high priority processes are chosen first. In this case the queueing of processes in the mutex control module may cause priority inversion. In the example we have the same three processes as above, the *sensor*, the *analyzer* and the *reporter*, with high, medium and low priorities, respectively. There are also two mutex variables, `M1` and `M2`, controlling two critical sections. The *sensor* uses the critical section controlled by `M1`, the *analyzer* uses the one controlled by `M2`, and the *reporter* locks both variables, first `M1` and then `M2`. The mutex `M1` controls access to the area shared by the *sensor* and the *reporter* as explained above. The mutex `M2` controls a shared area where the *analyzer* puts its results. These results will in turn be read by the *reporter* to be printed on the screen. The Verus code that implements the system is given below. We start by presenting the simplest version that suffers from priority inversion, and then proceed to change the design to correct the problem.

```
sensor(M1, req)
int M1;
boolean req;
{
  extern boolean proceed;
  boolean start, finish;

  req = false;
  start = false; finish = false;
```

The parameters and local variables of the *sensor* are declared above. The parameter `M1` is the first mutex variable, and `req` is used by *sensor* to request access to `M1`. The variable `proceed` is used to decide when the process will try to lock `M1`. It can wait indefinitely outside the critical region, but it can also decide at any time to proceed. By declaring `proceed` as an external variable, we allow it to change non-deterministically, effectively modeling the desired behavior, as will be seen shortly. The variables `start` and `finish` are used to flag the beginning and end of execution of *sensor*.

The *sensor* executes continuously. Line 9 starts an infinite loop inside which *sensor* executes. Line 10 stops the process until the variable `proceed` is true. Since proceed is an external variable, it can become true at any time. The process will then wait for a non-deterministic amount of time before proceeding. This models the behavior that *sensor* may decide to access the critical region at any time, or never at all.

```
 9   while (true) {
10     while (!proceed) wait(1);
```

Once it decides to proceed, *sensor* flags that it wants to access the mutual exclusion region in line 11 by asserting the variable `req`. This variable is used by the mutex control module to decide which process accesses the mutual exclusion region first. In line 12 the process flags that it has started executing and waits until the mutex variable `M1` indicates that it has been granted the mutual exclusion region (line 13). The process `rt_mutex` (described below) guarantees that only one process at any point will be granted mutual exclusion.

```
11     req = true;
12     start = true;
13     while (M1 != 1) {wait(1); start = false; };
```

Lines 14 to 16 correspond to the *sensor* accessing the critical region for three time units. Notice that the variable `start` is asserted in line 12 and deasserted either in line 13 or 15. This pattern guarantees that `start` will be true for only one time unit, when *sensor* decides to execute. It is important for the computation of accurate response times that it is asserted for only one time unit. For example, if `start` is not deasserted in line 13 after the `wait`, `start` would correspond to the time interval between requesting the mutual exclusion until it is granted, and not the time instant that the process starts executing.

```
14     wait(1);               /* M1 is locked */
15     start = false;
16     wait(2);
```

Line 17 signals the mutex control module that *sensor* does not require mutual exclusion anymore. Line 18 flags that it has ended its execution. Line 20 is used to guarantee that `finish` is also asserted for only one time unit. Line 19 is important to make the effect of line 18 observable to other processes. If there was no `wait` statement in line 19, the variable `finish` would be asserted and deasserted in the same cycle, nullifying the assertion.

```
17      req = false;
18      finish = true;
19      wait(1);                /* M1 is unlocked */
20      finish = false;
21    };
22  }
```

The *analyzer* process is shown above. It is very similar to the *sensor* process, except that it requests the mutex `M2` instead of `M1`.

```
23  analyzer(M2, req)
24  int M2;
25  boolean req;
26  {
27    extern boolean proceed;
28    boolean start, finish;
29
30    req = false;
31    start = false; finish = false;
32    while (true) {
33      while (!proceed) wait(1);
34      req = true;
35      start = true;
36      while (M2 != 2) {wait(1); start = false;};
37      wait(1);                /* M2 is locked */
38      start = false;
```

```
    wait(2);
    req = false;
    finish = true;
    wait(1);            /* M2 is unlocked */
    finish = false;
  };
}
```

The *reporter* has a similar structure to both the *sensor* and the *analyzer*. The main difference is that it accesses both M1 and M2.

```
reporter(M1, M2, reqM1, reqM2)
int M1, M2;
boolean reqM1, reqM2;
{
  extern boolean proceed;
  boolean start, finish;

  reqM1 = false; reqM2 = false;
  start = false; finish = false;
  while (true) {
    while (!proceed) wait(1);
```

Initially, the *reporter* requests M1.

```
    reqM1 = true;
    start = true;
    while (M1 != 3) {wait(1); start = false;};
    wait(1);            /* M1 is locked */
    start = false;
```

Once M1 is locked, the *reporter* locks M2.

```
62      reqM2 = true;
63      while (M2 != 3) wait(1);
64      wait(3);            /* M2 is locked */
```

Finally, it unlocks both mutexes.

```
65      reqM2 = false;
66      reqM1 = false;
67      finish = true;
68      wait(1);            /* both are unlocked */
69      finish = false;
70    };
71  }
```

The process that controls access to the mutual exclusion regions is shown next. It has as outputs the mutex variables `M1` and `M2`. The inputs are the requests from all processes, `s_reqM1` comes from the *sensor*, `a_reqM2` comes from the *analyzer*, and `r_reqM1` and `r_reqM2` come from the *reporter*.

```
72  rt_mutex(M1, M2,
73           s_reqM1, a_reqM2, r_reqM1, r_reqM2)
74  int M1, M2;
75  boolean s_reqM1, a_reqM2, r_reqM1, r_reqM2;
76  {
```

Initially both mutexes are unlocked. The process then enters an infinite loop in which it receives requests and grants mutexes. The decision is based on a straightforward policy, if no one is requesting, the mutex stays unlocked (line 80). If only one process is requesting, the mutex is granted to it (lines 81 and 82). If both are requesting (line 83) the mutex is granted to process 1 (the *sensor*), since it has higher priority. The lower priority process may starve according to this policy, but this is allowed in real-time resource management. Notice that if there is contention we must wait until the current owner of the mutex releases it before changing its value. This is needed because mutual exclusion regions cannot be preempted.

```
  M1 = 0;
  M2 = 0;
  while (true) {
    if (!s_reqM1 && !r_reqM1) M1 = 0; else
    if ( s_reqM1 && !r_reqM1) M1 = 1; else
    if (!s_reqM1 &&  r_reqM1) M1 = 3; else
    if (M1 == 0)              M1 = 1;
```

The `rt_mutex` process is completed by handling `M2` in the same way as `M1`.

```
    if (!a_reqM2 && !r_reqM2) M2 = 0; else
    if ( a_reqM2 && !r_reqM2) M2 = 2; else
    if (!a_reqM2 &&  r_reqM2) M2 = 3; else
    if (M2 == 0)              M2 = 2;
    wait(1);
  };
}
```

Finally, `main` declares the mutex variables and instantiates all processes.

```
main()
{
  int M1, M2;
  boolean s_reqM1, a_reqM2, r_reqM1, r_reqM2;

  process
    p1 sensor(M1, s_reqM1),
    p2 analyzer(M2, a_reqM2),
    p3 reporter(M1, M2, r_reqM1, r_reqM2),
100       p0 rt_mutex(M1, M2, s_reqM1, a_reqM2,
101                      r_reqM1, r_reqM2);
102   }
```

Stuttering

The program presented above implements the system described. However, one of the characteristics of the model generated from this program differs from the actual system. In Verus all processes are synchronized, that is, one step of the global model corresponds to exactly one step in each process. In the actual implementation of the system this is not a correct assumption, processes execute asynchronously. This has important consequences in the behavior of the model. For example, in the model above unbounded priority inversion does *not* occur, it is avoided by the implicit dependencies introduced by the synchronization of the modules. However, these dependencies do not exist in the real system, and must be eliminated. This is accomplished by a technique called *stuttering*. It has the effect of making a transition in the model take a nondeterministic number of time units to occur, eliminating implicit synchronization dependencies. An improved `rt_mutex` process is shown below that models the correct asynchronous behavior. The integer variable `stut` is used to determine when mutex decisions should take place. A decision about granting the mutex *must* be made if `stut` is zero, but it may or may not be made otherwise. Since `stut` is continuously incremented (modulo 10), a decision can be delayed up to 10 time units, but no longer.

The difference between the original `rt_mutex` and the new one is the use of the `stut` variable. Lines 81 and 82 have been replaced, now the decision about granting the mutex depends on the value of `stut`.

```
72  rt_mutex(M1, M2,
73           s_reqM1, a_reqM2, r_reqM1, r_reqM2)
74  int M1, M2;
75  boolean s_reqM1, a_reqM2, r_reqM1, r_reqM2;
76  {
```

```
int stut;

M1 = 0;
M2 = 0;
while (true) {
  if (!s_reqM1 && !r_reqM1) M1 = 0; else
  if ( s_reqM1 && !r_reqM1 && M1 != 1) {
    if (stut == 0) M1 = 1; else
                   M1 = select{0, 1};
  } else
  if (!s_reqM1 &&  r_reqM1 && M1 != 3) {
    if (stut == 0) M1 = 3; else
                   M1 = select{0, 3);
  } else
  if (M1 == 0)              M1 = 1;
```

Mutex M2 is handled in the same way.

```
  if (!a_reqM2 && !r_reqM2) M2 = 0; else
  if ( a_reqM2 && !r_reqM2 && M2 != 2) {
    if (stut == 0) M2 = 2; else
                   M2 = select{0, 2};
  } else
  if (!a_reqM2 &&  r_reqM2 && M2 != 3) {
    if (stut == 0) M2 = 3; else
                   M2 = select{0, 3};
  } else
  if (M2 == 0)              M2 = 2;
```

Finally, the value of `stut` is incremented at each step as seen below. The value is continuously incremented modulo 10.

```
        if (stut < 10) stut = stut + 1; else
                       stut = 0;
        wait(1);
    };
}
```

We have analyzed the model above using RTCTL model checking and the quantitative algorithms described. The first property verified is mutual exclusion. One way to check if the access to the critical sessions is exclusive is to create variables that are asserted when the process enters the critical region, and deasserted when it leaves the region. These variables are similar to the `start` and `finish` variables presented above. We can then write a CTL formula that states that no two of these variables are asserted at the same time.

There is another way of checking the same property without adding variables to the model. Notice that each process is inside the critical region when it goes through some specific wait statements in the program. The sensor processor is inside the critical region controlled by `M1` when it executes the wait statements 3, 4 and 5 in the source code. The analyzer is inside `M2` when it executes waits 3, 4 and 5. The reporter is inside `M1` when it executes the waits between 3 and 7, and inside `M2` when it executes waits 5 to 7.

Since the wait counters are variables in the model, it is possible to write CTL formulas that reference them and check to see if any critical region is being violated. The region controlled by `M1` can be checked by the formula:

AG !(*sensor_in_M1* && *reporter_in_M1*)

where *sensor_in_M1* is

```
    (sensor.wc == 3 || sensor.wc == 4 || sensor.wc == 5)
```

and *reporter_in_M1* is

```
(reporter.wc == 3 || reporter.wc == 4 || reporter.wc == 5 ||
           reporter.wc == 6 || reporter.wc == 7))
```

Finally, the region controlled by `M2` is checked by the formula:

$$\texttt{AG !(}\mathit{analyzer\_in\_M2}\ \texttt{\&\&}\ \mathit{reporter\_in\_M2}\texttt{)}$$

where *analyzer_in_M2* is

```
(analyzer.wc == 3 || analyzer.wc == 4 || analyzer.wc == 5)
```

and *reporter_in_M2* is

```
(reporter.wc == 5 || reporter.wc == 6 || reporter.wc == 7)
```

Properties about the response time of the processes in the model have also been checked. We have found that the *analyzer* has a bounded response time, the following property is true of the model:

$$\mathbf{AG}(\mathit{analyzer.start} \rightarrow \mathbf{AF}_{\leq 15}\ \mathit{analyzer.finish})$$

However, the *sensor* process can starve. The following sequence of events leads to an unbounded priority inversion:

- The *reporter* locks `M1`.
- The *analyzer* locks `M2`.
- The *sensor* wants to lock `M1`, but it is already locked, so it waits.
- The *reporter* wants to lock `M2`, but it is locked, so it waits.
- The *analyzer* is continuously generating data, and after unlocking `M2`, it locks the mutex again to insert new data into the buffer. The *reporter* never locks `M2`, since it has lower priority than the *analyzer.*
- The *sensor* is waiting for the *reporter*, and the *reporter* is waiting indefinitely for the *analyzer.* Therefore, the *sensor* is blocked by the *analyzer* indefinitely.

We have implemented the basic priority inheritance protocol described previously to solve this problem. The solution works as follows. Whenever the *sensor* is waiting for the *reporter*, the task being executed by the *reporter* becomes a high priority task. We then make the *reporter* a high priority process temporarily, so it will release the lock the *sensor* wants faster. The *analyzer* eventually notices that the *reporter* has become a high priority process. At this point it will yield M2 to the *reporter.* After unlocking M1, the *reporter* will have its old priority restored.

To implement priority inheritance we can easily change the Verus program above. A boolean variable M2inherit is declared. It will signal when the priority inheritance mechanism must be activated. The initial value for this variable is false, and it becomes true whenever the *sensor* tries to lock M1. In the code for the *sensor* process, lines 13 and 14 are replaced by:

```
13      while (M1 != 1) {
14        M2inherit = true;
15        wait(1);
16        start = false;
17      };
18      wait(1);                /* M1 is locked */
19      M2inherit = false;
```

In the rt_mutex process we must make sure that whenever M2inherit is true, the reporter is given priority over the analyzer. This is implemented by replacing line 99 in the rt_mutex code by:

```
 99     if (M2 == 0) {
100       if (M2inherit) M2 = 3; else
101                      M2 = 2;
102     };
```

The modifications proposed guarantee that there is no unbounded priority inversion in the system. The *sensor* now has a bounded response time, as attested by the following property of the model:

$$\mathbf{AG}(\mathit{sensor.start} \rightarrow \mathbf{AF}_{\leq 30}\, \mathit{sensor.finish})$$

In fact, by using quantitative analysis we have determined the exact best and worst case response times for the processes:

Process	Min. time	Max. time
sensor	3	26
analyzer	3	∞
reporter	4	∞

Conclusions

This example shows that priority inversion is an important problem that can affect the behavior of real-time systems in subtle but significant ways. It can make even simple systems unpredictable. In the example we also discuss a solution, priority inheritance, and show how it can make the system predictable by bounding the maximum priority inversion time.

We also demonstrate in this example how the Verus language can be used to specify and analyze a real-time system. The Verus code for the system is fully presented as well as its analysis. It shows how Verus can be used to describe and analyze complex systems. The description is detailed enough to hopefully serve as a starting point for writing programs describing similar systems. The analysis performed is very straightforward, but this is by no means a restriction on the method. The following sections will describe more sophisticated examples and analyses.

6.2 An Aircraft Controller

One of the most critical applications of real-time systems is in aircraft control. It is extremely important that time bounds are not violated in such systems. This section briefly describes an aircraft control system used in military airplanes. We have attempted to make this model as realistic as possible. It is shown how some of its timing constraints can be checked using the quantitative algorithms described.

System Description

The control system for an airplane can be characterized by a set of sensors and actuators connected to a central processor. This processor executes the software to analyze sensor data and control the actuators. Our model describes this control program and defines its requirements so that the specifications for the airplane are met. The requirements used are similar to those of existing military aircraft, and are derived from those described in [60].

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

- Navigation: Computes aircraft position. Takes into account data such as speed, altitude, and positioning data received from satellites or ground stations.
- Radar Control: Receives and processes data from radars. It also identifies targets and target position.
- Radar Warning Receiver: This system identifies possible threats to the aircraft.
- Weapon Control: Aims and activates aircraft weapons.
- Display: Updates information on the pilot's screen.
- Tracking: Updates target position. Data from this system are used to aim the weapons.
- Data Bus: Provides communication between processor and external devices.

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

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

[1] Weapon protocol is an aperiodic process with a deadline of 200 ms.

[2] Weapon release has a period of 200 ms, but its deadline is 5 ms.

Figure 29. Timing requirements for aircraft controller

Concurrent processes are used to implement each subsystem. Communication among the various processes is done indirectly. No data are shared directly by two subsystems. Processes communicate only through data servers called *monitor tasks.* Each system maintains a server process that accepts requests for data, and returns the desired information. The various subsystems in each system update the data in the servers. Monitor tasks only accept requests, respond to them, and then block. They are assigned low priority, and priority inheritance is used to maintain predictability [11,66].

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

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

In order to enforce the different timing constraints of the processes, priority scheduling is used. Predictability is guaranteed by scheduling the processes using *Rate Monotonic Scheduling* (RMS) [50,59].

Model of the Aircraft Control System

We have modeled this control system in the Verus system. Model checking has been used to verify its functional correctness, while its timing correctness has been checked using the quantitative algorithms described previously. Most of the characteristics described above were implemented, although some abstractions have been performed for simplicity. A more detailed description of the implementation follows.

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

The time to request data from a monitor task and wait for the response is assumed to be small compared to the total execution time. This is reasonable if we assume an efficient implementation. Sending request and response messages takes only a small amount of time. Processing in the monitor tasks is also fast, considering the limited range of functions performed. The assumption can only be violated if blocking due to synchronization is long. The access pattern to the monitor tasks, however, minimizes this possibility. They simply receive requests, retrieve data from memory, and return it. There are no nested critical sections. Moreover, the priority inheritance protocols used maintain predictability and eliminate the possibility of unbounded blocking due to synchronization [11,66]. Since blocking times can be computed, we assume they are included in the execution time defined. A more detailed model can be constructed to remove this assumption, but because of the reasons outlined above, we believe this would not change the results significantly. In order to optimize response time, we have implemented a preemptive scheduler. It accepts requests for execution and chooses the highest priority process requesting the CPU. If a request arrives from a higher priority process after execution has started, the scheduler preempts the executing process and starts the higher priority one. When a process finishes executing it resets its request, and the scheduler chooses another process. If data were shared directly, synchronization could cause deadlocks. This could happen, for example because of cyclic dependencies among locks. Monitor tasks avoid this problem because they eliminate the possibility of complex data dependencies.

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

Verification Results

Schedulability is one of the most important properties of a real-time system. It states that no process will miss its deadline. In this example the deadlines are the same as the periods (except for the weapon release subsystem). The following table summarizes the execution times computed by the algorithms. Processes are shown in decreasing order of priority. Deadlines are also shown so that schedulability can be easily checked. Minimum and maximum execution times are given for both preemptive and non-preemptive schedulers.

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

Figure 30. Aircraft controller schedulability results (times are in the form [min,max])

We can see from the table above that the process set is schedulable using preemptive scheduling. An analysis of a similar process set using RMS showed that only the first eight processes were guaranteed to meet their deadlines [60]. From our results we can also identify many important parameters of the system. For example, the response time is usually very low for best-case computations, but it is also good for the worst case. Most processes take less than half their required time to execute. This indicates that the system is still not close to saturation, although the total CPU utilization is high.

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

As another example of how the designer can use these results, we can analyze the response time for the display graphic subsystem. The periodicity of this subsystem is 80 ms and a shorter period might be desired to make motion look continuous. However, the response time of this process can be as high as 44 ms. Changing the period to 40 ms would most likely make it miss its deadline. The designer may choose to decrease it to 50 ms, but this is still close to the response time, and the increased load might make the system unschedulable. The model can be easily changed to check this hypothesis, but our analysis shows that it is unlikely that this modification will preserve schedulability.

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

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

In this example we have shown that Verus allows the analysis of complex realistic systems. We have been able to determine the schedulability of the system and understand its behavior in detail. We have also been able to produce information about its behavior that might have been difficult to obtain using other methods, such as the response time of the weapons subsystem.

6.3 A Robotics System

One application of real-time systems that is becoming increasingly common is in robotics. Guaranteeing that tasks are executed within their expected deadline is critical for the integrity of a robot and for the success of its mission. The computation of quantitative properties can assist in validating such systems. The example discussed in this section is derived from the one in [38]. It describes a real robot used in nuclear reactors to measure the shapes of pipes by moving around them with a distance sensor. The robot architecture has three subsystems, *motor*, *measurement* and *command*. The motor subsystem controls the robot movements and position. The function of the measurement subsystem is to activate and control the distance sensors. Finally, the command subsystem receives commands from the communication link and sends them to the appropriate tasks.

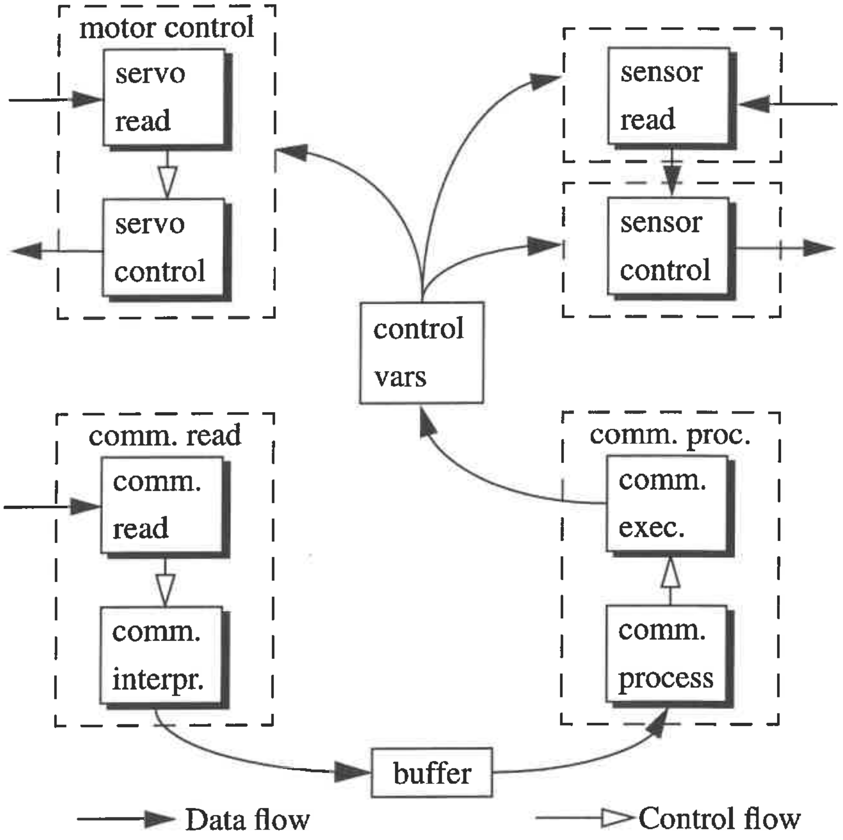


Figure 31. Robot architecture

Each subsystem consists of a set of tasks. The motor subsystem contains one task, *motor_control*. Its function is to receive data from sensors in the servo motors, and actuate them. The task consists of two subtasks, *servo_read* and *servo_control*. The first one is an interrupt routine that reads data directly from the physical devices. The second one processes these data and outputs control signals to the motors at a lower priority. The measurement subsystem has two tasks, *sensor_read* and *sensor_control*. The first task reads data from the distance sensors and preprocesses it. This information is then sent to *sensor_control*, which processes it further and outputs the results to a remote system to be analyzed. Finally, the command subsystem also has two tasks. The *command_read* task receives commands from the communication link and interprets them. It consists of two subtasks: an interrupt routine, followed by a second subtask that has a lower priority. The final task of this subsystem is *command_process*. Its first subtask receives the command interpreted by *command_read*, and the second one then executes the command. Control variables that are updated by this subtask are used to communicate commands to all other subsystems.

All tasks are periodic, and their timing requirements reflect the characteristics of the environment in which the robot works and the robot's expected response time. These requirements are summarized in the table below. Each task is presented as a sequence of components, each with a different execution time and priority. A component may correspond to a subtask, or subtasks may be split in more than one component due to synchronization. For example, the first components of both *motor_control* and *command_read* correspond to their interrupt routines and execute at high priorities. Synchronization accounts for the other components. For example, the last component of *command_process* updates control variables that will be used by other tasks. Interference from other tasks is avoided by accessing those variables at a high priority level. The other components have been created to reflect the synchronization pattern between processes sharing data (in this case between *sensor_read* and *sensor_control*), and between *command_read* and *command_process*. Priority inheritance protocols have been used to avoid priority inversion [66]. These protocols change the priority of the tasks at synchronization points, thus dividing the tasks into components.

Task	Period	Exec. Times			Deadline	Priorities		
		C_1	C_2	C_3		P_1	P_2	P_3
Motor control	40	1	5	-	40	10	7	-
Sensor read	100	10	5	5	100	4	8	4
Sensor control	50	8	12	-	50	5	8	-
Command read	200	10	20	3	200	9	2	3
Command process	400	2	12	10	400	3	1	6

Figure 32. Timing requirements for aircraft controller

The Analysis of the Robotics System

Computing response times for all processes generated the results in the table below. This table shows that the task set is schedulable. Moreover, the maximum execution times of many tasks are close to their deadlines. This indicates a high load on the system; it is unlikely that adding more tasks to the task set would produce a schedulable system. This information allows the designer to optimize or fine tune the system.

Task	Deadline	Exec. times	
		min	max
Motor control	40	6	16
Sensor read	100	45	95
Sensor control	50	20	49
Command read	200	181	190
Command process	400	219	223

Figure 33. Schedulability analysis for original system

Using the results computed by our algorithms, we have been able to suggest changes to the design and to analyze the effects of such changes. In the original design *sensor_read* generates data that are used by *sensor_control*. However, the two tasks execute independently of one another. In some cases *sensor_control* might execute even if data are not yet available. In this case, *sensor_control* uses data generated by the previous instantiation of *sensor_read*, which may be obsolete. We have changed the system to avoid this problem and have analyzed the resulting design. The modification consists of making the termination of *sensor_read* trigger the execution of *sensor_control*. Care must be taken, however, because the processes involved have different periods; *sensor_read* executes every 100 ms, while *sensor_control* executes every 50 ms. We change the system so that *sensor_read* signals the execution of *sensor_control* every 100 ms, but *sensor_control* also executes independently 50 ms after *sensor_read* runs. In this case one instantiation of *sensor_control* is synchronized with *sensor_read* while the other is independent. The schedulability analysis of the modified example is given in the table:

Task	**Deadline**	**Exec. times**	
		min	max
Motor control	40	6	16
Sensor read	100	20	36
Sensor control	50	21	121
Command read	200	91	91
Command process	400	96	296

Figure 34. Schedulability analysis for modified system

The new design is not schedulable, since *sensor_control* can take up to 121 ms to execute. We can use the same quantitative algorithms to find out more about the behavior of the system and to correct the problem. A more detailed analysis reveals that the two instantiations of *sensor_control* have very distinct behaviors. Whenever executing periodically (and independent of *sensor_read*), *sensor_control* takes between 21 and 121 ms to finish. However, whenever executing after *sensor_read*, it takes exactly 26 ms to execute in the modified model. This shows that the periodic execution of *sensor_control* is the bottleneck of the system. One solution to the problem is simply removing the periodic instantiation of *sensor_control*. This solution was easily implemented, and the schedulability analysis is presented in table:

Task	Deadline	Exec. times	
		min	max
Motor control	40	6	16
Sensor read	100	20	36
Sensor control	50	26	26
Command read	200	91	91
Command process	400	70	270

Figure 35. Schedulability analysis for final system

The system is again schedulable, but now *sensor_control* executes only once every 100 ms. Is this a satisfactory solution? Again, we can use the same algorithms to analyze the modified design. By computing the time between the end of the execution of *sensor_read* and the beginning of *sensor_control* we can verify if data produced by the first task is being consumed timely by the second one. In the modified model this time is between 1 and 7 ms, meaning that data produced by *sensor_read* are promptly consumed by *sensor_control*. Therefore we can conclude that in spite of changing the periodicity of *sensor_control* we are still maintaining predictability. The condition counting algorithms have also been useful in analyzing the performance of this model. We have been able to verify how the old periodicity of *sensor_control* relates to the new model. We can consider all execution paths from the time *sensor_read* starts until *sensor_control* finishes as the active period for the measurement subsystem. During such a period, how many times can the 50 ms time-out occur? In other words, how many times would *sensor_control* be activated using the original periodicity during an active period? The result is from 1 to 3 times. We conclude that the modified system satisfies the original timing constraints, even though it has a lighter load.

In this example we have been able to analyze the behavior of the robot from several perspectives. We have determined that it would meet its deadlines, but that it was inefficient. We have discussed how to optimize the design and then we have been able to analyze the performance of the modified design.

6.4 A Medical Monitoring System

This section presents a patient monitoring system derived from the one presented in [31]. It is a realistic example that models many features existing in actual systems. The example has been expanded to show how the algorithms described in this paper can be used to analyze models of industrial complexity. The resulting model for this example has more than 10^{13} states but its timing characteristics can be computed in a few seconds.

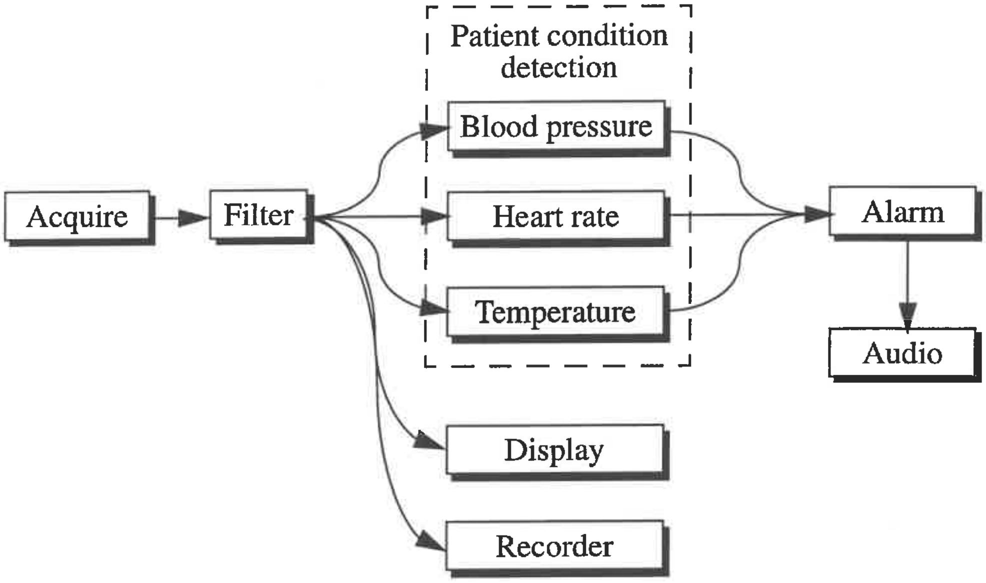


Figure 36. The patient monitoring system

The system consists of a set of processes and can be seen in figure 36. The *acquire* process is the only periodic process in the system, all others are aperiodic. *Acquire* executes every

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

Most processes in this system are aperiodic in nature. Because of this, methods such as the rate monotonic scheduling [50,59,68] cannot be directly used to analyze this process set. For example, the assignment of priorities to processes is more complex than in the periodic case which can use the RMS algorithms. In this example priorities have been assigned heuristically, and quantitative algorithms have been used to investigate the efficiency of the assignment. Initially, the priority order defined was, from the highest to the lowest priority process: *acquire*, *filter*, *blood_pressure*, *heart_rate*, *temperature*, *display*, *recorder*, *alarm*, and *audio*.

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

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

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

(2) Minimum and maximum times between the start and end of execution of each process. If this time is less than infinity, then the system is schedulable.

Figure 37. Response times for the original medical monitor

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

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

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

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

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

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

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

6.5 The PCI Local Bus

The PCI Local Bus [45,46] is a high performance bus architecture that can have a data width of 32 or 64 bits. It has been designed by Intel to be used in its latest family of processors. Intel's goal is to offer a fast bus design at low cost that will accommodate current as well as future systems. PCI buses can be found in systems based on Alpha, Pentium and Pentium Pro processors. The majority of Pentium based systems manufactured today employ the PCI bus.

A typical PCI system can be seen below. The most important subsystems connected to the bus are the processor, a video controller, a SCSI controller, and an ISA bridge controller, which connects the PCI bus to a slower ISA bus. Modems, floppy disk controllers and other low speed components are connected to the ISA bus. Main memory and the secondary cache are connected directly to the processor using a PCI-memory-processor bridge. Other components can be added to the system. Usually expansion slots are provided for this purpose.

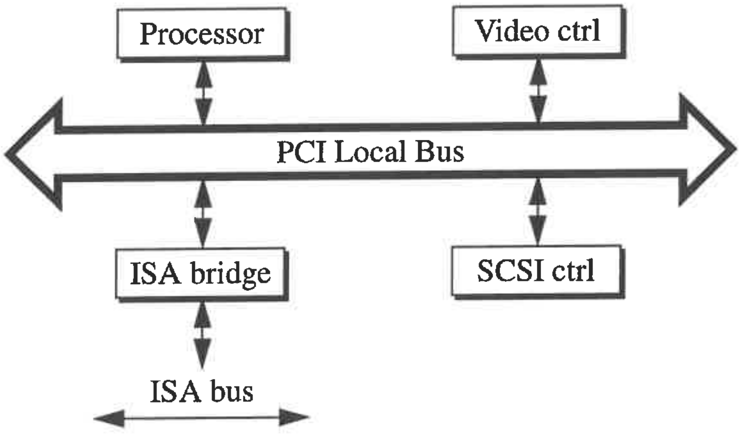


Figure 38. The PCI local bus

Each of the subsystems shown above is allowed to request access to the bus and issue transactions. Slave subsystems are also supported; such subsystems respond to transactions, but do not issue them. A simplified PCI transaction is shown in the figure below. The request for a transaction starts when a subsystem asserts its request line `REQ`. It then waits until being granted the bus by the arbitration subsystem, which is indicated by the assertion of the `GNT` line. This phase is known as the *arbitration phase*. The next phase is the *bus acquisition phase*. The bus might not be idle when the new master is determined because the previous transaction may still be transferring data. Another transaction cannot be issued before all data has been transferred. The bus is idle whenever both signals `FRAME` and `IRDY` are deasserted in the same cycle, giving access of the bus to the new master. At this point the master asserts the `FRAME` signal, indicating the end of the bus acquisition phase and the beginning of a transaction. It also has to assert the signal `IRDY`, meaning that it is ready to send (or receive) data. The bus master has to wait for the target subsystem to respond by asserting its `TRDY` signal. This indicates that the target is ready to supply (or receive) data. The time interval between the start of a transaction and the assertion of the `TRDY` signal is called the *target response phase*. Data transfer starts when both `IRDY` and `TRDY` are asserted. One clock cycle before the end of the data transfer phase the `FRAME` signal is deasserted. At the next cycle both `IRDY` and `TRDY` are deasserted, and the bus becomes idle. In addition, transactions can be cancelled in various situations. This feature of the protocol is discussed in more detail later.

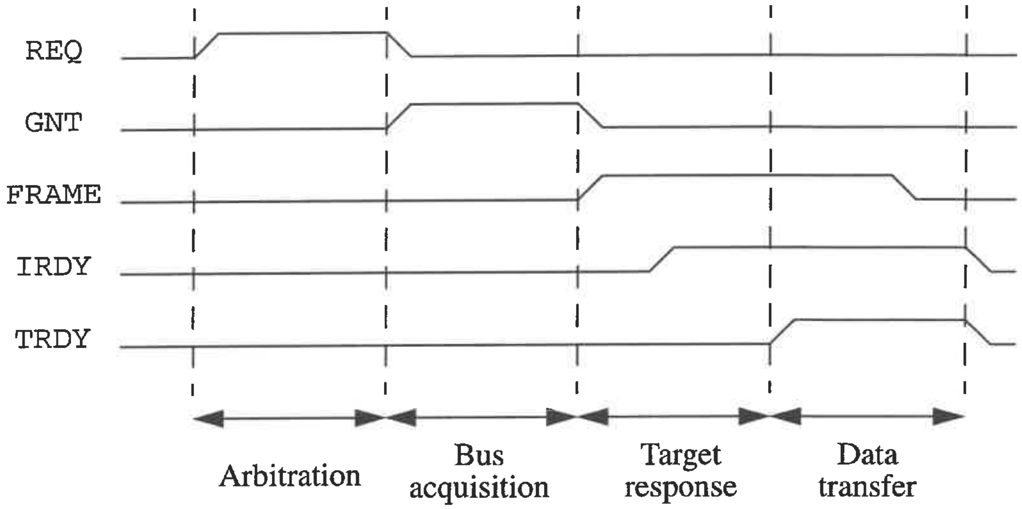


Figure 39. A transaction in the PCI Bus

Arbitration in the PCI bus is implemented by a *two phase arbiter* as seen in the next figure. Each arbiter bank chooses among its incoming requests, and sends its decision to the following bank. The output of `bank2` will be the new bus master. The decision is based on the `policy` signal, which can be set to *fixed priority* or *round-robin*. If all policies are set to the same value, the global arbitration policy will be either fixed priority or round-robin. However, mixed arbitration policies are possible by combining different policies in the banks. Our model for the PCI bus follows the description above. Arbitration policies can be set to any possible combination, allowing mixed arbitration policies. However, in our model we must make some restrictions to the protocol described. For example, we must restrict the amount of data being transferred in one transaction. If this restriction is not implemented, no bounds on response time can be determined. In our model a single transaction can transfer between 1 and 16 cache lines of data. Our analysis will show how the information generated by this model can be used to determine the response time for models without this restriction. A similar approach has to be taken with the possibility of cancelling an ongoing transaction. Again, in order to prevent starvation, we must bound the number of times a transaction may be cancelled. Our final model for the PCI bus has 10^7 reachable states out of a state space of 10^{18} states. The transition relation uses less than 10,000 BDD nodes, and the verification was completed in minutes.

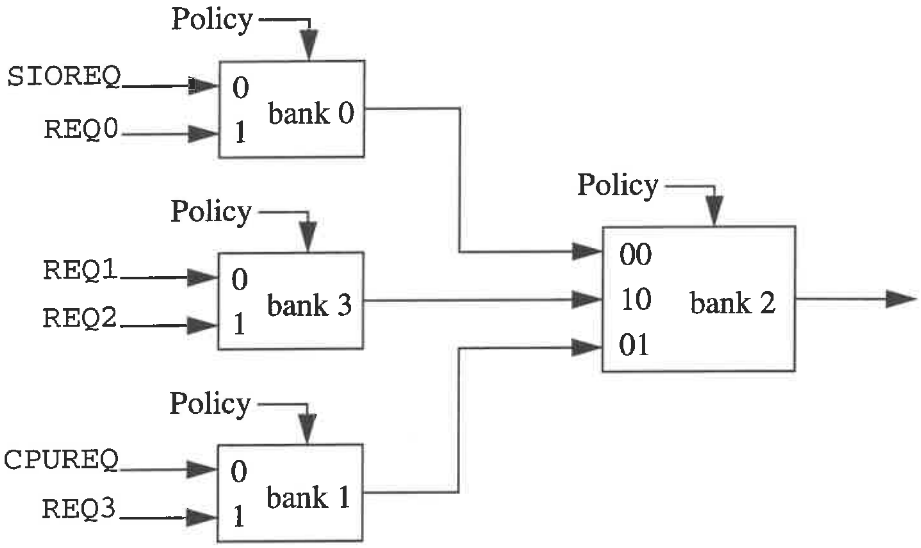


Figure 40. The PCI arbiter

Verification and Performance Analysis of the PCI Bus

Our analysis concentrates on the verification of issues such as transaction termination and arbitration fairness as well as on transaction performance. Being able to estimate the response time of a transaction is extremely important in any bus design, especially in one which has high performance a primary goal. The bus data transfer rate and the overhead imposed by arbitration and communication protocols are examples of parameters involved in such an analysis. If those parameters cannot be determined, it will not be possible to design an optimized system that fully utilizes the available resources.

Moreover, the PCI bus is a good alternative for critical applications in which a bounded response time is vital. However, if the worst case response time of a transaction in the PCI bus hasn't been specified, such applications will most likely be implemented using other bus architectures. By bounding the worst time response of a transaction we hope to help application designers to evaluate the use of the PCI bus more accurately.

The correctness of the PCI bus protocol can be verified using the CTL model checker. For example, absence of starvation for bus access and transaction termination can be verified by the following formulas:

$$\mathbf{AG}\,(\mathtt{REQ} \rightarrow \mathbf{AF}\,\mathtt{GNT})$$

$$\mathbf{AG}\,(\mathtt{start\_transaction} \rightarrow \mathbf{AF}\,\mathtt{end\_transaction})$$

The properties above show that the response time of PCI transactions is bounded, but they give no indication of their performance. We will use the quantitative algorithms described to determine the response time for transactions. The results of our quantitative analysis also determine the correctness of the algorithm, for example, a transaction always finishes if its maximum response time is less than infinity.

In our performance analysis we will follow the structure of the protocol by computing the response time for each phase of the transaction separately. In this way we can have a better understanding of the behavior of the protocol. By computing the latency of each phase we are able to assert the efficiency of each step in the protocol and obtain the global behavior by adding individual figures. Results will be grouped into two categories, *total bus acquisition latency* and *total transaction latency.* The first category corresponds to the total time between a request being made on the bus and the subsystem actually being able to use the bus. The second category represents the total usage of the bus, that is, the time between asserting the `FRAME` signal until the end of data transfer. The table below shows the response times when the arbitration policy is set to round-robin in all banks and transaction cancelling is not allowed. Notice that in all cases discussed in this paper the latency for the data transfer phase varies between 1 and 16 clock cycles, there is no overhead associated with it. For that reason, this column will not be shown in the tables.

Bus Master	Arbitration	Bus Acquis.	Total bus acquis.	Target response	Total trans.
ISA	[1,95]	[1,18]	[2,113]	[1,2]	[2,18]
SCSI	[1,95]	[1,18]	[2,113]	[1,2]	[2,18]
Video	[1,38]	[1,18]	[2,56]	[1,2]	[2,18]
Processor	[1,38]	[1,18]	[2,56]	[1,2]	[2,18]

Figure 41. Response times for global round-robin policy

From the table above we can see two interesting properties of the system. The total transaction latency is at most 18 clock cycles, and in this case 16 clock cycles of data are transmitted. This means that once a master is able to use the bus, it can send data very efficiently. Another characteristic of the protocol is reflected on the bus acquisition times. The maximum of 18 cycles corresponds to one transaction. After being granted the bus the new master may have to wait for at most one more transaction to complete. This shows that once the bus is granted to a master, it will not be granted to another before the first one issues its transaction. Therefore no starvation can occur after a master is granted the bus. This property can be verified by the following CTL formula:

$$\mathbf{AG}\ (\mathtt{GNT} \rightarrow \mathbf{A}[\mathtt{GNT}\ \mathbf{U}\ \mathtt{FRAME}])$$

A more intriguing result can be seen in the arbitration latency results. The first two subsystems can take almost twice as long to access the bus as the others. In a round-robin environment, all subsystems should be granted equal usage of the resource, but this is not true in our example. By analyzing the execution traces produced by our tools we are able to determine the reason for the unfair access to the bus. The problem arises from the connection of the request lines to the arbiter as seen below. The ISA bridge and the SCSI controller are connected together to bank0, while the video and the processor subsystems are alone in their banks. If bus traffic is high, the ISA bridge and the SCSI subsystems may have to wait for the one another before their request reaches bank2. Subsequently they may have to wait for subsystems connected to the other banks to execute before being granted the bus. In other words, they compete in both levels of arbitration, while the other subsystems only compete in the last level. This causes the worst time latency to be approximately twice as long for these subsystems. We can conclude from these results that two level arbitration *may* have a different behavior than an equivalent one level arbiter. In this case the problem is caused by an asymmetric connection of request lines.

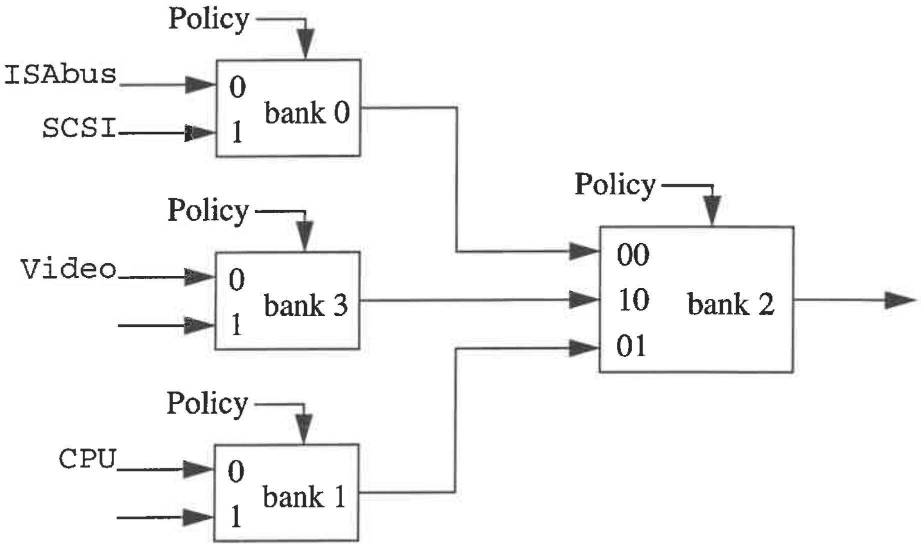


Figure 42. Connections of request lines to the arbiter

We can also use these results to analyze the overhead imposed by the communication protocol on the transaction time. We have already seen that after asserting the FRAME signal there is an overhead of 2 clock cycles. This overhead is independent of the transfer size. If a transaction is allowed to transfer more than 16 cache lines of data at once, the total utilization of the bus will increase. The designers of the bus can use this information to determine which is the best transfer size for a given system. The following two formulas have been used to verify the above statements:

$$\mathbf{AG}\ (\mathtt{FRAME} \rightarrow \mathbf{AF}_{\leq 2}\ (\mathtt{state} = \mathtt{DATA\_TRANSFER}))$$

$$\mathbf{AG}\ ((\mathtt{state} = \mathtt{DATA\_TRANSFER}) \rightarrow$$
$$\mathbf{A}[\mathtt{state} = \mathtt{DATA\_TRANSFER}\ \mathbf{U}\ \mathtt{end\_transaction}])$$

The first formula states that at most two cycles after the transaction starts, it will enter the data transfer phase. The second formula states that once a transaction is in the data transfer phase, it will continue in this phase until its end.

The overhead associated with arbitration can be computed in a similar way. It is more complex, however, because the arbitration latency depends not only on the transaction time, but also on the number of active request lines. We use the condition counting algorithms to uncover more details about this problem. We compute the number of transactions issued on the bus between the time a master requests access and the time it is granted the bus. Up to 5 transactions can be issued during this period for the ISA bridge and the SCSI subsystems, and up to 2 transactions can be issued for the video and processor subsystems. Total transaction time for each of these intermediate transactions is 18 clock cycles. By comparing the total effective data transfer time with the maximum arbitration time, we can see that each intermediate transaction has an arbitration time of one clock cycle. These results are also valid for the video and processor subsystems. We can conclude that the arbitration latency can be computed by the formula:

$$Arbitration\_Latency = n * (Transaction\_Latency + 1),$$

where n is the maximum number of intermediate transactions that can be issued between a request and the corresponding grant (computed with the condition counting algorithms). This formula does not depend on maximum data transfer size.

The above results assume a global round-robin policy. The behavior of the system under a fixed priority arbitration policy has also been studied. The ISA bridge is the highest priority subsystem on the bus. Its response time is much lower in the fixed priority configuration than in the round-robin one. However, all other subsystems may starve, since the ISA bridge can continuously issue transactions. Notice that only the arbitration time, but not the transaction time, is affected by the arbitration policy. These response times can be used by the designer to check if the performance of the PCI bus is adequate for a critical application. Other combinations of arbitration policies are possible, but are not presented here for the sake of brevity.

Bus Master	Arbitration	Bus Acquis.	Total bus acquis.	Target response	Total trans.
ISA	[1,19]	[1,18]	[2,37]	[1,2]	[2,18]
SCSI	[1,∞]	[1,18]	[2,∞]	[1,2]	[2,18]
Video	[1,∞]	[1,18]	[2,∞]	[1,2]	[2,18]
Processor	[1,∞]	[1,18]	[2,∞]	[1,2]	[2,18]

Figure 43. Response times for global fixed priority policy

The model described above allows a detailed analysis of the behavior of the PCI bus protocol. Some features of the actual bus, such as parity or data width, have been abstracted from our model, since they do not affect the timing of transactions. However, there are other features that do affect timing such as the possibility of a transaction being cancelled. Errors on the bus may occur, the target may be slow, or unable to produce the data. For example, a transaction requesting data from the ISA bus will most likely experience a long delay, simply because of the relative speeds of the ISA and PCI buses. In the model described above this feature has been abstracted out by the assumption that the target of a transaction responds immediately. A more realistic model that allows transactions to be cancelled has also been implemented.

In order to account for long delay responses and aborted transactions we introduce the concept of transaction cancellation in our model. Transactions may be cancelled any time they are in progress. Transaction cancellations model the fact that in the actual PCI bus whenever a target is unable to answer for a long time, it aborts the transaction, which is reissued later. We model this situation by cancelling the transaction and restarting it immediately by issuing another request. However, reissuing the transaction immediately would not correctly model the response time of a very slow target. To accommodate this situation, in our model a cancelled transaction is restarted as many times as necessary to accommodate the target response time. Using the algorithms described we compute the overhead caused by cancelling and restarting a transaction, and use this result to determine the number of retries for the response delay of a given target.

Moreover, unlimited cancellations may cause starvation. Therefore, in order to compute the worst time response, we must limit the number of cancellations allowed. A cancellation brings the bus to the idle state, as can be verified by the following CTL formula:

$$\mathbf{AG}\ (\texttt{ABORT} \rightarrow \mathbf{AX}\ \texttt{BUS\_IDLE})$$

As a consequence, consecutive cancellations have the same behavior, because a cancellation brings the system into the same state as before the transaction. Therefore, the total overhead caused by *n* cancellations is *n* times the overhead of a single cancellation. Therefore, it suffices to consider the situation in which at most one cancellation occurs. The results for a global round-robin arbitration policy in the presence of at most one transaction cancellation are presented below.

Bus Master	**Arbitration**	**Bus Acquis.**	**Total bus acquis.**	**Target response**	**Total trans.**
ISA	[1,95]	[1,18]	[2,113]	[1,6]	[2,132]
SCSI	[1,95]	[1,18]	[2,113]	[1,6]	[2,132]
Video	[1,38]	[1,18]	[2,56]	[1,6]	[2,75]
Processor	[1,38]	[1,18]	[2,56]	[1,6]	[2,75]

Figure 44. Response times for global round-robin policy, maximum one cancel

In this table we can see that arbitration latency is not affected by transaction cancellations. The reason is that whenever a transaction is cancelled the current bus master releases the bus and becomes last in the round-robin queue. On the other hand, total transaction latency increases significantly. The execution trace of the transaction with the worst latency shows the following sequence of events (for the ISA bridge subsystem):

- A transaction starts but is cancelled just before completion, after 17 clock cycles.
- Another request is made to complete it in the next cycle (one extra clock cycle).
- An arbitration sequence of 79 cycles follows.

- A bus acquisition phase starts and takes 17 clock cycles.
- The transaction starts again, completing after 18 cycles.

The arbitration sequence appearing in item 3 is the same as in the worst case, except that the request is made when *the bus is already idle* because of the cancellation. The difference of 16 clock cycles corresponds to one maximum data transfer phase done by another bus master, as shown by the counterexample for the worst case arbitration latency (not presented for brevity). The total delay caused by the first three items is the equivalent of a worst case arbitration latency plus two clock cycles, caused by the cancellation. A bus acquisition phase and a transaction latency phase, in which no cancellation occurs, account for the last 35 cycles. We can see then that the overhead imposed by a transaction cancellation consists of a worst case arbitration latency, a maximum bus acquisition phase, a maximum transaction latency (without cancellations) and one extra clock cycle. Again, this formula applies for the video and processor subsystems. These results may be used to estimate the performance of an implementation of the PCI in the presence of transaction aborts. The formula derived gives the overhead for one transaction cancellation, and can be extended to many cancellations as well. In this manner, the worst response time in various configurations of the system can be computed.

To summarize the results of our analysis, we have been able to:

- Model the PCI Local bus protocol and verify its correctness. In the round-robin case no starvation of subsystems occur, and transactions always finish, even in the presence of limited cancellations.
- Determine the minimum and maximum latencies for each phase of the protocol, and show which phases are affected by changes in the parameters (such as arbitration policy and presence of cancellations).
- Compute response times independent of specific values for the data transfer phase.
- Determine response time in the presence of limited transaction aborts using the condition counting algorithms described.

These results allow the designers of the protocol to understand its actual behavior and how this behavior changes when parameters of the system are modified. We believe that this is valuable information when verifying and optimizing a new hardware system. This example shows that our method can be used to analyze the performance of modern hardware designs that have very complex behavior.

6.6 A Distributed Real-Time System

In this section we analyze a distributed real-time system. 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 [4]. 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 100 ms between the generation of multimedia data and its processing by the control processor.

The traffic in the network has been modeled as proposed in [69]. 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 [69]. 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).

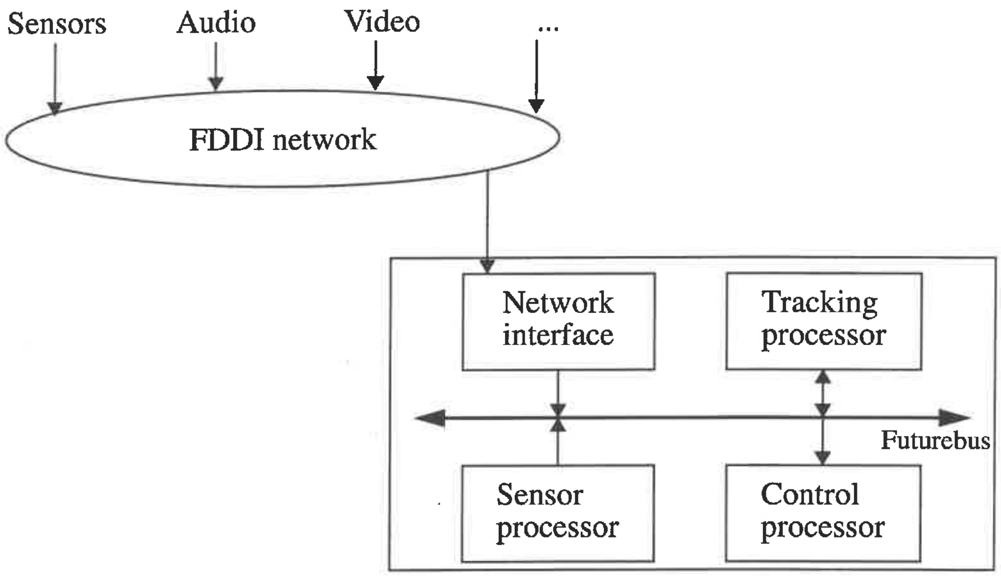


Figure 45. Distributed System Architecture

In the multiprocessor, four active processors are connected through a Futurebus+ [44]. 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 7 ms at each time. A *sensor* processor reads data from sensors every 40 ms. It buffers the data and sends it once every four readings to the tracking processor. The *tracking* processor processes the data and sends it to the *control* processor. Both sensor and tracking data use the bus for 3 ms each. The deadline for sensor data to be processed is 785 ms. 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 46. Priority scheduling is also used in the control processor, with priorities 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.

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

Figure 46. Timing requirements for tasks in the control processor (times in ms)

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 have 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 100 ms 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], that is, the video 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 in the interval [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 the data by τ_3 in the control processor. This deadline is 785 ms. 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 on 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. A counterexample for the longest response time confirmed our speculations.

In this system both sensor and tracking processors access the bus periodically, sending data every 160 ms. In the counterexample, however, data required two periods of 160 ms 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 160 ms.

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. A very serious problem with real-time systems is priority-inversion. It occurs when high-priority tasks are blocked by low priority ones. 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 want to determine the existence of priority inversion between the time the sensor produces data until the time the tracking processor processes it. 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, even though it 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 less critical ones. It would be easy to adapt this analysis to different data patterns, and optimize the response time for any other application. 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.

Chapter 7 Conclusions

This work presents a new method for specifying and verifying real-time systems. The system being verified is specified in the Verus language and then compiled into a state-transition graph. Algorithms derived from symbolic model checking are used to compute quantitative timing information about the model. There are several advantages to the new approach. For example, the Verus language has been especially designed to allow a straightforward description of the temporal characteristics of programs, simplifying the expression of real-time systems in general. Also, the quantitative information produced by the verification algorithms not only allows the designer to check for its temporal and functional correctness, but also provides insight into how well the system works, or how seriously it fails. The algorithms presented generate more detailed information about the behavior of a system than previous methods and can be used in several different ways to analyze a real-time system.

The method has been applied to several actual systems, and, in each case, we have been able to produce useful information that can enable the designers to understand the behavior better and to improve the system. These examples demonstrate the versatility and power of the method, and can be used to emphasize important aspects of the Verus approach:

- The priority inversion example has demonstrated that the synchronous composition model used in Verus does not restrict the applicability or expressive power of the tool. It may be argued that some applications are inherently asynchronous, and therefore not well suited for the proposed method. In this example, however, it is shown how stuttering can be used to introduce asynchronism into the model even without sacrificing the synchronous composition model.
- The aircraft controller example illustrates the importance of the efficiency of the symbolic algorithms used in Verus. As discussed, one of the most restrictive factors in formal verification is the number of concurrent processes in the system, due to the complexity of the parallel composition algorithm. In this example we have shown that systems with a large number of processes can be efficiently verified by Verus. The aircraft controller has 15 concurrent processes, but quantitative information about the system can be computed in seconds using a 486 computer.
- The versatility of the method can be seen in the robotics and medical monitoring examples. Because of their design, neither example can be directly analyzed by the rate monotonic algorithms. In fact, a complex extension to RMS has been developed to handle systems such as the robotics controller presented [38]. The quantitative analysis performed cannot be directly done using standard model checking. In Verus, however, both systems have been implemented and verified in a straightforward way. This shows how Verus can easily accommodate several different types of systems and properties.

  Moreover, in both cases the analysis performed has uncovered inefficiencies in the design and suggested optimizations. Results such as these can be very difficult to obtain using other tools. After modifying the design, the same algorithms have been used to show that the performance of the system has improved. These examples show the versatility of the specification language, and the usefulness of the results produced by the verification algorithms.
- The PCI local bus analysis has shown that not only real-time systems can benefit from the new method. This example shows how timing properties of non real-time systems can be analyzed, and how those results can be used to help understand the behavior of systems that are not usually associated with real-time.

Another important issue has been discussed in the analysis of the PCI bus. Even though limited to finite-state systems, Verus can produce results that can be extrapolated to larger classes of systems. In this example restrictions to the model have been implemented to ensure that the model is finite. Later the results generated by the analysis have been extrapolated to different configurations of the PCI that are not limited by those restrictions. This shows that the results produced by the algorithms can be even more general and useful than a superficial verification might lead to believe. Sometimes a careful analysis can produce information that might have seen impossible to obtain due to the characteristics of the method, as has been the case in this example.

- The distributed real-time system analyzed shows a combination of all these features into an analysis that would be very difficult to perform using other tools. It is a large complex system that exceeds the complexity of systems that can be verified using continuous time tools. It cannot be analyzed directly by RMS, because RMS does not allow a complete analysis of distributed systems. Moreover the quantitative information produced would be very difficult if not impossible to obtain using techniques such as model checking.

  In Verus, however, we have been able to model the system naturally. The quantitative algorithms have been applied directly to the model. In this case they also uncovered inefficiencies and suggested optimizations to the design. Finally, the efficiency of the algorithms allowed results to be produced in minutes in all cases.

These examples show that Verus is an efficient and useful tool for analyzing real-time systems. It can perform analyses that are impossible or extremely difficult to do using previous methods. The technique can be efficiently applied to complex real-time and non real-time systems and can assist in determining their correctness and in understanding their behavior. It can ultimately contribute to the implementation of more efficient and reliable systems.

Future Work

This work can be extended in many directions. For example, the examples discussed previously show that many different types of analyses can be performed. The verification of more examples is the key to developing new ways to produce and analyze information about the behavior of real-time and non real-time systems. Particularly, analyzing systems in different areas might prove useful in finding new ways to look at the information produced. Examples of areas in which Verus may be useful are flexible manufacturing, other types of industrial controllers, circuit design and software systems.

An important extension of the method is the implementation of non-unit transitions in the model. Allowing transitions that take longer than one time unit to occur can help increase the efficiency of verification. Less states in the model are generated because long transitions are not expanded into a sequence of unit transitions. A model that allows transitions to take time t to occur, where t is within a defined time range has been developed [12]. Research is needed, however, to implement an efficient parallel composition algorithm for this model.

An important aspect of a model with non-unit transitions is the fact that with each transition is associated an entity, in this case a time range. But other models can be derived from this one by associating transitions with other entities. One useful example is associating probabilities with transitions. This would allow the computation of the probability of reaching a certain state, or the probability of reaching a steady-state. A probabilistic model checker has many important applications, but much research is still needed in this area.

Another potentially important extension to Verus comes from noticing that two different types of variables exist in the model. Variables that model the control or data in the system, and variables that model time. The behavior of these two sets of variables can be very different. Time variables are manipulated in a restricted way, usually by resetting or incrementing their value. There may be more efficient representations for those variables that optimize the representation of these operations. One promising candidate representation is

BMDs [7]. BMDs are very efficient for representing arithmetic operations, and can perhaps increase the efficiency of the verification on Verus.

Several other extensions can also be of benefit, such as extensions to the Verus language, or the exploration of symmetry in the models. We believe that this work can be only the beginning of a new research area on formal verification of real-time systems using quantitative algorithms. We hope to be able to continue this work and explore some of these research directions, generating in the end an even more efficient and useful tool.

Chapter 8 References

[1] R. Alur, C. Courcourbetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Symposium on Logics in Computer Science*, pp. 414-425, 1990.

[2] R. Alur and D. Dill. Automata for modeling real-time systems. In *Lecture Notes in Computer Science, 17th ICALP.* Springer-Verlag, 1990.

[3] R. Alur and T. Henzinger. Logics and models of real-time: a survey. In: *Lecture Notes in Computer Science, Real Time: Theory in Practice*. Springer-Verlag, 1992.

[4] ANSI Std. *FDDI Token Ring Media Access Control*, s3t95/83-16 edition, 1986.

[5] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. In: *Science of Computer Programming*, vol. 19, 1992.

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

[7] R. E. Bryant and Y. A. Chen. Verification of arithmetic functions with binary moment diagrams. Technical Report CMU-CS-94-160, Carnegie Mellon University, 1994.

[8] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *VLSI 91*, Edinburgh, Scotland, 1990.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference,*, June 1990.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5^{th} Symposium on Logics in Computer Science*, 1990.

[11] S. V. Campos. The priority inversion problem and real-time symbolic model checking. Technical Report CMU-CS-93-125, Carnegie Mellon University, 1993.

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

[13] 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.

[14] 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.

[15] S. V. Campos, E. M. Clarke, W. Marrero and M. Minea. Verifying the performance of the PCI local bus using symbolic techniques. In: *ICCD*, 1995.

[16] S. V. Campos, E. M. Clarke, W. Marrero and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. In: *Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.

[17] Z. Chaochen, C. Hoare and A. Ravn. A calculus of durations. *Information Processing Letters*, 40, 5, 1991.

[18] D. Clarke, H. Ben-Abdallah, I. Lee, H. Xie and O. Sokolsky. XVERSA: an integrated graphical and textual toolset for the specification and analysis of resource-bound real-time systems. In: *Proceedings of the 8th Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 1102. Springer-Verlag, 1996.

[19] 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*. LNCS 131, Springer-Verlag, 1981.

[20] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244-263, 1986.

[21] E. Clarke, O. Grumberg, K. Hamaguchi. *Another look at LTL model checking*. Technical Report CMU-CS-94-114, Carnegie Mellon University, School of Computer Science, 1994.

[22] E. Clarke, O. Grumberg, and H. Hamaguchi. Another look at LTL model checking. In: *Proceedings of the Sixth Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 818, pages 415-427. Springer-Verlag, 1994.

[23] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus$^+$ cache coherence protocol. In *Proceedings of the 11th CHDL*, 1993.

[24] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, 1992.

[25] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In: *REX'93 School/Workshop: A Decade of Concurrency*, Nordwijkerhout, The Netherlands, June1993.

[26] O. Coudert, C. Berthet, and J. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems, International Workshop*, Grenoble,France, *Lecture Notes in Computer Science*, vol. 407, Springer-Verlag, June, 1989.

[27] P. Clements, C. Heitmeyer, G. Labaw, and A. Rose. MT: a toolset for specifying and analyzing real-time systems. In *IEEE Real-Time Systems Symposium*, 1993.

[28] R. Cleaveland, P. Lewis, S. Smolka and O. Sokolsky. The concurrency factory: a development environment for concurrent systems. In: *Proceedings of the 8th Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 1102. Springer-Verlag, 1996.

[29] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In: *Proceedings of the 8th Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 1102. Springer-Verlag, 1996.

[30] D. Dill. The Murφ verification system. In: *Proceedings of the 8th Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 1102. Springer-Verlag, 1996.

[31] P. Drongowski. Software architecture in real-time systems. In *IEEE Workshop on Real-Time Applications*, 1993.

[32] E.A. Emerson and Chin Laung Lei. Modalities for Model Checking: Branching Time Strikes Back. In Twelfth Symposium on Principles of Programming Languages, January, 1985.

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

[34] J. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu and M. Sighireanu. CADP: a protocol validation and verification toolbox. In: *Proceedings of the 8th Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 1102. Springer-Verlag, 1996.

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

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

[37] P. Le Guernic, M. Le Borgne, T. Gautier and C. Le Maire. Programming real time applications with Signal. Technical Report 1446, INRIA, Rennes, 1991.

[38] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering*, 20(1), 1994.

[39] R. Hardin, Z. Har'El and R. Kurshan. COSPAN. In: *Proceedings of the 8th Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 1102. Springer-Verlag, 1996.

[40] C. Heitmeyer and N. Lynch. The generalized railroad crossing: a case study in formal verification of real-time systems. *IEEE Real-Time Systems Symposium*, 1994.

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

[42] T. Henzinger, P. Ho, and H. Wong-Toi. HyTech: the next generation. In *IEEE Real-Time Systems Symposium*, 1995.

[43] G. Holzmann and D. Peled. The state of Spin. In: *Proceedings of the 8th Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 1102. Springer-Verlag, 1996.

[44] IEEE Standard Board and American National Standards Institute. *Standard Backplane Bus Specification for Multiprocessor Architectures: Futurebus+*, ansi/ieee std 896.1 edition, 1990.

[45] Intel Corporation. *82378 System I/O (SIO) — PCI Local Bus*, 1993.

[46] Intel Corporation. *PCI Local Bus Specification*, 1993.

[47] F. Jahanian and D. Stuart. A method for verifying properties of modechart specifications. In: *IEEE Real-Time Systems Symposium*, 1988.

[48] M. Joseph and P. Pandya, Finding response times in a real-time system. In: *The Computer Journal*, 29(5), 390-394, 1986.

[49] J. P. Lehoczky. Real-time resource management techniques. *Encyclopedia of Software Engineering*. John-Wiley & Sons, 1994.

[50] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *IEEE Real-Time Systems Symposium*, 1990.

[51] J. P. Lehoczky and L. Sha. Performance of real-time bus scheduling algorithms. *ACM Performance Evaluation Review*, 14, 1986.

[52] J. P. Lehoczky, L. Sha and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, 1989.

[53] 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.

[54] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2, 1982.

[55] H. Lewis. A logic of concrete time intervals. In *Proceedings of the 5th Symposium on Logic in Computer Science*, 1990.

[56] 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.

[57] O. Lichtenstein, A. Pnueli and L. Zuck. The Glory of the Past. In *Proc. Conf. Logics of Programs*, Lecture Notes in Computer Science 193, Springer-Verlag, 1985.

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

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

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

[61] 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, Springer-Verlag, 1989.

[62] K. L. McMillan. *Symbolic model checking - an approach to the state explosion problem.* Ph.D. thesis, SCS, Carnegie Mellon University, 1992.

[63] 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.

[64] D. T. Peng and K. G. Shin. A new performance measure for scheduling independent real-time tasks. Technical Report, Real-Time Computing Laboratory, University of Michigan, 1989.

[65] A. Pnueli. The Temporal Semantics of Concurrent Programs. In *Proceedings of the eighteenth conference on Foundation of Computer Science*, 1977.

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

[67] Y. Ramakrishna, P. Melliar-Smith, L. Moser, L. Dillon, and G. Kutty. Really visual temporal reasoning. In *IEEE Real-Time Systems Symposium*, 1993.

[68] 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.

[69] 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.

[70] B. Sprunt. *Aperiodic task scheduling for real-time systems.* Ph. D. dissertation, Department of Electrical and Computer engineering, Carnegie Mellon University, 1990.

[71] J. K. Strosnider. *Highly responsive real-time token rings.* Ph. D. dissertation, Department of Electrical and Computer engineering, Carnegie Mellon University, 1988.

[72] G. Thuau and D. Pilaud. Using the declarative language Lustre for circuit verification. In: *Workshops in Computing — Designing Correct Circuits*, Springer-Verlag, 1990.

[73] H. Touati, H. Savoj, B. Lin, R. Brayton and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *IEEE International Conference on Computer-Aided Design*, 1990.

[74] 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.

[75] F. Wang. Timing behavior analysis for real-time systems. In *Proceedings of the Tenth Symposium on Logic in Coomputer Science*, 1995.

[76] G. Winskel. *The Formal Semantics of Programming Languages, an Introduction.* The MIT Press, 1994, pp. 135-139.

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

[78] X. Zhao. Personal communication.

Index