Specification, Modelling, Verification and Runtime Analysis of Real Time Systems

Georgios Logothetis

Dissertation Thesis

University of Karlsruhe

Karlsruhe, July 17th, 2003

Reviewer:

Prof. Dr.-Ing. Detlef Schmid

Prof. Dr.rer.nat Peter Schmitt

to Claudia Alexia

This work is the result of my activities as researcher at the Institute for Computer Design and Fault Tolerance, University of Karlsruhe, Germany.

I am most grateful to the Director of the Institute, Prof. Dr.-Ing. Detlef Schmid, for his encouraging scientific and personal support during the past years. His personality and strong presence was always an important source of inspiration for me, giving me the necessary strength to continue and realize this work.

My special thanks go also to Prof. Dr.-Ing. Winfried Görke for many helpful and motivating discussions.

I am very grateful to Prof. Dr.rer.nat. Peter Schmitt for accepting the review of the manuscript.

Especially, I would like to thank Prof. Dr.rer.nat. Klaus Schneider, who supported me with countless scientific discussions and consistent hard work. His persistence and competence was always of great help to me.

Also I would like to thank Prof. Dr.-Ing. Hans Eveking for introducing me to the world of formal verification, evoking my interest to continue in this field.

Furthermore, I am very thankful to all my colleagues and friends at the Institute for Computer Design and Fault Tolerance for the excellent working atmosphere and cooperation.

Above all, I am most grateful to my wife, Claudia Alexia, for her love, endless patience and support during all the years.

Finally, many special thanks to my parents, my relatives and friends, in particular Evans Protopapas, Fanis Zampetakis, Stefanos Papadatos, Alexander and Claudia Aring as well as the family Winzinger.

Contents

List of Figures

Summary

Nowadays the use of computer systems has become a part of our everyday life. As these systems are increasingly involved in safety-critical applications, it is a challenge for computer scientists to develop design methodologies, theories and techniques, which are able to guarantee their correct design and operation.

For this purpose, formal methods have been established over the past decades as a very reliable solution. The idea is hereby to design, develop and realize a system within a formal framework, which allows the application of mathematical methods in order to obtain a formal verification of the system, i.e., a mathematical proof for its correctness with respect to given formal specifications. One of the most successful and reliable approaches for the verification of finite state systems has been model-checking: properties are given as formulae of propositional temporal logics and automatically verified by a graph-theoretic analysis of a formal model, which is usually a state transition system (Kripke structure). In contrast to simulation and test, formal methods deliver a mathematical proof for the correct function of a system and thus are able to determine errors in early design stages.

However, the correctness of a real-time system does not only depend on its functional behavior, but also on specific real-time constraints.

Within the scope of this work, we have focused on the use of formal methods in order to design, develop and realize correct safety-critical real-time systems. In particular, we have developed Equinox, a formal framework for the specification, modelling, verification and runtime analysis of real-time systems. The framework includes the real-time multi-tool JERRY and a compiler for the extended synchronous language Quartz. The major components of Equinox are described below.

1. *Real-time formal model: timed Kripke structures*

   Despite increasing computer performance and improved algorithms, the so-called state-explosion remains the most important problem in model-checking verification: the number of states in a system can increase exponentially with the size of its description. In real-time systems, the state explosion is even more increased by the additional consideration of real-time constraints. A very successful technique to obtain reduction of state-explosion is the abstraction of properties, that are irrelevant for the system's verification.

In order to allow abstractions in real-time systems, we have introduced timed Kripke structures as formal model. Besides the relative order of the events, a timed Kripke structure is able to describe their appearance in time. The time is hereby modeled discrete, i.e. by positive natural numbers. It also allows the direct use of established symbolic verification methods, as well as the use of abstractions without the loss of quantitative properties. Furthermore, the modeling of non-interruptible processes or actions is also possible.

2. *Industrial input formats and formal verification:*
   from synchronous languages to timed Kripke-structures

   Synchronous languages have been established in the industry for the design of reactive systems. In contrast to other real-time verification tools which read special input formats, synchronous languages offer a very attractive alternative for real-time verification purposes of industrial applications. Their advantage over many other system-level description languages, is their clean formal syntax and –semantics, which makes them well suitable for formal verification, allowing the generation of formal models.

   Within the scope of this work, we have developed methods that allow the generation of timed Kripke-structures directly out of synchronous languages. This enables for the first time a bridging between industrial real-time descriptions and formal real-time verification, i.e. a complete, formally verifiable design, development and realization of a real-time system out of an industrial real-time description. Up till now, other approaches must take special real-time description formats into account. This required a complete re-description of the system (or the use of special front-ends), which made their use for industrial applications very complicated or even impossible.

   A common characteristic of real-time systems is that they usually consist of many components operating in parallel; hence, they are concurrent systems. For symbolic model-checking there are no algorithms known, which can directly handle the parallel execution of processes. Therefore, up to now it was necessary to create a formal model to be verified by parallel composition of single real-time sub-models, which, however, combined with the use of real-time constraints lead to enormous state explosion. Furthermore, using such parallel composition techniques often leads to composed systems which violate fundamental requirements of a formal real-time model, due to timelock and deadlock problems.

   In order to overcome the above described problems, we have introduced techniques, which enable the direct generation of timed Kripke structures without parallel composition of single formal sub-models.

   Furthermore, to guarantee the correctness of a real-time system, it is not enough to verify its time constraints at an early design stage only, since the execution times of the system will depend on the target machine. Hence, for low-level real-time verification purposes one must be able to consider the execution times of a system, i.e. the physical times required by the target machine.

Within the scope of this work, we have developed methods that capture the behavior of a system in earlier (high-level) as well as in later (low-level) design stages. This allows also a low-level formal verification by considering physical execution times. The model is obtained in two steps: first a special, unit-delay Kripke structure is generated by the hardware-synthesis method. Afterwards it is transformed into a timed Kripke structure as follows:

- *Generation of high-level (architecture independent) formal models*

  For this purpose, we have presented extension techniques of synchronous languages, which allow the description of real-time systems together with the use of abstractions. The approach was implemented in Quartz, which is a variant of the established synchronous language Esterel.

  In order to generate timed Kripke structures, we have developed efficient abstraction methods, which benefit from the advantages of symbolic techniques and can thus handle systems with large state spaces. The approach first verify if the chosen abstraction is too coarse. If the degree of abstraction is acceptable, the timed Kripke structure is generated by another efficient symbolic technique, which transforms the abstract parts of the model into timed edges. Otherwise, the method is able to determine the locations where the abstraction is too coarse.

- *Generation of low-level (architecture dependent) formal models*

  For this purpose, we have developed a new method for exact low-level runtime analysis of synchronous programs, which determines the exact runtimes of all transitions of a system and at the same time generates a timed Kripke structure out of the unit-delay structure. This approach uses also symbolic techniques and can efficiently handle systems with large state spaces. Our low-level runtime analysis is based on the translation of synchronous programs in executable code for common architectures. Note that other existing runtime analysis methods are not suitable for verification purposes, since they do not consider formal environments and mostly give information only about the longest and shortest paths of a system.

3. *Real-time specification and verification logic:* JCTL

   For the specification and verification of real-time properties on timed Kripke structures, it was necessary to define a new temporal logic: JCTL was developed as a real-time extension of the temporal logic CTL. In contrast to other real-time logics, JCTL is directly defined on timed Kripke structures.

   JCTL allows the abstraction of irrelevant qualitative properties, without loss of any quantitative information. Should the abstraction be chosen to coarse, the model checking can alternatively be performed either by reduction of the abstraction, or on the first generated unit-delay structure.

The verification of JCTL formulae introduces new techniques, which are based on the partitioning of the calculation in a qualitative and a quantitative part. In particular, it is shown that JCTL can be defined completely on the basis of just a few basic operators. A great number of further JCTL operators can be defined as abbreviations of these basic operators and qualitative CTL-operators. Note that other abbreviation rules, already known from CTL, are not valid for real-time JCTL-operators.

The algorithms for model-checking JCTL formulae are based on fixpoint iterations. The state space is traversed by breadth-first search, using efficient symbolic techniques. The basic idea consists of interrupting the fixpoint iterations, when the time constraint of the temporal operator is reached.

As a direct result of the JCTL symbolic techniques, we have also developed efficient methods, which allow exact high- and low-level WCET and BCET analysis of real-time systems. For this purpose, the minimum and maximum computation paths are determined by breadth-first search through the state space of the model.

Chapter 1

Introduction

Σοφώτατον χρόνος· ἀνευρίσκει γάρ πάντα. [1]

ΘΑΛΗΣ (640 - 546 π.Χ.)

(Διογ. Λαερτ. Βίοι Φιλ. I, 35)

Nowadays the use of computers to create, store, exchange and use information has become a part of our everyday life. In fact we trust computers more and more to control safety-critical applications, like aircraft- and automobile systems, railway switching systems, banking systems, communication systems, complex production processes, nuclear power plants and military systems such as missiles [23, 25].

In most of these applications computer failures can lead to economical damage, environmental catastrophes, and in some cases, loss of human lives. Over the past decades, it has been a challenge for computer scientists to develop theories and techniques that guarantee that computer systems operate correctly, i.e., according to given specifications expressing their desired behavior. Moreover, decreasing time-to-market and the overall design costs require to check as early as possible in the design flow whether the desired specifications are met and detect logical design errors before they have been implemented. The traditional ways of debugging system designs have been simulation and testing. However, in many applications this process is exceedingly time-consuming and often provides only probabilistic measures of correctness. It is not unusual that more than 70% of the design time is spent for simulation.

To overcome this problem, many mathematical techniques for reasoning about the correctness of a system have been proposed [69, 43, 101, 76, 70, 96, 71]. These are generally known as *formal methods*. The idea is hereby to describe a system in a formal framework and then to apply mathematical methods to obtain a *formal verification* of the system, i.e., a *mathematical proof* for its correctness with respect to given *formal specifications*.

However, large formal system descriptions often tend to become complex and are therefore generally difficult to analyze. Such approaches can involve generating and proving literally hundreds of theorems and lemmas in detail, which requires also very experienced and mathematically oriented designers.

[1] Time is the wisest of things, for it finds out everything.
THALES (640 - 546 BC)

To overcome this problem, techniques have been proposed to automatically perform formal verification. One of the most successful and reliable approaches for the verification of finite state systems has been *model checking*, introduced by Queille and Sifakis [105] and Clarke and Emerson [32]. In contrast to manual techniques, model checking is completely automatic in the sense that the proof showing that a system satisfies the required specifications is automatically generated: properties are given as formulae of *propositional temporal logics* [101, 48] and automatically verified by a graph-theoretic analysis of a *formal model*, which is usually a state transition system *(Kripke structure)*. Informally, a Kripke structure consists of a finite number of states representing the system's state space and a number of transitions between states to capture the system's actual behavior.

The model-checking approach to the automatic verification of concurrent systems is one of the most impressive successes of theoretical computer science. 20 years after the first publications established model-checking as a theoretical possibility, the hardware industry employs several hundreds of highly specialized researchers working on the integration of model-checking techniques in the design process.

One of the advantages of this method is its efficiency: model checking is linear in the product of the size of the structure and the size of the formula, when the logic is the *branching-time* temporal logic CTL *(computation tree logic)* [33]. By developing special programming languages for describing transition systems, it became possible to check examples with several thousand states. This was sufficient to find subtle errors in a number of nontrivial, although relatively small, protocols and circuit designs [16]. If a formula is not true, model checking is able to 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.

The main practical limitation for model checking algorithms is caused by the so-called *state explosion problem*: the size, i.e., the number of states of the system can exponentially grow with the size of the implementation description. One popular approach to solve this problem relies on the *symbolic* (rather than *enumerative*) *representation* [2] of sets of states: hereby, sets of states are represented by a formula of state predicates. While the theoretical possibility of symbolic model checking by computing *fixpoints* was realized early by Emerson/Clarke [50] and Sifakis [118], the method has become practical only later, through the symbolic representation of state sets by *binary decision diagrams (BDDs)*, proposed by Bryant [17]. BDDs were first used for verification purposes by Coudert, Berthet and Madre [37] and for model checking by Burch, Clarke, McMillan, Dill and Hwang [21]. Representing transition systems implicitly using BDDs made it possible to verify systems with more than 10^{20} states. Refinements of the BDD-based techniques [18] have pushed the state count further up over 10^{100} states. Using other sophisticated methods like *partial order reductions* [100] and *symmetries* [52] allowed the verification of systems with even more states.

However, to be able to cope with large industrial examples, it was necessary to take advantage of methods based on different kinds of *abstractions*. Here, some details of the system can

[2] The notion of 'symbolic' representation has nothing in common with symbolic computations performed by computer algebra systems.

be declared to be irrelevant for verifi cation purposes, so that the verifi cation can concentrate only on the necessary facts and the resulting state space of the system to be verifi ed can be kept smaller. Such techniques have been developed in [35, 90], and are closely related to Cousot's *abstract interpretation* [39, 38] of programs.

1.1 Real-Time Systems

Only about 2% of the microprocessors that have been produced in 1999 were contained in desktop computers, while the vast majority was contained in so-called *embedded systems* [63, 107]. Embedded systems are parts of aircrafts, automobiles, domestic appliances, consumer electronics, etc. These systems do not have a user interface, but interact directly with their environment. Moreover, a great number of these systems are *reactive systems* [102]: Reactive systems are physical systems that respond to external *stimuli* and create desired effects in its environment by enabling, enforcing or preventing *events* in the environment. This defi nition encompasses all physical systems that exhibit some organized behavior, as well as interconnections of such systems into possibly large networks of dynamic and interacting components. Process controllers or signal processors are typical reactive systems.

A major factor which is of essential importance for the correctness of a system is *time*. Systems whose behavior depends on time constraints are called *real-time systems*. Designing a real-time system is a relatively error-prone task, especially when the system consists of several interacting processes, which is the usual case. The class of real-time systems includes many embedded and safety-critical systems that are subcomponents of larger complex systems operating in safety-critical environments [121]. This makes their correctness mandatory: the essential task is to guarantee that certain actions are executed within some strict deadlines or that they will start only after some point of time. To avoid expensive redesigns, it is important to check as soon as possible in the design flow whether these real-time constraints are met. In other words, not only the *qualitative* properties of the system are of importance, but also the *quantitative* ones.

To guarantee both, the qualitative and quantitative properties of real-time systems, several approaches to their formal verifi cation have been developed [2, 67, 27, 83, 6, 85, 40, 108] that are based on different formalisms for describing fi nite state transition systems endowed with some notion of time. Generally, an extended formal framework is required which usually consists of

- a real-time description language to capture the behavior of systems
- a real-time formal model to represent systems in a formal way
- a real-time temporal logic to express properties that a system should satisfy
- techniques for analyzing real-time systems with respect to properties

Traditionally there are two different ways to handle time constraints: *discrete time models* use the domain of integers to model time, while *dense time models* use the domain of reals. According to the time model, different specifi cation languages, formal models, temporal logics and verification techniques must be taken into account.

Generally, there are two main problems for the verification of real-time systems:

- the state space explosion

  A common characteristic of real-time systems is that they usually consist of many components operating in parallel; hence, they are *concurrent systems*. The parallel execution of processes combined with the use of real-time constraints causes in existing approaches an enormously increased state space explosion.

- the gap to industrial description languages

  Just like all other systems, real-time systems are usually described in industrial input languages, like VHDL, VERILOG, Esterel, C, C++, System C, etc. Unfortunately until now the existing real-time verification techniques require the use of some special input formats (like the ones considered in sections 1.1.4 and 1.1.5) for the description of real-time systems in order to obtain formal models that can be used for verification purposes. Taking advantage of these techniques in industry applications which are already described in other formats, requires their re-description in the appropriate special format, which is often a very complicated task. Furthermore it can not be guaranteed that the re-described system will have the same behavior as the original one.

In the following sections, we will consider the above mentioned verification problems together with the existing time models in greater detail.

1.1.1 Abstractions on Real-Time Systems

One of the most powerful methods that can be applied to fight the state explosion problem are abstraction techniques. Such methods are usually applied for the proof of qualitative temporal properties, like *safety properties* meaning that a property always holds, or *liveness properties* meaning that a property will hold at least once.

The basic idea is thereby to apply a predefined abstraction function to obtain an abstract system with a smaller state space. This is often the only way to make automatic finite-state verification procedures applicable to large systems. Even generic or infinite state spaces can be reduced to finite ones [82].

Abstraction techniques are of essential importance for the verification of real-time systems, where the state space explosion is caused by more than one factor, i.e. time constraints combined with parallel execution of processes. As this is not unusual for real-time systems, the state-explosion problem is even more serious for them: Experiences with automatic verification tools [6, 68, 41] have proved this claim.

First approaches to abstraction techniques for the verification of real-time properties have been developed in [78, 124, 42, 81, 82]. Unfortunately, none of these techniques can work without having to compromise on the quantitative information of the system, or on the expressiveness of the used temporal logic, or even on both.

Approaches considering dense time for example, must take abstractions like *region graphs* [3] or *quotient graphs* [124] into account, in order to reduce infi nite state spaces (arising due to the use of real numbers) into fi nite ones, so that symbolic verifi cation techniques can be applied:

Using so-called *strong time-abstracting bisimulations* [124], it is possible to achieve such reductions that preserve branching-time properties. However, the coarser the chosen abstraction is, the higher is the state space reduction, but also the more information is lost.

Generally, when sets of states are collected into abstract states, this means that the number of transitions to reach a certain state from another one is changed. As a consequence, information about quantitative time consumption is lost.

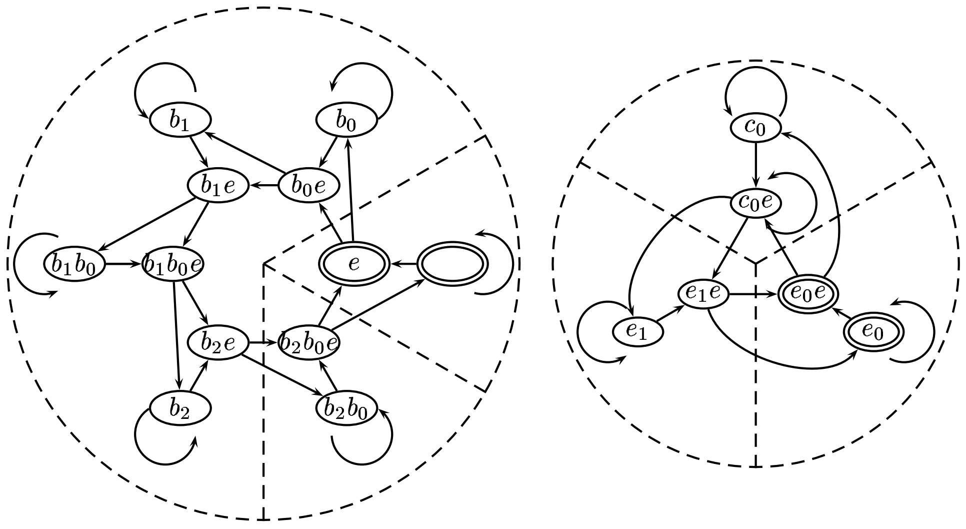


Figure 1.1: Abstraction of a 3-bit Modulo 6 Counter

An example is shown in Figure 1.1. The left hand side of Figure 1.1 shows the Kripke structure of a modulo 6 counter that counts whenever an enable signal e is seen. The counter's value is given by the variables $B := \{b_2, b_1, b_0\}$. The right hand side shows an abstract structure $\mathcal{K}_{abs}$ that is obtained for the values $A := \{0, 5\}$. In Figure 1.1, we have given segments of states where the counters value equals to 0 and 5 or is between these numbers. The counter of the concrete structure needs a minimum of 6 logical time units to count from 0 to 5. In the abstract structure however, this information is lost: we know that *some* time passes, but we don't know *how much*. Note also that the abstract structure contains more paths: we can remain infi nitely often in the state labeled with c_0e in the abstract structure, but there is no corresponding path in the concrete structure. Hence, the abstract structure is not simulated by the concrete one.

1.1.2 Description Languages for Real-Time Verification Purposes

As we explained in the previous sections, many approaches to verify real-time properties of formal real-time models already exist. Unfortunately, all existing real-time verifi cation tools like [29, 6, 40, 108] read special formats that describe the formal models directly. Hence, there are two possibilities for an application of these tools in an industrial design flow:

- Re-describe the systems in the appropriate special format, which is often a very complicated task. However, it can not be guaranteed that the re-described system will have the same behavior as the original one.
- Develop appropriate front-ends that compile system descriptions in the appropriate special format.

A great number of real-time systems can be considered as reactive systems, as they must react to stimuli of their environment. The examples mentioned above are all real-time systems. Consider for instance an airbag system in a modern vehicle. Sensors provide it with information of the current environment. In case of an accident, it is not enough to only guarantee that the airbag will just operate. It is of essential importance to guarantee that it will react and operate *within some time*. If it does not react in time, the system is not correct and will probably be of more harm than help to the driver.

A very important and successful approach for the description of reactive systems was the introduction of *synchronous programming languages* [60] like Esterel [8, 10], Lustre [61], Quartz [111, 112, 85] or other Esterel-variants [44, 74]. Synchronous languages can handle communication in reactive systems by instantaneous broadcasting. They are becoming more and more attractive for the design of reactive real-time systems. These languages have a discrete model of time, i.e. time is modeled by natural numbers $\mathbb{N}$. The basic paradigm of synchronous languages is the *perfect synchrony*: synchronous languages distinguish between *micro* and *macro steps* [64, 115]: The execution of a synchronous program from one point of time t to the next $t + 1$ is called a macro step and involves the execution of several, but always fi nitely many, micro steps. Micro steps are statements that are executed within zero time (in the programmer's model). A macro step consists of a fi nite number of micro steps and consumes always one logical unit of time after the execution of its micro steps. Further consumption of time, i.e., the beginning of a new macro step, must explicitly be programmed by means of special statements like Esterel's **pause** statement that consumes one logical unit of time. Synchronous languages are designed in such a way that a macro step can never contain a loop of micro steps. Instead, each loop of a synchronous program must contain at least one macro step. Consequently, all threads run in lockstep and automatically synchronize at each macro step.

Concerning the data flow, each variable, and hence, each data expression has one and only one value for each macro step. Hence, the semantics of a data type expression is a function of type $\mathbb{N} \to \alpha$ for some type α. The manipulations of the variables of a program are performed as micro steps of a macro step. These assignments or signal emissions determine the values of the variables at the current and the next macro step (this may result in so-called *causality problems*, but these problems have already found good solutions [9, 15], so we do not consider this issue here).

Hence, the entire semantics of a synchronous program $\mathcal{P}$ can be given as a fi nite state transition system $\mathcal{A}_\mathcal{P}$: the states of $\mathcal{A}_\mathcal{P}$ reflect the possible combinations of control flow locations of the program (a control flow location is a point in the program text, where the control flow might rest for one unit of time). As the language allows the implementation of parallel threads, there might be more than one current position of the control flow in the program. A transition between two control states is enabled if some condition on the data values is satisfi ed. Execution of a transition will then invoke some manipulations of the data values. Hence, the semantics can be represented by a fi nite state control flow that interacts with a data flow of fi nitely many variables of possibly infi nite data types.

To summarize, synchronous languages have important advantages for the specifi cation and verifi cation of real-time systems, since:

- synchronous languages support both, the design of software and hardware. They have notions of time at a logical level and statements to control threads like preemption and suspension. In early design phases, if the complete realization of a system is not yet known, one can not argue about physical time, since this depends on the hardware chosen for the realization. Using synchronous languages, one can achieve also realization independent descriptions of the system, which makes also possible to *reason about time at a logical level*.
- synchronous languages have clean formal semantics which allows the generation of formal models: Their compilation leads to a transition system which can be used for formal verifi cation purposes. A formal verifi cation of the semantics of synchronous languages by means of the HOL theorem prover [**?**] was performed in [111]. Furthermore, in [112, 5] was shown how asynchronous and nondeterministic systems can be modeled by synchronous languages.

Nowadays, synchronous languages are used in many industrial applications [54, 55, 123] and there already exist tools [62, 73, 112, 46] for verifying qualitative temporal properties of synchronous programs by symbolic model checking. In contrast to existing real-time verifi cation tools like [29, 6, 40, 108] which read special input formats, synchronous languages offer a very attractive alternative for real-time verifi cation purposes of industrial applications.

Unfortunately, far less is known about translating synchronous programs describing industrial real-time applications to real-time formal models necessary for later verification.

Furthermore, while the micro steps of a synchronous program are executed within zero time in the programmer's view, this is not the case for an implementation. Here, the micro steps *will consume physical time* $t > 0$, which depends on the chosen architecture[3]. Hence, for low-level real-time verifi cation purposes, one must be able to consider the execution times of a program, in other words, the physical times required by the micro steps.

Consequently, low-level real-time verification must be performed with respect to timed macro steps, i.e., transitions that correspond to non-interruptible atomic timed actions.

[3]Generally, it is viewed as a good programming style when the actual runtime of the macro steps is balanced, i.e. when all macro steps require equal or similar amount of physical time.

In the next section we will consider methods for determining execution times of a given program in more detail.

1.1.3 Runtime Analysis of Real-Time Systems

A popular method for determining execution times of a given program is *worst case execution time (WCET)* analysis [104]. It consists usually of two phases: the *high-level* and the *low-level*.

High-level WCET analysis is applied to an architecture independent description of the system and has the task to compute *path information* [103] like unfeasible computations or bounds on the maximal number of loop iterations. The results of the high level runtime analysis give important hints on bottlenecks of a real-time system in a very early design phase where not even prototypes of the system exist. Such fi gures are important for further design decisions like the hardware / software partitioning of an embedded real-time system or the choice between different microcontrollers. Unfortunately, high-level WCET analysis is undecidable when infi nite data types are used, and therefore only limited automation can be achieved. State of the art approaches use abstract interpretation [53], symbolic execution [91, 80], or special restrictions on the loops [66]. A major problem of the high-level WCET analysis is that the maximal number of computation steps of a statement (like a loop) may heavily depend on the input data, but some of the approaches do simply compute the WCET bounds for the substatements and add these afterwards. However, simply adding the maximal bounds for all substatements clearly yields highly pessimistic bounds. To estimate tight bounds, [7] proposed not to compute constants, but functions that depend on inputs as WCET bounds. This approach is able to compute tight bounds, but is certainly a deeply manual task, although guided by computer algebra systems.

Low-level analysis is done on the object code, and hence depends on the chosen hardware / software partitioning and the chosen architecture (microcontrollers). For simple microcontrollers like the still mainly used 8-bit processors, this is a straightforward task, but it is more complicated for modern architectures that require the consideration of caches, pipelines, branch prediction, interrupts, etc. There exist several approaches to estimate the worst- or best-case execution time of a given program on architectures with caches or super scalar execution by runtime analysis [65, 58, 97, 98].

Unfortunately, considering worst- or best-case execution times only, based on WCET analysis is not advisable for real-time verifi cation purposes, since this would yield in very inaccurate results. To be able to reason about real-time properties of a system, it is necessary to *consider and store in a formal model the exact execution times of all possible single transitions of a system, not only of its shortest or longest path*.

Some approaches like [65] can perform a so-called “point-to-point” analysis, but they do not consider formal semantics or formal models, so it is not possible to consider their results in a formal framework. In [117], an extension of the synchronous language Esterel has been presented that focuses on the runtime verifi cation of the perfect synchrony. However, it is assumed that the compiler preserves the ordering of the micro step statements, and therefore the approach is restricted to special compilation techniques like those proposed in [45]. A more powerful approach has recently been presented in [12]. There, Esterel programs are endowed with pragmas that contain the quantitative temporal information. This has no effect on code generation, but allows the generation of timed automata [2] (cf. section 1.1.5) for verification of temporal properties. However this approach also requires a low-level worst-case execution time (WCET) analysis in advance to obtain the real-time constraints. In other words, [12] can not refer to system descriptions in early design phases, since it needs additionally architecture-dependent runtime data which can only be obtained in late design phases. Considering WCET analysis, [12] can only refer to execution time results about longest execution paths and can not consider exact execution times. This reduces significantly the accuracy of the verification results obtained by [12]. Furthermore, it does not support abstractions, necessary for the verification of complex industrial applications.

1.1.4 The Discrete Time Approach

Considering time constraints with integers usually requires a state transition system *(Kripke Structure)* as formal model. Real-time extensions of the branching time temporal logic CTL are usually considered in order to perform a graph-theoretic analysis of Kripke structures with respect to time constraints. To denote the time required to move from one state to another, the transitions of the system are often labeled by numbers. In general, there are two possible interpretations of such structures:

Jumping Interpretation I_J**:** A transition from state s_1 to state s_2 with label $k \in \mathbb{N}$ means that at any time t_0, where we are in state s_1, we can perform an atomic action that requires k units of time. The action terminates at time $t_0 + k$, where we are in state s_2. *There is no information about the intermediate points of time* $t_0 < t < t_0 + k$.

Stuttering Interpretation I_S**:** A transition from state s_1 to state s_2 with label $k > 1$ is seen as abbreviation for a stuttering sequence $s_1 \xrightarrow{1} s_{1,1} \xrightarrow{1} \dots \xrightarrow{1} s_{1,k-1} \xrightarrow{1} s_2$ where s_1 is repeated, according to the time consumption.

Clearly, only interpretation I_J can be used in a setting where non-interruptible, atomic timed transitions are considered. Moreover, only interpretation I_J allows more powerful abstraction techniques than stuttering simulations. It is therefore surprising that none of the existing real-time extensions of CTL is based on interpretation I_J, although this is the more general (expressive) one. In this work we define *timed Kripke structures (TKSs)* as the ones labeled according to jumping interpretation I_J. To distinguish between timed and untimed Kripke structures we call the untimed ones also *unit delay structures (UDSs)*.

The development of discrete real-time extensions of CTL has been initiated by Emerson et al. in [47, 49, 51], where the temporal operators have been extended by time bounds to limit the number of fixpoint iterations required to evaluate the considered temporal expression.

The models used in [47, 49, 51] were still traditional finite-state transition systems where each transition requires a single unit of time.

In order to represent real-time systems in a more compact way, [27] introduced timed transition systems, where transitions are labeled by natural numbers that denote the time consumption of the action associated with the transition. The meaning of these timed transitions is in our terms the stuttering interpretation I_S.

In [57] a new real-time temporal logic was introduced, where both interpretations I_J and I_S were unfortunately mixed: Different temporal operators of this logic interpret transitions differently, i.e., either according to I_J or I_S. As the meaning of timed transitions therefore depends on the context, it is impossible to reason about the meaning of timed transitions. In particular, it is not possible to define composition of structures. Moreover, the temporal operators of [57] that rely on interpretation I_S have a misleading semantics as we will outline in detail in section 4.2.1.

Finally, [109] became aware of the misleading semantics of [57] (see page 11 of [109]), and therefore defined another temporal logic that is solely based on interpretation I_S. In principle, their logic is defined on unit delay transition systems that are obtained by expanding (cf. section 4.1) a given timed transition system. As we will explain in section 4.2.2, different expansions of a timed transition system yield in different transition systems that are not bisimilar to each other, so that the results obtained for an expanded structure can not be easily transferred back to the timed transition system [83]. Hence, the work of [109] is essentially based on expanded, i.e., untimed transition systems, and may therefore be more or less viewed as a variant of Emerson's work [47, 49, 51].

To summarize, the mentioned real-time extensions of CTL have the following problems (we discuss them in more detail in chapter 4):

- None of the mentioned real-time extensions of CTL is based on interpretation I_J that is necessary to benefit from abstraction techniques.
- None of the mentioned real-time extensions of CTL has the ability to consider timed transitions as *atomic* ones. They all must assume that a single timed transition is a stuttering sequence of a number of *unit delayed* transitions and must therefore take several consequences into account. For example, none of these extensions has a *time bounded next state operator* to express facts about actions that correspond with a single transition. Hence, properties regarding *non-interruptible atomic timed actions* necessary for the low-level real-time verification of synchronous programs can not be handled. This is also the case for all non-interruptible processes, like *'Is there a non-stop flight from New York to Paris with a duration of at most 9 hours?'*
- The mentioned real-time extensions of CTL compute the set of states of a timed transition system $\mathcal{K}$ where a real-time CTL formula holds, by translating the problem to a CTL model checking problem on a unit delay structure. For this purpose however, they take certain *assumptions* about the *expansion semantics* of the timed system into account. However, different expansion semantics of timed transition systems may yield in different results (s. section 4.2.2), and furthermore, different expansions are not equivalent to each

other [83]. Since there are multiple ways to expand a timed system, the choice of a certain expansion semantics is unclear. Moreover, it is not satisfactory to assume that *all* transitions of a timed system behave according to the same expansion semantics.

Hence, there has been much confusion and misconception about the definition of real-time extensions of the famous temporal logic CTL. The previous approaches [27, 57, 109] have all been implemented in efficient symbolic model checking tools, but the underlying formalisms are – from a logical perspective – questionable.

Furthermore the above mentioned approaches have to take enormous state space explosions into account due to parallel execution of processes. In [108] for instance, it is required to model the system as a parallel composition of several single timed transition systems. Since these single systems can not be easily composed, it was proposed to expand (cf. section 4.1) them first, compute their product as unit-delayed systems and finally, transform them back into timed transition systems again. Moreover, as discussed above, assuming some certain *expansion semantics* (cf. section 4.2.2) leads to questionable results.

1.1.5 The Dense Time Approach

Beneath the real-time extensions of CTL that are defined on discrete time models, there are also very successful approaches that are based on a continuous model of time [2, 67, 40, 6], i.e., they use the domain of real numbers to represent time. These approaches usually rely on *timed automata*, [4] *timed petri nets* [106], or *timed process algebras* [99]. The most popular of these formalisms are timed automata. These are finite state automata endowed by a finite set of real-valued clocks. Verification purposes based on timed automata use the *timed computation tree logic* TCTL, introduced in [3] as a real-time extension of the branching-time logic CTL.

However, as already mentioned in section 1.1.1, abstractions/approximations must be taken into account in order to reduce the infinite state spaces (arising due to the use of real numbers) into finite ones. Then TCTL model checking can be reduced to CTL model checking. For this purpose, constructions of finite state systems, like *region graphs* [3] or *quotient graphs* [124] were proposed. The basic idea in such systems is that each region essentially consists of the set of concrete states that are equivalent, in the sense that they can lead to the same regions in the future. Approaches associated with such models - especially with region graphs [1, 92, 30], or if consideration of symbolic (branching-time) properties is required - are often in practice computationally infeasible because of the huge size and the very expensive construction of the finite systems. The region graph for instance, grows exponentially, not only in the number of automata and the number of clock variables, but also in the largest integer constant used in the clock constraints of the automata.

Timed automata are considered to be a powerful formalism, since they allow processes to have several clocks starting, pausing, or reseting independently of each other and of the environment. For the verification engineer, it is possible to start, pause or reset these clocks of each process individually, according to the specifi cation requirements. However, problems arise when parallel execution of processes must be considered. The fi nal system to be verifi ed must be obtained using parallel composition of timed automata. However, the composed system often violates fundamental requirements of a formal real-time model [125]: In such a model, it should be possible to take discrete transitions infi nitely often (discrete progress), but also to let time pass infi nitely often, and this *without upper bounds* (time progress). The implicit requirement of time progress, known as *non-zenoness* [67], means that the formal model should not contain *deadlocks*, i.e. states with no successors or *timelocks*, i.e. states from which the time cannot progress. When checking properties, only models with non-zeno behavior should be taken into account. A timed automaton is *deadlock-free* if none of its reachable states is a deadlock and *timelock-free* if none of its reachable states is a timelock. When the time progress at a certain state of a timed automaton stops, then some action becomes *urgent* at this state [13]. In order to capture deadlock and timelock freedom, the notion of *well-timed* systems was introduced in [67].

Deadlocks and timelocks are modelling errors caused by the verifi cation engineer, since any timed automaton assumed to capture the behavior of a reactive system should act infi nitely often and should not block time. Composing independently time steps and transitions may easily lead to such problems. The compositional description of timed systems that satisfy even weak well-timedness requirements, is a very complicated task [14]. The timed automata approach still lacks a robust theory of composition, which would avoid deadlocks and timelocks, but also preserve the independence of the urgency requirements of the components. First approaches for a formal framework for urgency have been introduced in [119, 14]. However, these approaches assume a special sub-class of timed automata and it still remains to see if analysis techniques can be extended to consider such frameworks.

1.2 Contributions of this Work

The systems in the world we are living in are *already synchronized* according to physical laws. We consider therefore as not natural to assume that a subsystem behaves according to its own clock which starts, pauses or resets independently from some clocks of other subsystems that communicate and interact with each other. This assumption leads to synchronization-, urgency- and timelock problems [125, 13, 119, 14], for example when single timed subsystems are ***first*** modeled as theoretically stand-alone parts and must ***then*** be composed in order to express their parallel operation and obtain a fi nal timed model for the entire, unique system. This is the case with the timed automata formalism: these problems arise in many cases due to inconsistent timed specifi cations, given by humans.

In the real world, it is much more natural to first accept the naturally given synchronization of all physical systems and then determine the elapsed time between their events, not vice versa.

Following these principles, our goal is to apply them to real-time systems:

- Define a real-time formal model that is based on jumping interpretation I_J (cf. 1.1.4) and allows the consideration of non-interruptible atomic timed transitions together with the use of abstractions without loss of quantitative information.
- Develop a technique that takes full advantage of an industrial used description language. Extend the language in order to use it for real-time specification- and abstraction purposes.
- Specify systems in such a way, that allows to easily compose them and construct *one single model* which is *already synchronized*. This model can then be easily endowed by time attributes determined by low-level analysis, or it can be transformed into a much more compact real-time system by the use of high-level abstractions. The final single real-time model does not suffer from the problems mentioned in sections 1.1.4 and 1.1.5. It can then be used for low-level or high-level verification purposes, according to the requirements.
- Define a real-time temporal logic that takes advantage of symbolic model checking techniques and can handle atomic timed transitions and abstractions without loss of quantitative information.
- Develop efficient real-time symbolic techniques for model checking and runtime analysis purposes

Finally, regarding the use of dense time, there is one question that still remains to be answered: *where does the dense time come from in praxis?*

It may sound logical to allow in a theoretical model for an event to take place after $\sqrt{2}$ seconds, but in today's praxis we are not able to *measure* such a time exactly.

In this work we present a complete formal framework for the specification, modelling, verification and runtime analysis of real-time systems, which we have implemented as our tool Equinox. The structure of this framework is given in Figure 1.2.

In the first place, we define *timed Kripke structures (TKS)* [83] based on jumping interpretation I_J (cf. section 1.1.4) as real-time formal model. The model considers timed transitions as atomic ones and allows the use of abstractions without loss of quantitative information.

We also introduce the branching-time temporal logic JCTL [83, 84] as the *first real-time extension* of CTL that is based on interpreting timed transition systems with interpretation I_J. JCTL directly allows the abstraction of real-time systems by ignoring their irrelevant qualitative properties, but without loosing their quantitative ones.

For example, we can model processes that compute some values within a certain limit of time with a single transition, that does not state anything about the values of the variables *during the computation*. In contrast to other real-time temporal logics, JCTL has a next-state operator equipped with time bounds, which make the logic powerful enough to reason about real-time constraints of atomic actions. We also present symbolic model checking techniques for JCTL that we have implemented in our model checking tool JERRY, using the CUDD BDD library [120]. Moreover, we analyze the complexity of the JCTL model checking algorithms. It turns

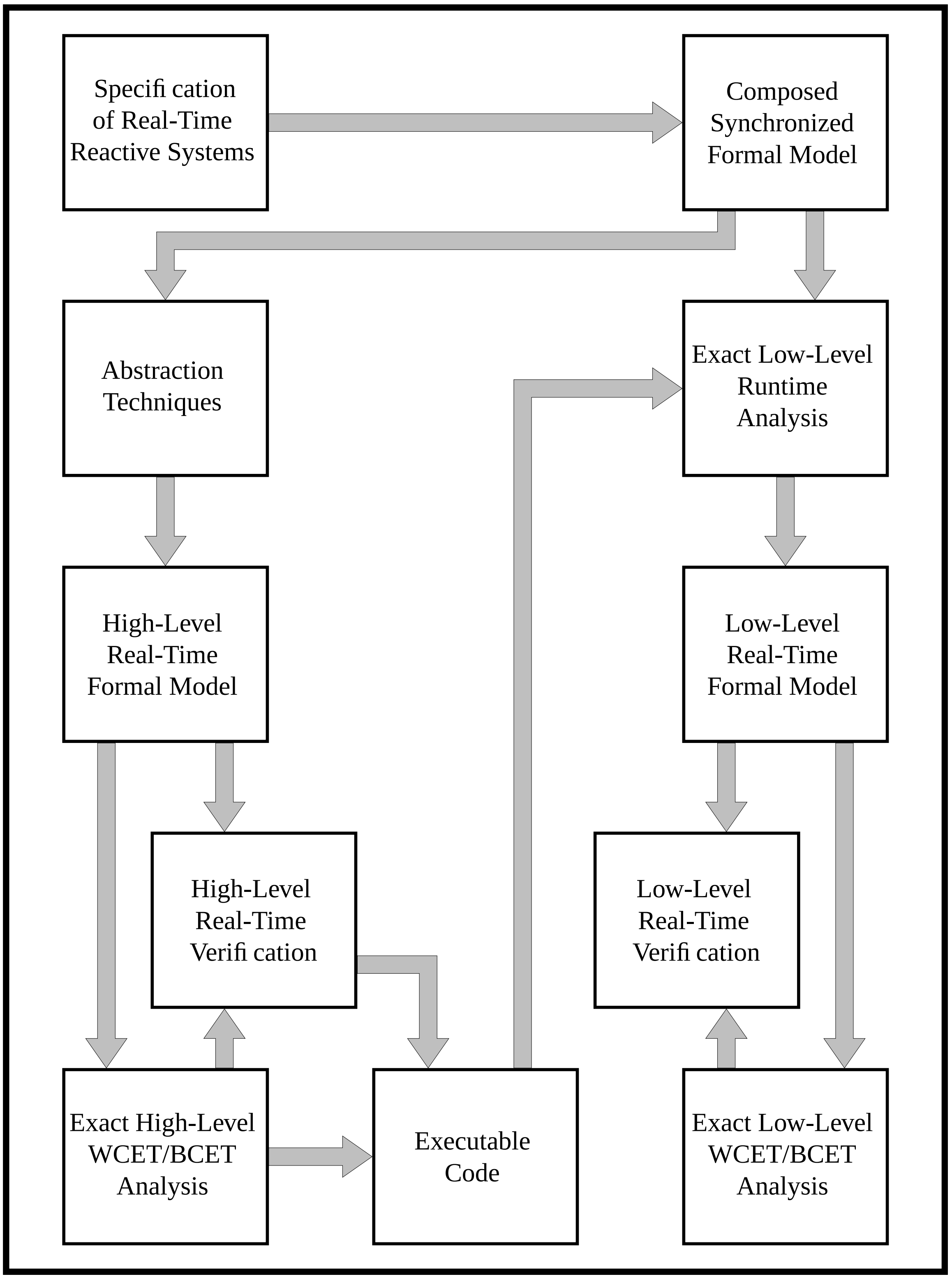

Specifi cation of Real-Time Reactive Systems
Composed Synchronized Formal Model
Abstraction Techniques
Exact Low-Level Runtime Analysis
High-Level Real-Time Formal Model
Low-Level Real-Time Formal Model
High-Level Real-Time Verifi cation
Low-Level Real-Time Verifi cation
Exact High-Level WCET/BCET Analysis
Executable Code
Exact Low-Level WCET/BCET Analysis

Figure 1.2: Equinox: A Formal Framework for the Specifi cation, Modelling, Verifi cation and Runtime Analysis of Real-Time Systems

out that the complexity to compute the set of states where a given JCTL formula holds is the same as for previous approaches like [57].

Furthermore, we propose techniques for the translation of synchronous programs in real-time models that are endowed with notions of either *logical* or *physical* time.

For models considering logical time, we introduce an extension of Esterel together with its translation to abstract real-time models (in the sense of [83]). Our extension allows the programmer to declare irrelevant program locations. After the usual compilation of the program into a transition system, certain states are thus to be ignored. For this purpose, we present an efficient algorithm that uses symbolic[4] techniques to generate a *single timed Kripke structure* as a *unique* real-time model from the output of the compiler [85]. The algorithms presented in [83, 84] are then used to check quantitative temporal properties of the generated TKS.

Our goal is to show how abstractions can be incorporated in synchronous programs to obtain abstract real-time models that retain the quantitative temporal information. In particular, the programmer can generate abstract real-time models without having the need of special knowledge about verification techniques. A well-known problem of all approaches based on abstraction techniques is that the chosen abstraction might be too coarse. In this case, our technique is able to detect the problematic program locations.

In order to generate models that consider physical time, we present a method that performs an *exact and detailed low-level (architecture-dependent) runtime analysis* of synchronous programs [89, 87, 88]. Our approach computes the exact execution times of *all possible single transitions* of a system and simultaneously generates a timed Kripke structure as real-time formal model, whose transitions are labeled with numbers denoting the exact execution times of the macro step that corresponds to the transition. Note that the generated low-level timed Kripke structure consists of timed transitions which are *non-interruptible atomic actions* that will synchronize at each macro step. Calculating exact execution times, our method overcomes the problems of other approaches (cf. section 1.1.3), which must take less accurate WCET results into account for the generation of their low-level models. The generated formal models can then be directly used for architecture-dependent real-time verification, as well as for further analysis purposes like WCET and BCET [86].

It is important to note that our methods allow the generation of real-time models without having the need of parallel composition of single submodels, which is one of the most important drawbacks of other approaches, as described in sections 1.1.4 and 1.1.5. Parallel composition is already performed during compilation of the synchronous program, before the generation of the single real-time model.

Having constructed real-time models considering logical as well as physical time, we then propose a new and completely automatic approach to *exact WCET analysis* of these models [86].

[4]The notion ‘symbolic’ is used here in the sense of ‘symbolic model checking’ which means that we represent transition relations and state spaces implicitly by means of propositional formulae.

Our technique can be applied to both, low-level– and high-level WCET analysis, according to the time notion of the model. For a given program, we take advantage of symbolic state space exploration to compute the exact best and worst runtimes in terms of macro steps for all inputs at once. Hence, our method overcomes the known problem of computing only highly pessimistic results due to simply adding maximal bounds (cf. section 1.1). We are even able to compute the input sequences that require these bounds. For this purpose, we have to assume that all data types were finite[5], so that the overall problem becomes decidable.

Generally, we propose the following design flow:

1. Describe the system under consideration as a synchronous program.
2. Verify desired specifications with real-time constraints [83, 84] and perform exact WCET analysis [86] at a logical level in order to find design errors and to estimate the runtime of complex tasks in terms of macro steps [85].
3. Based on the previous step, choose appropriate hardware to realize the embedded system and automatically derive code for software or hardware design [111, 112].
4. Perform exact low-level runtime analysis to determine the actual reaction time of the embedded real-time system.
5. Verify desired specifications with real-time constraints and perform exact WCET analysis at a physical level.
6. Generate verified executable code, circuit netlists, etc.

The outline of the work is as follows: In chapter 2 we give an overview of fundamental theoretical formalisms on which this work is based. In the first place, we present the formalisms on which symbolic model checking techniques are based like Kripke structures, binary decision diagrams and branching-time temporal logics. Then we will explain the basics of synchronous languages, which are widely used to generate formal models necessary for verification purposes. In chapter 3 we define our timed transitions systems and our real-time temporal logic JCTL. Having given the necessary definitions, we then discuss in chapter 4 the deficiencies of the above mentioned related approaches in more detail. In chapter 5 we will then proceed with the definition of a symbolic model checking procedure for JCTL and analyze its complexity.

In chapter 6 we present techniques that allow the translation of synchronous programs to timed Kripke structures which are endowed with notions of *logical*-, as well as *physical* time: In section 6.1 we extend synchronous languages by abstractions in order to declare irrelevant program locations. Then we generate an abstract real-time model by ignoring the irrelevant states, while retaining the quantitative information. In contrast to the programmer's model of synchronous languages, we consider in section 6.2 the physical time that is required for a particular hardware to execute the micro steps that belong to the corresponding macro step of a transition. Hence, at

[5]For embedded systems, this restriction is not a severe one. Moreover, modelling integers with a finite, constant bitwidth is even more accurate and allows one to detect problems with overflows and underflows.

this level micro steps do consume time, and it is a challenging task to compute this consumption of time for particular instances of processors.

In chapter 7 we propose an exact WCET and BCET analysis algorithm: Using symbolic state space exploration, we compute for all inputs the lengths of all computations between given program locations.

Finally, the case studies and experimental results that we have obtained by our tool Equinox are presented in chapter 8.

Chapter 2

Theoretical Background

Μή διά φόβον, ἀλλά διά το δέον ἀπέχεσθαι ἁμαρτημάτων. [1]

ΔΗΜΟΚΡΙΤΟΣ (470 - 369 π.Χ.)

(Απόσπ. 41, Diels)

In the following, we will give an overview of fundamental theoretical formalisms on which this work is based. These formalisms are very successful in the description, modelling and formal verifi cation of qualitative-only system properties and are commonly used in industrial applications. In the fi rst place, we will briefly present the formalisms on which symbolic model checking techniques are based like Kripke structures, binary decision diagrams and branching-time temporal logics. Then we will explain the basics of the synchronous language Quartz, which is widely used to generate formal models necessary for verifi cation purposes, thanks to its clean formal semantics.

2.1 Symbolic Model Checking

2.1.1 Kripke Structures

As explained in section 1, symbolic model checking approaches prove automatically the correctness of a system's specifi cations by means of a graph-theoretic analysis of a model that represents the system. It considers systems modeled as Kripke structures, which are state-transition graphs over some set of variables $\mathcal{V}$. Kripke structures are formally defi ned as follows:

Definition 1 (Kripke Structure) *A Kripke structure over the binary variables $\mathcal{V}$ is a tuple $(\mathcal{I}, \mathcal{S}, \mathcal{U}, \mathcal{L})$, where $\mathcal{S}$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{S}$ is a set of initial states, and $\mathcal{U} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of transitions. For any state $s \in \mathcal{S}$, the set $\mathcal{L}(s) \subseteq \mathcal{V}$ is the set of variables that hold on s.*

[1] Avoid errors; not for fear, but for duteousness.
DEMOKRITOS (470 - 369 BC)

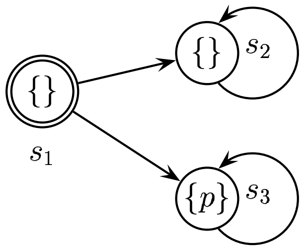


Figure 2.1: A Kripke Structure

Kripke structures may be pictorially drawn as given in Figure 2.1, where initial states are drawn with double lines. They consist of a fi nite number of states representing the system's state space and a number of transitions between states to capture the system's actual behavior. The choice of a transition is nondeterministic. The binary relation on $\mathcal{U}$ is total, i.e. each state has at least one successor. $\mathcal{L}$ is a labeling which assigns to each state a set of variables, those intended to be true at the state. Intuitively, the states of a structure could be thought of as corresponding to the states of a concurrent program, the state transitions of which are specifi ed by the binary relation $\mathcal{U}$. In symbolic model checking approaches the transition relation is represented implicitly by boolean formulae and states are not explicitly enumerated.

2.1.2 Binary Decision Diagrams (BDDs)

Boolean formulae can be represented by binary decision diagrams (BDDs) [17]. This usually results in a much smaller representation for the transition relation, allowing the size of the models being verifi ed to increase signifi cantly. BDDs are directed acyclic graphs. The nodes of a BDD correspond to the variables of the formula. Descendants of a node are labeled 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 that corresponds to the value assigned to the variable in the node is followed. The leaf indicates if the formula is satisfi ed or not for that particular assignment.

However, BDDs impose a total ordering in which the variables occur in this sequence. For any boolean formula and with a fi xed variable ordering there exists a unique BDD [17]. The size of the BDD 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 most practical cases. Using a good variable ordering is very important. But fi nding the optimal order is in itself a NP-complete problem. Nevertheless, there are many heuristics that work quite well in practice, like [56, 24].

2.1.3 Computation Tree Logic (CTL)

The branching-time temporal logic CTL (Computation Tree Logic) is used in symbolic model checking approaches to specify and reason about correctness properties of concurrent programs [33]. CTL is a temporal logic extended by *path-quantifiers*. Formally, the notion of a path in a Kripke structure is defi ned as follows:

Definition 2 (Path in a Kripke Structure) *A path σ through a Kripke structure is a function $\sigma : \mathbb{N} \rightarrow \mathcal{S}$ such that $\forall i \in \mathbb{N}.(\sigma^{(i)}, \sigma^{(i+1)}) \in \mathcal{U}$ holds (we write the function application with a superscript). Hence, $\sigma^{(i)}$ is the $(i+1)th$ state on path σ. The set of paths starting in a state s is furthermore denoted as $\mathcal{P}_{\mathcal{K}}(s)$.*

Since every state in a Kripke structure has at least one successor, a path $s_0, s_1, s_2, \ldots$ is an infinite sequence of states so that $(s_i, s_{i+1}) \in \mathcal{U}$. Intuitively, a path captures the notion of an execution sequence.
Generally, temporal logics are *propositional logics* combined with *modal operators*. They were originally developed by philosophers in order to reason about the validity of statements according to time. Typical modal operators of a temporal logic are:

- G: *global* validity at all points of time
- F: validity at some point of time in the *future*
- X: validity at the *next* point of time
- U: validity *until* some point of time in the *future*

These make possible to express properties like “nothing new happens”: $\mathsf{G}(\neg new)$. A popular logic that takes advantage of the above operators is the *Linear-Time Temporal Logic (LTL)* [101]. However, it is often of essential importance to be able to quantify over several paths while performing a system’s analysis. For this purpose, CTL was introduced in [33], as an extension of temporal logics by *universal path-quantifiers* $\forall$ and *existence path-quantifiers* $\exists$. Formulae 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. Each operator consists of two parts: a path quantifier followed by a temporal operator. Path quantifiers indicate that the property should be true on all paths from a given state (A), or on some paths from a given state (E). Formally, the definition of the syntax and semantics of CTL are given as follows:

Definition 3 (Syntax of CTL) *Given a set of variables $\mathcal{V}$, the set of CTL formulae is the least set satisfying the following rules, where φ and ψ denote arbitrary CTL formulae, and $a, b \in \mathbb{N}$ are arbitrary natural numbers:*

- $\mathcal{V} \subseteq \mathsf{CTL}$, *i.e, any variable is a CTL formula*
- $\neg\varphi$, $\varphi \wedge \psi \in \mathsf{CTL}$
- $\mathsf{EX}\varphi \in \mathsf{CTL}$
- $\mathsf{EG}\varphi \in \mathsf{CTL}$
- $\mathsf{E}[\varphi \ \underline{\mathsf{U}} \ \psi] \in \mathsf{CTL}$

The semantics of CTL is defined with respect to Kripke structures. If a formula φ is true in a state s of a structure $\mathcal{K}$, we write $(\mathcal{K}, s) \models \varphi$.

Definition 4 (Semantics of CTL) *Given a Kripke structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{U}, \mathcal{L})$, and $s \in \mathcal{S}$, then the semantics of the logic is recursively defined as follows:*

- $(\mathcal{K}, s) \models p$ *iff* $p \in \mathcal{L}(s)$ *for any* $p \in \mathcal{V}$
- $(\mathcal{K}, s) \models \neg\varphi$ *iff* $(\mathcal{K}, s) \not\models \varphi$
- $(\mathcal{K}, s) \models \varphi \wedge \psi$ *iff* $(\mathcal{K}, s) \models \varphi$ *and* $(\mathcal{K}, s) \models \psi$
- $(\mathcal{K}, s) \models \mathsf{EX}\ \varphi$ *iff there is a path* $\sigma \in \mathcal{P}_{\mathcal{K}}(s)$ *with* $(\mathcal{K}, \sigma^{(1)}) \models \varphi$
- $(\mathcal{K}, s) \models \mathsf{EG}\ \varphi$ *iff there is a path* $\sigma \in \mathcal{P}_{\mathcal{K}}(s)$ *such that for all* $i \in \mathbb{N}$ *holds* $(\mathcal{K}, \sigma^{(i)}) \models \varphi$

- $(\mathcal{K}, s) \models \mathsf{E}[\varphi\ \underline{\mathsf{U}}\ \psi]$ *iff there is a path* $\sigma \in \mathcal{P}_{\mathcal{K}}(s)$ *and an* $i \in \mathbb{N}$ *with*

$$\big((\mathcal{K}, \sigma^{(i)}) \models \psi\big) \wedge \big(\forall j < i.\ (\mathcal{K}, \sigma^{(j)}) \models \varphi\big)$$

Given a Kripke Structure $\mathcal{K}$ *and a* CTL *formula* φ*, we denote the set of states of* $\mathcal{K}$ *where* φ *holds as* $[\![\varphi]\!]_{\mathcal{K}}$*. We also denote the value of a formula* φ *in a state* s *with* $\varphi(\!|s|\!)$*.*

Intuitively, $(\mathcal{K}, s) \models \mathsf{EX}\ \varphi$ means that the state s has a at least one direct successor state s' that satisfi es φ.

$(\mathcal{K}, s) \models \mathsf{E}[\varphi\ \underline{\mathsf{U}}\ \psi]$ means that there is a path σ starting in $\sigma^{(0)} = s$, where either ψ immediately holds at $\sigma^{(0)}$, or φ holds for the fi rst i states $\sigma^{(0)}, \sigma^{(1)}, \ldots, \sigma^{(i-1)}$ and ψ holds on $\sigma^{(i)}$. If we allow that ψ *must not necessarily hold* on $\sigma^{(i)}$, then we call the operator a *weak until operator*, denoted without underline as $\mathsf{E}[\varphi\ \mathsf{U}\ \psi]$. Analogous, $\mathsf{E}[\varphi\ \underline{\mathsf{U}}\ \psi]$ is also often called *strong until operator*.

$(\mathcal{K}, s) \models \mathsf{EG}\ \varphi$ means that there is a path σ starting in $\sigma^{(0)} = s$, such that φ holds for any state $\sigma^{(i)}$.

In the above defi nitions, only basic operators of the logic were used. Further operators of CTL can be defi ned as abbreviations of the basic ones:

Definition 5 (Further CTL **Temporal Operators)** *Further temporal operators in* CTL *can be defined as follows, where* p *is an arbitrary variable:*

- $\mathsf{1} := p \vee \neg p$
- $\mathsf{0} := p \wedge \neg p$
- $\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$
- $\varphi \rightarrow \psi := \neg\varphi \vee \psi$
- $\mathsf{EF}\ \varphi := \mathsf{E}[1\ \underline{\mathsf{U}}\ \varphi]$
- $\mathsf{AX}\ \varphi := \neg\mathsf{EX}\ \neg\varphi$
- $\mathsf{AF}\ \varphi := \neg\mathsf{EG}\ \neg\varphi$
- $\mathsf{AG}\ \varphi := \neg\mathsf{EF}\ \neg\varphi$
- $\mathsf{A}[\varphi\ \underline{\mathsf{U}}\ \psi] := \neg\mathsf{E}[\neg\psi\ \underline{\mathsf{U}}\ \neg\varphi \wedge \neg\psi] \wedge \neg\mathsf{EG}\ \neg\psi$

CTL formulae enable the expression of properties regarding to several paths as illustrated in Figure 2.2. To summarize, CTL operators have intuitively the following meaning:

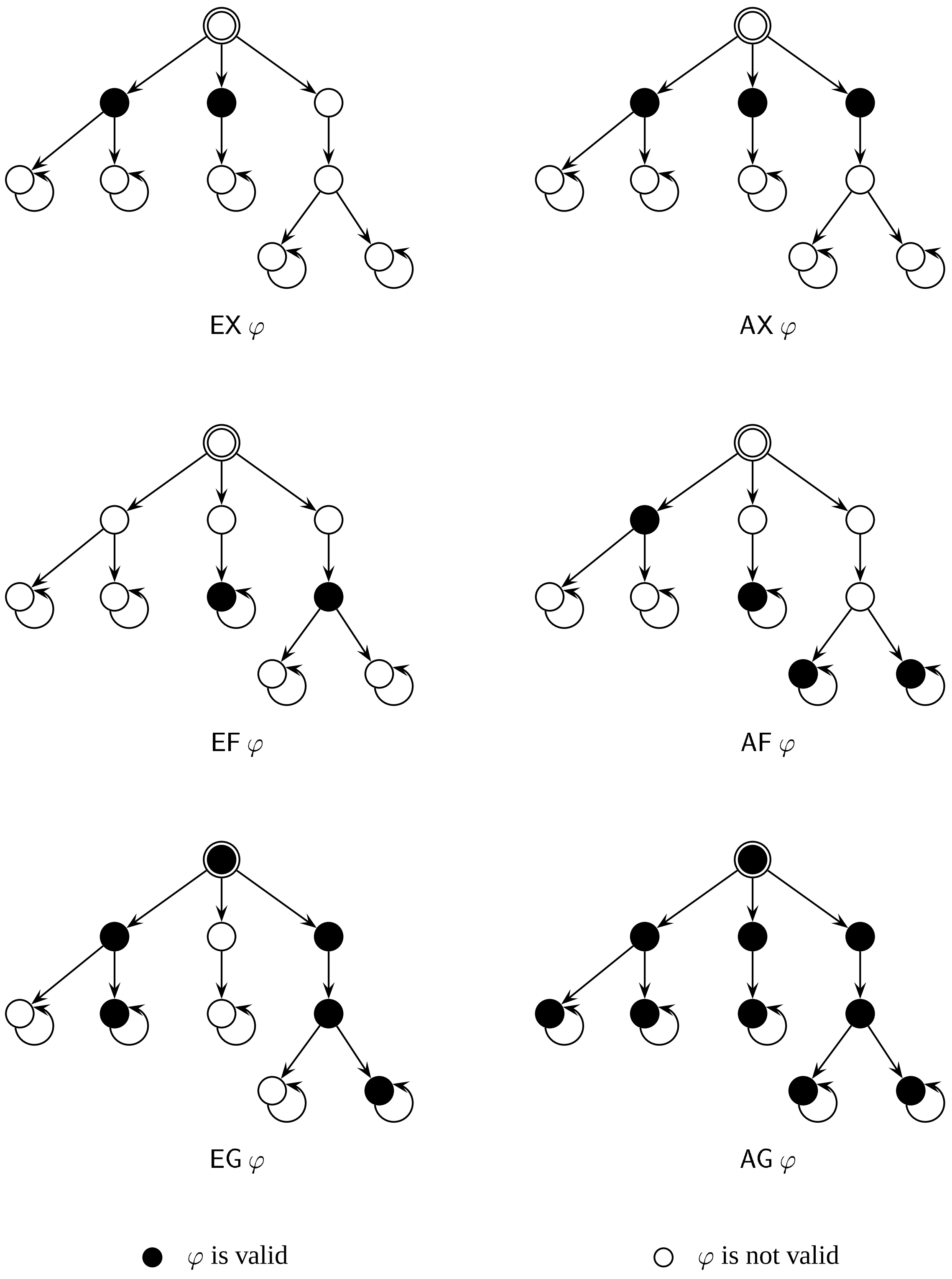


Figure 2.2: CTL Operators

- $\mathsf{EX}\ \varphi$: a state, where φ is valid is *reachable* at the *next* point of time
- $\mathsf{EF}\ \varphi$: a state, where φ is valid is *reachable* at some point of time in the *future*
- $\mathsf{EG}\ \varphi$: there *exists* an infi nite sequence of states, where φ is valid at all points of time
- $\mathsf{E}[\varphi\ \underline{\mathsf{U}}\ \psi]$: ψ is valid, or there *exists* a sequence of states, where φ holds until ψ holds
- $\mathsf{AX}\ \varphi$: the validity of φ is *inevitable* at the *next* point of time
- $\mathsf{AF}\ \varphi$: the validity of φ is *inevitable* at some point of time in the *future*
- $\mathsf{AG}\ \varphi$: φ is valid at all points of time
- $\mathsf{A}[\varphi\ \underline{\mathsf{U}}\ \psi]$: ψ is valid, or φ holds at all sequences of states, until ψ *inevitably* holds

CTL symbolic model checking is linear in the product of the size of the structure and the size of the formula [33]. The algorithms of the basic operators are based on *fixpoint computations*. In the following, we give a brief description of this notion. For a more detailed description, the reader is referred to chapter 3 of Schneider's [113] or to [34, 36].

Considering a set of states $\mathcal{S}$ of a Kripke structure, then one can denote a lattice $\mathcal{D}(\mathcal{S})$ of predicates over $\mathcal{S}$, where each predicate is identifi ed with the states in $\mathcal{S}$ that make it true and uses set inclusion as ordering.

Definition 6 (Fixpoints) *Consider a set of states $\mathcal{S}$ of a Kripke structure, a lattice $\mathcal{D}(\mathcal{S})$ of predicates over $\mathcal{S}$ and a function $\mathcal{F} : \mathcal{D} \to \mathcal{D}$ that maps sets of states to sets of states. A set $d \subseteq \mathcal{D}$ is a fixpoint of the function $\mathcal{F}$ if $\mathcal{F}(d) = d$ holds. The least fixpoint $\check{d}$ and the greatest fixpoint $\hat{d}$ of $\mathcal{F}$ can be characterized as follows:*

- $\check{d} = \mathsf{inf}\{d \in \mathcal{D} | \mathcal{F}(d) = d\}$
- $\hat{d} = \mathsf{sup}\{d \in \mathcal{D} | \mathcal{F}(d) = d\}$

Tarski and Knaster provided an iteration for computing least and greatest fi xpoints [122, 79]. In particular, it is possible to show that

$\check{d}\ \mathcal{Z}[\mathcal{F}(\mathcal{Z})] = \cup_i \mathcal{F}^i(False)$ and
$\hat{d}\ \mathcal{Z}[\mathcal{F}(\mathcal{Z})] = \cap_i \mathcal{F}^i(True)$

Both fi xpoints can be computed by iteration, where

for the least fi xpoint$\check{d}$ holds $\mathcal{Z}^0 = False$ and $\mathcal{Z}^{i+1} = \mathcal{Z}^i \cup \mathcal{F}(\mathcal{Z}^i)$, and
for the greatest fi xpoint$\hat{d}$ holds $\mathcal{Z}^0 = True$ and $\mathcal{Z}^{i+1} = \mathcal{Z}^i \cap \mathcal{F}(\mathcal{Z}^i)$

The fi xpoints are found when $\mathcal{Z}^i = \mathcal{Z}^{i+1}$. Since the number of states in a Kripke structure is fi nite, termination is guaranteed, because there can be no infi nite sequence of $\mathcal{Z}^i$'s such that $\mathcal{Z}^i \neq \mathcal{Z}^{i+1}$. In a Kripke structure $\mathcal{K}$, a CTL formula φ can be represented by the set of states $\{s | (\mathcal{K}, s) \models \varphi\} \in \mathcal{D}(\mathcal{S})$, that satisfi es φ. Then, the algorithms for the basic CTL operators $\mathsf{E}[\varphi\ \underline{\mathsf{U}}\ \psi]$ and $\mathsf{EG}\ \varphi$ are given by means of fi xpoint computations as follows:

- The set of states where $\mathsf{E}[\varphi\ \underline{\mathsf{U}}\ \psi]$ is true can be determined by computing least fixpoints: $\mathsf{E}[\varphi\ \underline{\mathsf{U}}\ \psi] = \check{d}\ \mathcal{Z}[\psi \vee (\varphi \wedge \mathsf{EX}\ \mathcal{Z})]$

  It was shown (s. [34]) that $\mathsf{E}[\varphi\ \underline{\mathsf{U}}\ \psi]$ is the least fixpoint for $\mathcal{F}(\mathcal{Z}) = \psi \vee (\varphi \wedge \mathsf{EX}\ \mathcal{Z})$. Hence, we have

  $\mathsf{E}[\varphi\ \underline{\mathsf{U}}\ \psi] = \psi \vee (\varphi \wedge \mathsf{EX}\ \mathsf{E}[\varphi\ \underline{\mathsf{U}}\ \psi])$

- The set of states where $\mathsf{EG}\ \varphi$ is true can be determined by computing greatest fixpoints: $\mathsf{EG}\ \varphi = \hat{d}\ \mathcal{Z}[\varphi \wedge \mathsf{EX}\ \mathcal{Z}]$

  It was shown (s. [34]) that $\mathsf{EG}\ \varphi$ is the greatest fixpoint for $\mathcal{F}(\mathcal{Z}) = \varphi \wedge \mathsf{EX}\ \mathcal{Z}$. Hence, we have

  $\mathsf{EG}\ \varphi = \varphi \wedge \mathsf{EX}\ \mathsf{EG}\ \varphi$

Finally, for the basic operator $\mathsf{EX}\ \varphi$, the model checking algorithm must guarantee that

$$(\mathcal{K}, s) \models \mathsf{EX}\ \varphi \Leftrightarrow \exists s'.(\varphi(\!|s'|\!) \wedge (s, s') \in \mathcal{U})$$

The relational product $\exists s'.(\varphi(\!|s'|\!) \wedge (s, s') \in \mathcal{U})$ can be computed using basic operations for set manipulation [20]. It is a fundamental operation for symbolic model checking approaches, since it is involved in all basic CTL operators and appears very frequently. This makes its efficient computation of essential importance. Efficient techniques for this purpose are given in [19] and are implemented in many BDD packages like [120].

2.2 The Synchronous Language QUARTZ

Quartz [112, 114, 111, 116] is a variant of the synchronous language Esterel [10, 54] that differs from Esterel in some minor points. The semantics of Quartz has been defined in [112] and a hardware synthesis for its compilation has been presented in [111]. An extension of the latter including schizophrenia problems has been given in [116]. A complete reference is given in [114]. In the following, we briefly describe the basics of Quartz and Esterel; for more details on Quartz the reader is referred to [112, 114, 111, 116], for more details on Esterel, the reader should consult the Esterel primer [10], which is an excellent introduction to synchronous programming.

Beneath the comfortable programmer's model given by the macro step abstraction, synchronous languages like Quartz provide a rich set of statements for manipulating the execution of concurrent threads. In particular, there are several statements for *preemption* and *suspension*, and different forms of *concurrency like synchronous, asynchronous or interleaved execution*.

In synchronous languages, time is modeled by the natural numbers $\mathbb{N}$, so that the semantics of an expression is a function of type $\mathbb{N} \to \alpha$ for some type α. Quartz distinguishes between two kinds of variables, namely *event variables* and *state variables*. The semantics of an event variable is a function of type $\mathbb{N} \to \mathbb{B}$, while the semantics of a state variable may have the more general type $\mathbb{N} \to \alpha$. The main difference is however the data flow: the value of a state variable y is 'sticky', i.e. if no data operation has been applied to y, then its value does not change. On the other hand, the value of an event variable x is not stored: at the next step, the value of x would be reset to 0 (we denote Boolean values as 1 and 0), if it is not explicitly made 1 at the considered point of time. Hence, the value of an event variable x is 1 at a point of time if and only if there is at least one thread that emits x at this point of time.

Event variables are made present with the **emit** statement, while state variables are manipulated with assignments. Of course, any event or state variable may also be an input variable, so that their values are determined by the environment only. Emissions and assignments are all data manipulating statements. The execution of these statements, as well as the execution of most other statements does not consume time (in the programmer's view). A complete list of all basic statements of Quartz is given in Fig. 2.3, where S, S_1, and S_2 are also basic statements of Quartz, ℓ is a location variable, x is an event variable, y is a state variable, and σ is a Boolean expression:
In general, a statement S may be started at a certain point of time t_1, and may terminate at time $t_2 \geq t_1$, but it may also never terminate. If S immediately terminates when it is started ($t_2 = t_1$), it is called *instantaneous*, otherwise we say that the execution of S takes time, or simply that S *consumes time*. Whether a statement is instantaneous or not may depend on input or local variables. There is only one basic statement that consumes time, namely the **pause** statement. In other words, the **pause** statements are the only statements where the control flow may rest. For this reason, we endow **pause** statements with unique location variables ℓ. These labels are used in [112, 114, 111, 116] as state variables to encode the control flow automaton.

The semantics of Quartz and Esterel can be defined in several ways that lead all to the same transition system. In particular, there is a semantics based on process-algebraic transition rules, and a direct translation into hardware circuits [9]. A detailed explanation of the semantics of Quartz is given in [112, 114, 116]. The control flow of a statement S has been defined by the control flow predicates [112, 114] inside (S), instant (S), enter (S), terminate (S), and move (S), and the data flow of S has been defined by the set of guarded commands guardcmd (φ, S):

inside (S) is the disjunction of the **pause** labels occurring in S. Therefore, inside (S) holds at some point of time iff at this point of time, the control flow is at some location inside S.

instant (S) holds iff the control flow can not stay in S when S would now be started. This means that the execution of S would be instantaneous at this point of time.

- **nothing** (empty statement)
- **emit** x and **emit delayed** x (emissions)
- $y := \tau$ and $y :=$ **delayed** τ (assignments)
- ℓ : **pause** (consumption of time)
- **if** σ **then** S_1 **else** S_2 **end** (conditional)
- $S_1; S_2$ (sequential composition)
- $S_1 \parallel S_2$ (synchronous parallel composition)
- $S_1 \mid\mid\mid S_2$ (asynchronous parallel composition)
- **choose** S_1 $[\!]$ S_2 **end** (nondeterministic choice)
- **do** S **while** σ (iteration)
- **suspend** S **when** σ (suspension)
- **weak suspend** S **when** σ (weak suspension)
- **abort** S **when** σ (abortion)
- **weak abort** S **when** σ (weak abortion)
- **local** x **in** S **end** (local event variable)
- **local** $y : \alpha$ **in** S **end** (local state variable)
- **now** σ (instantaneous assertion)
- **during** S **holds** σ (invariant assertion)

Figure 2.3: Basic Syntax of Quartz

enter (S) describes where the control flow will be at the next point of time, when S would now be started.

terminate (S) describes all conditions where the control flow is currently somewhere inside S and wants to leave S. Note however, that the control flow might still be in S at the next point of time since S may be entered at the same time, for example, by a surrounding loop statement.

move (S) describes all internal moves, i.e., all possible transitions from somewhere inside S to another location inside S.

guardcmd (φ, S) is a set of pairs of the form $(\gamma, \mathcal{C})$, where $\mathcal{C}$ is a data manipulating statement, i.e., either an emission or an assignment. The meaning of $(\gamma, \mathcal{C})$ is that $\mathcal{C}$ is immediately executed whenever the guard γ holds.

Using the above control flow predicates, one can define a finite-state transition system that defines the control flow of a statement [112, 114].

The data flow is determined by the guarded commands guardcmd (φ, S) that appear as conditional emissions and assignments on the transitions of the control flow transition system. In case that only finite data types were used, it is possible to translate a program to a classical finite state (Mealy) automaton, as shown in [112]. There, a given synchronous program is translated into a finite-state automaton (representing the control flow) whose transitions are labeled with a set of conditional assignments (representing the data flow). Each transition directly corresponds to a macro step of the program. This description is given in an implicit form, so that it can be used for symbolic state space exploration [11, 22].

For example, for the body statement of module *RussMult* given in Figure 6.5 of section 6.2, we obtain the following results ($\mathsf{X}\varphi$ means that φ holds at the next point of time):

- $\mathsf{inside}\,(S) \equiv \ell \vee rdy$
- $\mathsf{instant}\,(S) \equiv 0$
- $\mathsf{enter}\,(S) \equiv \mathsf{X}rdy$
- $\mathsf{terminate}\,(S) \equiv 0$
- $\mathsf{move}\,(S) \equiv \left(\begin{array}{l} rdy \wedge \mathsf{X}rdy \wedge (\neg req \vee (y=0))\vee \\ rdy \wedge \mathsf{X}\ell \wedge req \wedge (y \neq 0)\vee \\ \ell \wedge \mathsf{X}\ell \wedge (y \neq 0)\vee \\ \ell \wedge \mathsf{X}rdy \wedge (y=0) \end{array}\right)$

Based on the presented basic statements, one can define a couple of several macro statements whose semantics is then simply given by the macro expansion. The most popular ones (most of them are used by the Esterel language) are given in Fig. 2.4.

- **while** σ **do** S **end** $:\equiv \left(\begin{array}{l} \textbf{if } \sigma \textbf{ then} \\ \quad \textbf{do } S \textbf{ while } \sigma \\ \textbf{else nothing end} \end{array}\right)$
- ℓ : **halt** :≡ **do** ℓ : **pause while** 1
- **loop** S **end** :≡ **while** 1 **do** S **end**
- ℓ : **loop** S **each** σ
  :≡ **loop abort** $S; \ell$: **halt when** σ **end**
- ℓ_0 : **every** σ ℓ_1 : **do** S **end**
  :≡ ℓ_0 : **await** $\sigma; \ell_1$: **loop** S **each** σ
- ℓ : **sustain** x :≡ **do emit** $x; \ell$: **pause while** 1
- ℓ : **await** σ :≡ **do** ℓ : **pause while** $\neg\sigma$
  :≡ **abort** ℓ : **halt when** σ
- ℓ : **await immediate** σ
  :≡ **while** $\neg\sigma$ **do** ℓ : **pause end**
  :≡ **abort** ℓ : **halt when immediate** σ

Figure 2.4: Popular Macro Statements of Quartz

Chapter 3

The Real-Time Temporal Logic JCTL

Τό φύσει εἶναι κρείττω τἀληϑῆ καί δίκαια τῶν ἐναντίων. [1]
ΑΡΙΣΤΟΤΕΛΗΣ (384 - 332 π.Χ.)
(Ρητ. Τέχνη, 1355α, 22-23)

In the next sections, we introduce timed transition systems based on interpretation I_J (cf. 1.1.4) and explain more in detail the meaning of this interpretation, which allows abstraction of real-time systems without loss of quantitative information. Then we define a new real-time temporal logic JCTL, which is defined on such transition systems. Having given the syntax and semantics of JCTL, we then show how different operators can be expressed as abbreviations of the basic JCTL operators.

3.1 Timed Kripke Structures and Abstractions

To model real-time systems we introduce in this section *timed Kripke structures (TKS)* over some set of variables $\mathcal{V}$. Timed Kripke structures are formally defined as follows:

Definition 7 (Timed Kripke Structure (TKS)) *A timed Kripke structure over the variables $\mathcal{V}$ is a tuple $(\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$, such that $\mathcal{S}$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states, and $\mathcal{R} \subseteq \mathcal{S} \times \mathbb{N} \times \mathcal{S}$ is the set of transitions. For any state $s \in \mathcal{S}$, the set $\mathcal{L}(s) \subseteq \mathcal{V}$ is the set of variables that hold on s. We furthermore demand that for any $(s, t, s') \in \mathcal{R}$, we have $t > 0$ and that for any $s \in \mathcal{S}$, there must be a $t \in \mathbb{N}$ and a $s' \in \mathcal{S}$ such that $(s, t, s') \in \mathcal{R}$ holds.*

Timed Kripke structures may be pictorially drawn as given in Figure 3.1, where initial states are drawn with double lines. Some approaches, e.g. [27] label their transitions with intervals $[a, b]$ of time. It is easily seen that TKSs subsume these models since we can add for any $t \in [a, b]$ a new transition between the considered two states.

[1]True and right things are by nature stronger than their opposites.
ARISTOTELES (384 - 332 BC)

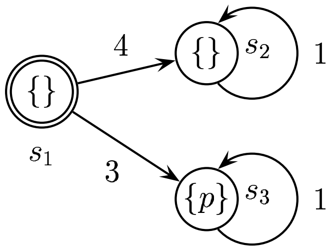


Figure 3.1: A Timed Kripke Structure

It is crucial to understand what is modeled by a TKS. We use interpretation I_J: *A transition from state s to state s' with label $k \in \mathbb{N}$ means that at any time t_0, where we are in state s, we can perform an atomic action that requires k units of time. The action terminates at time $t_0 + k$, where we are in state s'. In particular, there is no information about the intermediate points of time t with $t_0 < t < t_0 + k$.*

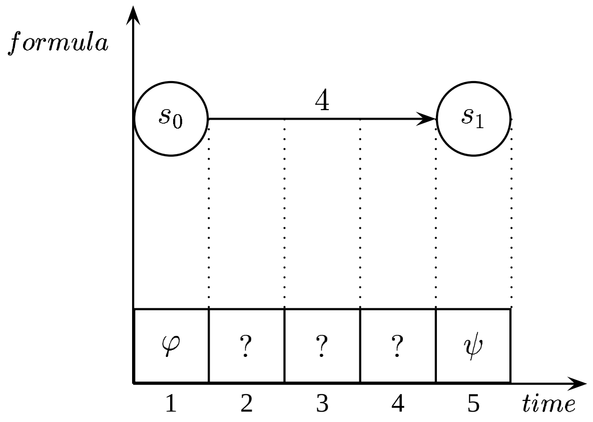


Figure 3.2: A Timed Transition in a TKS

As example, consider Figure 3.2, which shows a timed transition in a TKS. At the point of time $t = 1$ we are in state s_0 where the formula φ is valid. The transition from s_0 to s_1 requires 4 time units. At the point of time $t = 5$, state s_1 is reached, where the formula ψ is valid. No information is given about the validity of a formula for the points of time $t = 2, t = 3, t = 4$, between the states s_0 and s_1.

It's easy to see that normal Kripke structures are special cases of TKS that are obtained by restricting TKSs so that $(s, t, s') \in \mathcal{R}$ implies $t = 1$. To avoid confusion, we call the 'normal Kripke structures' *unit delay structures (UDS)*:

Definition 8 (Unit Delay Structure (UDS)) *A TKS $\mathcal{K}$ = ($\mathcal{I}$, $\mathcal{S}$, $\mathcal{R}$, $\mathcal{L}$) is called to be unit delay structure (UDS) iff $\mathcal{X}_\mathcal{K} := \{t \mid (s, t, s') \in \mathcal{R}\} = \{1\}$.*

Finally, note that it is also possible to consider certain infi nite sets of transitions in a TKS; we will see this in more detail in section 6.1.3. Roughly speaking, we could allow labels with linear constraints, as e.g. $\{2n + 3m + 5 \mid n, m \in \mathbb{N}\}$ or $\{n \in \mathbb{N} \mid n > 10\}$. The reason is that these labeled transitions can be replaced by fi nitely many states including some cycles.

3.2 JCTL as a Real-Time Extension of CTL

In this section, we introduce the *first real-time extension* JCTL *of* CTL*, that is based on interpreting timed transition systems with interpretation I_J* so that the logic directly allows the abstraction of real-time systems by ignoring their irrelevant qualitative properties, but without loosing their quantitative ones. For example, we can model processes that compute some values within a certain limit of time with a single transition, that does not state anything about the values of the variables *during the computation*. JCTL has a next-state operator equipped with time bounds, which make the logic powerful enough to reason about real-time constraints of *atomic timed actions*. Note that the existing approaches do not allow this.

3.2.1 Syntax and Semantics of JCTL

For the JCTL defi nition below, only a small subset of basic logical operators is required, that will be extended further by abbreviations.

Definition 9 (Syntax of JCTL) *Given a set of variables $\mathcal{V}$, the set of* JCTL *formulae is the least set satisfying the following rules, where φ and ψ denote arbitrary* JCTL *formulae, and $a, b \in \mathbb{N}$ are arbitrary natural numbers:*

- $\mathcal{V} \subseteq$ JCTL*, i.e, any variable is a* JCTL *formula*
- $\neg\varphi$, $\varphi \wedge \psi \in$ JCTL
- $\mathsf{E}\underline{\mathsf{X}}^{[a+1,b]}\varphi \in$ JCTL
- $\mathsf{E}\underline{\mathsf{X}}^{\geq a+1}\varphi \in$ JCTL
- $\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[a,b]}\ \psi] \in$ JCTL
- $\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\geq a}\ \psi] \in$ JCTL
- $\mathsf{EG}^{[a,b]}\varphi \in$ JCTL
- $\mathsf{EG}^{\geq a}\varphi \in$ JCTL

Note that, in addition to the strong- and weak until operators, JCTL is also equipped with *strong- and weak next operators*.

Also note that, as we will show in section 3.2.3, the basic operators $\mathsf{EG}^{[0,b]}$ and $\mathsf{EG}^{\geq a}$ can be defi ned as abbreviations of JCTL and CTL operators. Nevertheless, in order to simplify formulae notions, but also to be able to directly compare JCTL with other existing real-time logics (cf. section 4), we use $\mathsf{EG}^{[0,b]}$ and $\mathsf{EG}^{\geq a}$ as reference.

The semantics of JCTL is defi ned with respect to a TKS. For the defi nition of the semantics, we need the notion of paths in timed Kripke structures. Formally, the notion of a path in a TKS is defi ned as follows:

Definition 10 (Path in a Timed Kripke Structure) *A path π through a timed Kripke structure is a function $\pi : \mathbb{N} \to \mathcal{S}$ such that $\forall i \in \mathbb{N}.\exists t \in \mathbb{N}.(\pi^{(i)}, t, \pi^{(i+1)}) \in \mathcal{R}$ holds (we write the function application with a superscript). Hence, $\pi^{(i)}$ is the $(i+1)th$ state on path π. For a given path π, we define an* associated time consumption function *τ_π, so that π and τ_π satisfy the condition $\forall i \in \mathbb{N}.(\pi^{(i)}, \tau_\pi^{(i)}, \pi^{(i+1)}) \in \mathcal{R}$.*

Note that τ_π is not uniquely defi ned for a fi xed path π, since we may have more than one transition between two states that are labeled with different numbers. The set of paths starting in a state s is furthermore denoted as $\mathsf{Paths}_\mathcal{K}(s)$.

Definition 11 (Semantics of JCTL**)** *Given a TKS $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$, and $s \in \mathcal{S}$, then the semantics of the logic is recursively defined as follows:*

- $(\mathcal{K}, s) \models p$ *iff* $p \in \mathcal{L}(s)$ *for any* $p \in \mathcal{V}$
- $(\mathcal{K}, s) \models \neg\varphi$ *iff* $(\mathcal{K}, s) \not\models \varphi$
- $(\mathcal{K}, s) \models \varphi \wedge \psi$ *iff* $(\mathcal{K}, s) \models \varphi$ *and* $(\mathcal{K}, s) \models \psi$
- $(\mathcal{K}, s) \models \mathsf{E}\underline{\mathsf{X}}^{[a+1,b]}\varphi$ *iff there is a path $\pi \in \mathsf{Paths}_\mathcal{K}(s)$ with associated duration function τ_π with*
$$\left(a+1 \le \tau_\pi^{(0)} \le b\right) \wedge \left((\mathcal{K}, \pi^{(1)}) \models \varphi\right)$$
- $(\mathcal{K}, s) \models \mathsf{E}\underline{\mathsf{X}}^{\ge a+1}\varphi$ *iff there is a path $\pi \in \mathsf{Paths}_\mathcal{K}(s)$ with associated duration function τ_π with*
$$\left(a+1 \le \tau_\pi^{(0)}\right) \wedge \left((\mathcal{K}, \pi^{(1)}) \models \varphi\right)$$
- $(\mathcal{K}, s) \models \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[a,b]}\ \psi]$ *iff there is a path $\pi \in \mathsf{Paths}_\mathcal{K}(s)$ with associated duration function τ_π and an $i \in \mathbb{N}$ with*
$$\left(a \le \sum_{j=0}^{i-1} \tau_\pi^{(j)} \le b\right) \wedge \left((\mathcal{K}, \pi^{(i)}) \models \psi\right) \wedge \left(\forall j < i.\ (\mathcal{K}, \pi^{(j)}) \models \varphi\right)$$
- $(\mathcal{K}, s) \models \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\ge a}\ \psi]$ *iff there is a path $\pi \in \mathsf{Paths}_\mathcal{K}(s)$ with associated duration function τ_π and an $i \in \mathbb{N}$ with*
$$\left(a \le \sum_{j=0}^{i-1} \tau_\pi^{(j)}\right) \wedge \left((\mathcal{K}, \pi^{(i)}) \models \psi\right) \wedge \left(\forall j < i.\ (\mathcal{K}, \pi^{(j)}) \models \varphi\right)$$
- $(\mathcal{K}, s) \models \mathsf{EG}^{[a,b]}\varphi$ *iff there is a path $\pi \in \mathsf{Paths}_\mathcal{K}(s)$ with associated duration function τ_π, such that for all $i \in \mathbb{N}$, we have*
$$\left(a \le \sum_{j=0}^{i-1} \tau_\pi^{(j)} \le b\right) \to \left((\mathcal{K}, \pi^{(i)}) \models \varphi\right)$$

- $(\mathcal{K}, s) \models \mathsf{EG}^{\geq a}\varphi$ *iff there is a path* $\pi \in \mathsf{Paths}_{\mathcal{K}}(s)$ *with associated duration function* τ_π*, such that for all* $i \in \mathbb{N}$*, we have*

$$\left(a \leq \sum_{j=0}^{i-1} \tau_\pi^{(j)}\right) \rightarrow \left((\mathcal{K}, \pi^{(i)}) \models \varphi\right)$$

Given a TKS $\mathcal{K}$ *and a* JCTL *formula* φ*, we denote the set of states of* $\mathcal{K}$ *where* φ *holds as* $[\![\varphi]\!]_{\mathcal{K}}$.

Intuitively, the semantics of JCTL can be explained as follows:

- $(\mathcal{K}, s) \models \mathsf{E}\underline{\mathsf{X}}^{[a+1,b]}\varphi$ means that the state s has a direct successor state s' that satisfi es φ and can be reached in time $t \in [a+1, b]$.
- $(\mathcal{K}, s) \models \mathsf{E}\underline{\mathsf{X}}^{\geq a+1}\varphi$ means that the state s has a direct successor state s' that satisfi es φ and can be reached in time $t \geq a+1$.
- $(\mathcal{K}, s) \models \mathsf{E}[\varphi\, \underline{\mathsf{U}}^{[a,b]}\, \psi]$ means that there is a path π starting in $\pi^{(0)} = s$ and a number $i \in \mathbb{N}$ so that for the fi rst i states $\pi^{(0)}, \pi^{(1)}, \ldots, \pi^{(i-1)}$ the property φ holds, and ψ holds on $\pi^{(i)}$, and the time $t := \sum_{j=0}^{i-1} \tau_\pi^{(j)}$ *required to reach* state $\pi^{(i)}$ satisfi es the numerical relations $a \leq t$ and $t \leq b$.
- $(\mathcal{K}, s) \models \mathsf{E}[\varphi\, \underline{\mathsf{U}}^{\geq a}\, \psi]$ means that there is a path π starting in $\pi^{(0)} = s$ and a number $i \in \mathbb{N}$ so that for the fi rst i states $\pi^{(0)}, \pi^{(1)}, \ldots, \pi^{(i-1)}$ the property φ holds, and ψ holds on $\pi^{(i)}$, and the time $t := \sum_{j=0}^{i-1} \tau_\pi^{(j)}$ *required to reach* state $\pi^{(i)}$ satisfi es the numerical relation $a \leq t$.
- $(\mathcal{K}, s) \models \mathsf{EG}^{[a,b]}\varphi$ means that there is a path π starting in $\pi^{(0)} = s$, such that for any state $\pi^{(i)}$ that is reached within a time $t := \sum_{j=0}^{i-1} \tau_\pi^{(j)}$ with $t \in [a, b]$, we have $(\mathcal{K}, \pi^{(i)}) \models \varphi$. Hence, φ holds in the interval $[a, b]$.
- $(\mathcal{K}, s) \models \mathsf{EG}^{\geq a}\varphi$ means that there is a path π starting in $\pi^{(0)} = s$, such that for any state $\pi^{(i)}$ that is reached within a time $t := \sum_{j=0}^{i-1} \tau_\pi^{(j)}$ with $t \geq a$, we have $(\mathcal{K}, \pi^{(i)}) \models \varphi$. Hence, φ holds for all states on π that are reached at time a or after time a.

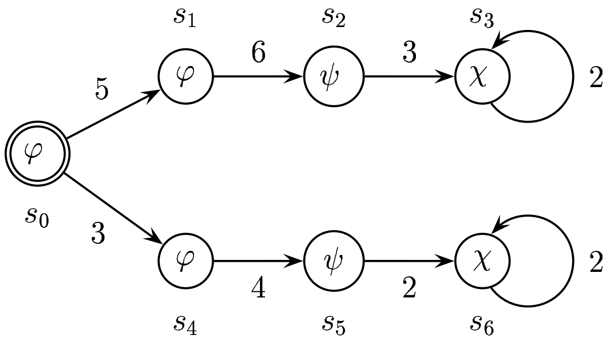


Figure 3.3: An Example for the JCTL Semantics

As example for the JCTL semantics, consider the timed Kripke structure shown in Figure 3.3. There, we have:

- $\llbracket \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\leq 9}\ \psi] \rrbracket = \{s_0, s_1, s_2, s_4, s_5\}$
- $\llbracket \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\leq 6}\ \psi] \rrbracket = \{s_1, s_2, s_4, s_5\}$
- $\left\llbracket \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[3,5]}\ \psi] \right\rrbracket = \{s_4\}$
- $\llbracket \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\geq 9}\ \psi] \rrbracket = \{s_0\}$
- $\llbracket \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\geq 12}\ \psi] \rrbracket = \{\}$
- $\llbracket \mathsf{EG}^{\leq 4}\varphi \rrbracket = \{s_0, s_1\}$
- $\left\llbracket \mathsf{EG}^{[2,4]}\varphi \right\rrbracket = \{s_0, s_1\}$
- $\llbracket \mathsf{EG}^{\geq 4}\varphi \rrbracket = \{\}$
- $\llbracket \mathsf{E}\underline{\mathsf{X}}^{\leq 7}\varphi \rrbracket = \{s_0\}$
- $\llbracket \mathsf{E}\underline{\mathsf{X}}^{\leq 4}\varphi \rrbracket = \{s_0\}$
- $\left\llbracket \mathsf{E}\underline{\mathsf{X}}^{[4,8]}\varphi \right\rrbracket = \{s_0\}$
- $\llbracket \mathsf{E}\underline{\mathsf{X}}^{\geq 2}\varphi \rrbracket = \{s_0\}$
- $\llbracket \mathsf{E}\underline{\mathsf{X}}^{\geq 6}\varphi \rrbracket = \{\}$

Note that in the results of $\left\llbracket \mathsf{EG}^{[2,4]}\varphi \right\rrbracket$ the state s_1 holds trivially, since, starting at s_1, there is no state at all to be found in the interval $[2, 4]$. Hence, due to the defi nition of $\left\llbracket \mathsf{EG}^{[a,b]}\varphi \right\rrbracket$, we have

$$false \rightarrow \left((\mathcal{K}, \pi^{(i)}) \models \varphi\right)$$

which is always true.

The states in $\left\llbracket \mathsf{EG}^{[a,b]}\varphi \right\rrbracket$ that do not only trivially hold, can be easily determined by the following conjunction:

$$\left\llbracket \mathsf{EG}^{[a,b]}\varphi \right\rrbracket \wedge \left\llbracket \mathsf{E}[1\ \underline{\mathsf{U}}^{[a,b]}\ \varphi] \right\rrbracket$$

3.2.2 JCTL Operators as Abbreviations

In this section we show that the set of the basic JCTL operators is complete for a CTL-like logic. In particular, we show that *further* JCTL *operators, like e.g. the formulae* $\mathsf{A}\underline{\mathsf{X}}^{\kappa}\varphi$, $\mathsf{A}[\varphi\ \underline{\mathsf{U}}^{\kappa}\ \psi]$, *and* $\mathsf{AG}^{\kappa}\varphi$ *can be defined in terms of* $\mathsf{E}\underline{\mathsf{X}}^{\kappa}\varphi$, $\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\kappa}\ \psi]$, *and* $\mathsf{EG}^{\kappa}\varphi$. It is well-known that all CTL formulae can be rewritten so that only the operators EX, $\mathsf{E}[\cdot\ \underline{\mathsf{U}}\ \cdot]$, and EG are used with the following equations (cf. defi nition 5 in section 2.1.3):

- $\mathsf{AX}\varphi :\equiv \neg\mathsf{EX}\neg\varphi$
- $\mathsf{AG}\varphi :\equiv \neg\mathsf{EF}\neg\varphi \equiv \neg\mathsf{E}[1\ \underline{\mathsf{U}}\ (\neg\varphi)]$
- $\mathsf{A}[\varphi\ \underline{\mathsf{U}}\ \psi] :\equiv \neg\mathsf{E}[(\neg\psi)\ \underline{\mathsf{U}}\ \neg(\varphi \vee \psi)] \wedge \neg\mathsf{EG}\neg\psi$

The question is now whether such definitions can be used for real-time extensions as well. In order to prove JCTL equations, we use an abstraction of the time constraint to an ordinary signal and a subsequent LTL theorem proving [2], i.e. considering the validity of the formula on a single path (without path quantifiers). In particular, we embed the time constraint κ into the boolean formulae, which allows us to consider time constraints at boolean level, as parts of boolean formulae. For this purpose, we define a complete set of real-time temporal operators according to JCTL semantics as follows:

Definition 12 (JCTL Macro Operators) *We define:*

- $\underline{\mathsf{X}}^{\kappa}\varphi :\equiv \mathsf{X}[\kappa \wedge \varphi]$
- $\mathsf{X}^{\kappa}\varphi :\equiv \mathsf{X}[\kappa \rightarrow \varphi]$
- $\mathsf{G}^{\kappa}\varphi :\equiv \mathsf{G}[\kappa \rightarrow \varphi]$
- $\mathsf{F}^{\kappa}\varphi :\equiv \mathsf{F}[\kappa \wedge \varphi]$
- $[\varphi\ \underline{\mathsf{U}}^{\kappa}\ \psi] :\equiv [\varphi\ \underline{\mathsf{U}}\ (\kappa \wedge \psi)]$
- $[\varphi\ {}^{\kappa}\underline{\mathsf{U}}\ \psi] :\equiv [(\kappa \rightarrow \varphi)\ \underline{\mathsf{U}}\ \psi]$
- $[\varphi\ \mathsf{U}^{\kappa}\ \psi] :\equiv [\varphi\ \mathsf{U}\ (\kappa \wedge \psi)]$
- $[\varphi\ {}^{\kappa}\mathsf{U}\ \psi] :\equiv [(\kappa \rightarrow \varphi)\ \mathsf{U}\ \psi]$
- $[\varphi\ \underline{\mathsf{B}}^{\kappa}\ \psi] :\equiv [\varphi\ \underline{\mathsf{B}}\ (\kappa \wedge \psi)]$
- $[\varphi\ {}^{\kappa}\underline{\mathsf{B}}\ \psi] :\equiv [(\kappa \wedge \varphi)\ \underline{\mathsf{B}}\ \psi]$
- $[\varphi\ \mathsf{B}^{\kappa}\ \psi] :\equiv [\varphi\ \mathsf{B}\ (\kappa \wedge \psi)]$
- $[\varphi\ {}^{\kappa}\mathsf{B}\ \psi] :\equiv [(\kappa \wedge \varphi)\ \mathsf{B}\ \psi]$

It is not hard to see that the CTL equation is not true for $\mathsf{A}\underline{\mathsf{X}}^{\kappa}$, since the equation $\underline{\mathsf{X}}^{\kappa}\varphi = \neg\underline{\mathsf{X}}^{\kappa}\neg\varphi$ is not valid. For this reason, we have to define another real-time next-operator as follows: $\mathsf{X}^{\kappa}\varphi :\equiv \mathsf{X}[\kappa \rightarrow \varphi]$. We therefore have to distinguish between a weak and strong real-time next-operator (obviously, we have $\underline{\mathsf{X}}^{\kappa}\varphi \rightarrow \mathsf{X}^{\kappa}\varphi$). Then, we have the negation laws $\underline{\mathsf{X}}^{\kappa}\varphi = \neg\mathsf{X}^{\kappa}\neg\varphi$ and $\mathsf{X}^{\kappa}\varphi = \neg\underline{\mathsf{X}}^{\kappa}\neg\varphi$, and therefore also the following definitions:

- $\mathsf{EX}^{\kappa}\varphi :\equiv \mathsf{EX}(\kappa \rightarrow \varphi) \equiv \mathsf{EX}\neg\kappa \vee \mathsf{EX}\varphi \equiv \mathsf{E}\underline{\mathsf{X}}^{\overline{\kappa}}1 \vee \mathsf{E}\underline{\mathsf{X}}^{\geq 0}\varphi$
- $\mathsf{A}\underline{\mathsf{X}}^{\kappa}\varphi :\equiv \neg\mathsf{EX}^{\kappa}\neg\varphi$
- $\mathsf{AX}^{\kappa}\varphi :\equiv \neg\mathsf{E}\underline{\mathsf{X}}^{\kappa}\neg\varphi$

[2]see [113] for more details

Hence, all the real-time next-operators can be defined by $\mathsf{E}\underline{\mathsf{X}}^{\kappa}$, but we have to use more complicated definitions, and we must have complementary time constraints (this means that the set of time constraints must be closed under complement). The definition of $\mathsf{AG}^{\kappa}\varphi$ (and $\mathsf{EF}^{\kappa}\varphi$), on the other hand, is straightforwardly given by the following equations:

- $\mathsf{AG}^{\kappa}\varphi :\equiv \mathsf{AG}(\kappa \to \varphi) \equiv \neg\mathsf{EF}(\kappa \wedge \neg\varphi) \equiv \neg\mathsf{EF}^{\kappa}\neg\varphi$
- $\mathsf{EF}^{\kappa}\varphi :\equiv \mathsf{EF}(\kappa \wedge \varphi) \equiv \mathsf{E}[1\ \underline{\mathsf{U}}\ (\kappa \wedge \varphi)] \equiv \mathsf{E}[1\ \underline{\mathsf{U}}^{\kappa}\ \varphi]$

Finally, consider $\mathsf{A}[\varphi\ \underline{\mathsf{U}}^{\kappa}\ \psi]$. The equation that is the basis for the definition used in CTL is the LTL theorem $[\varphi\ \underline{\mathsf{U}}\ \psi] = \neg\,[(\neg\psi)\ \underline{\mathsf{U}}\ \neg(\varphi \vee \psi)] \wedge \neg\mathsf{G}\neg\psi$, which may be rewritten to the more readable form $[\varphi\ \underline{\mathsf{U}}\ \psi] = [\psi\ \mathsf{B}\ \neg(\varphi \vee \psi)] \wedge \mathsf{F}\psi$, and once more to $[\varphi\ \underline{\mathsf{U}}\ \psi] = [\psi\ \underline{\mathsf{B}}\ \neg(\varphi \vee \psi)]$. Consider now what happens, when we want to use the same transformations for real-time operators, too. To see this, consider the following transformations of the right hand side:

$$\begin{array}{rl} & \neg\mathsf{E}[(\neg\psi)\ \underline{\mathsf{U}}^{\kappa}\ \neg(\varphi \vee \psi)] \wedge \neg\mathsf{EG}^{\kappa}\neg\psi \\ \equiv & \neg\mathsf{E}[(\neg\psi)\ \underline{\mathsf{U}}\ (\kappa \wedge \neg(\varphi \vee \psi)] \wedge \neg\mathsf{EG}(\kappa \to \neg\psi) \\ \equiv & \mathsf{A}[\psi\ \mathsf{B}\ (\kappa \wedge \neg(\varphi \vee \psi)] \wedge \mathsf{AF}(\kappa \wedge \psi) \\ \equiv & \mathsf{A}[\psi\ \mathsf{B}\ (\kappa \wedge \neg\varphi \wedge \neg\psi)] \wedge \mathsf{AF}^{\kappa}\psi \\ \equiv & \mathsf{A}[(\kappa \to \varphi)\ \mathsf{U}\ \psi] \wedge \mathsf{AF}^{\kappa}\psi \end{array}$$

It is not difficult to see that this is not equivalent to $\mathsf{A}[\varphi\ \underline{\mathsf{U}}^{\kappa}\ \psi]$, i.e. the equation

$$\mathsf{A}[\varphi\ \underline{\mathsf{U}}^{\kappa}\ \psi] \equiv \neg\mathsf{E}[(\neg\psi)\ \underline{\mathsf{U}}^{\kappa}\ \neg(\varphi \vee \psi)] \wedge \neg\mathsf{EG}^{\kappa}\neg\psi \qquad (1)$$

does not hold.

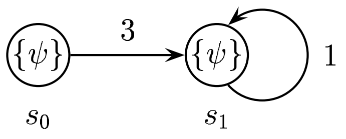


Figure 3.4: Invalidity of the equation (1)

To see this in an example, consider the structure given in Figure 3.4. We have $(\mathcal{K}, s_0) \not\models \mathsf{A}\left[\varphi\ \underline{\mathsf{U}}^{\geq 3}\ \psi\right]$, since φ does not hold up to the state where a time ≥ 3 is consumed, i.e. in state s_0. On the other hand, we have $(\mathcal{K}, s_0) \models \mathsf{AF}^{\geq 3}\psi$, and also $(\mathcal{K}, s_0) \models c.\mathsf{A}[(c \geq 3 \to \varphi)\ \mathsf{U}\ \psi]$.

In the next definition, we systematically define a complete set of real-time temporal operators.

To this end, we consider – if necessary – the different types of time constraints (i.e. $<$, $\leq$, $=$, $\leq$, $>$) individually. Then, further JCTL operators can be expressed as abbreviations:

Definition 13 (Further JCTL Temporal Operators) *Let κ be any time constraint, i.e., $[a, b]$, $\sim k$ with $\sim \in \{<, \leq, =, \leq, >\}$, or the empty constraint. Let $a, b \in \mathbb{N}$ and p be an arbitrary variable. Then, we define further temporal operators in* JCTL *as follows:*

- *Boolean Operators:*
  - $\mathsf{1} := p \vee \neg p$
  - $\mathsf{0} := p \wedge \neg p$
  - $\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$
  - $\varphi \rightarrow \psi := \neg\varphi \vee \psi$
- $\mathsf{E}\underline{\mathsf{X}}$ *Operators:*
  - $\mathsf{E}\underline{\mathsf{X}}\varphi := \mathsf{E}\underline{\mathsf{X}}^{\geq 1}\varphi$
  - $\mathsf{E}\underline{\mathsf{X}}^{\leq a}\varphi := \mathsf{E}\underline{\mathsf{X}}^{[1,a]}\varphi$
  - $\mathsf{E}\underline{\mathsf{X}}^{<a}\varphi := \mathsf{E}\underline{\mathsf{X}}^{[1,a-1]}\varphi$
  - $\mathsf{E}\underline{\mathsf{X}}^{>a}\varphi := \mathsf{E}\underline{\mathsf{X}}^{\geq a+1}\varphi$
  - $\mathsf{E}\underline{\mathsf{X}}^{=a+1}\varphi := \mathsf{E}\underline{\mathsf{X}}^{[a+1,a+1]}\varphi$
  - $\mathsf{E}\underline{\mathsf{X}}^{\neq a}\varphi := \mathsf{E}\underline{\mathsf{X}}^{<a}\varphi \vee \mathsf{E}\underline{\mathsf{X}}^{>a}\varphi$
- $\mathsf{E}[\cdot\ \underline{\mathsf{U}}\ \cdot]$ *Operators:*
  - $\mathsf{E}[\varphi\ \underline{\mathsf{U}}\ \psi] := \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\geq 0}\ \psi]$
  - $\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\leq a}\ \psi] := \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[0,a]}\ \psi]$
  - $\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{<a}\ \psi] := \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[0,a-1]}\ \psi]$
  - $\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{>a}\ \psi] := \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\geq a+1}\ \psi]$
  - $\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{=a}\ \psi] := \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[a,a]}\ \psi]$
- EG *Operators:*
  - $\mathsf{EG}\varphi := \mathsf{EG}^{\geq 0}\varphi$
  - $\mathsf{EG}^{\leq a}\varphi := \mathsf{EG}^{[0,a]}\varphi$
  - $\mathsf{EG}^{<a}\varphi := \mathsf{EG}^{[0,a-1]}\varphi$
  - $\mathsf{EG}^{>a}\varphi := \mathsf{EG}^{\geq a+1}\varphi$
  - $\mathsf{EG}^{=a}\varphi := \mathsf{EG}^{[a,a]}\varphi$

The following equations can be used to define further real-time temporal operators:

- X *Operators:*
  - $\mathsf{EX}^{\kappa}\varphi \equiv \mathsf{E}\underline{\mathsf{X}}^{\overline{\kappa}}1 \vee \mathsf{E}\underline{\mathsf{X}}^{\geq 1}\varphi$
  - $\mathsf{A}\underline{\mathsf{X}}^{\kappa}\varphi :\equiv \neg\mathsf{EX}^{\kappa}\neg\varphi$
  - $\mathsf{AX}^{\kappa}\varphi :\equiv \neg\mathsf{E}\underline{\mathsf{X}}^{\kappa}\neg\varphi$
- F *and* G *Operators:*
  - $\mathsf{EF}^{\kappa}\varphi \equiv \mathsf{E}[1\ \underline{\mathsf{U}}^{\kappa}\ \varphi]$

- $\mathsf{AF}^{\kappa}\varphi \equiv \neg\mathsf{EG}^{\kappa}\neg\varphi$
- $\mathsf{AG}^{\kappa}\varphi \equiv \neg\mathsf{E}[1\ \underline{\mathsf{U}}^{\kappa}\ \neg\varphi]$

- *strong-until ($\underline{\mathsf{U}}$) operators:*
  - $\mathsf{A}\Big[\varphi\ \underline{\mathsf{U}}^{[a,b]}\ \psi\Big] \equiv \mathsf{A}\big[\varphi\ \underline{\mathsf{U}}^{\geq a}\ \psi\big] \wedge \neg\mathsf{EG}^{[a,b]}\neg\psi$
  - $\mathsf{A}\big[\varphi\ \underline{\mathsf{U}}^{\geq a+1}\ \psi\big] \equiv \mathsf{AG}^{\leq a}\,(\varphi \wedge \mathsf{AXA}[\varphi\ \underline{\mathsf{U}}\ \psi])$
  - $\mathsf{A}[\varphi\ \underline{\mathsf{U}}\ \psi] \equiv \neg\mathsf{E}[(\neg\psi)\ \underline{\mathsf{U}}\ \neg(\varphi \vee \psi)] \wedge \neg\mathsf{EG}\neg\psi$

- *weak-until (U) operators:*
  - $\mathsf{E}[\varphi\ \mathsf{U}^{\kappa}\ \psi] \equiv \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\kappa}\ \psi] \vee \mathsf{EG}\varphi$
  - $\mathsf{A}\big[\varphi\ \mathsf{U}^{\geq a+1}\ \psi\big] \equiv \mathsf{AG}^{\leq a}\,(\varphi \wedge \mathsf{AXA}[\varphi\ \mathsf{U}\ \psi])$
  - $\mathsf{A}\big[\varphi\ \mathsf{U}^{\leq b}\ \psi\big] \equiv \mathsf{A}\big[\varphi\ \underline{\mathsf{U}}^{\leq b}\ (\psi \vee \mathsf{AG}\varphi)\big]$
  - $\mathsf{A}[\varphi\ \mathsf{U}\ \psi] \equiv \neg\mathsf{E}[(\neg\psi)\ \underline{\mathsf{U}}\ \neg(\varphi \vee \psi)]$

- *before ($\underline{\mathsf{B}}$, B) operators:*
  - $\mathsf{E}[\varphi\ \mathsf{B}^{\kappa}\ \psi] \equiv \neg\mathsf{A}[(\neg\varphi)\ \underline{\mathsf{U}}^{\kappa}\ \psi]$
  - $\mathsf{E}[\varphi\ \underline{\mathsf{B}}^{\kappa}\ \psi] \equiv \neg\mathsf{A}[(\neg\varphi)\ \mathsf{U}^{\kappa}\ \psi]$
  - $\mathsf{A}[\varphi\ \mathsf{B}^{\kappa}\ \psi] \equiv \neg\mathsf{E}[(\neg\varphi)\ \underline{\mathsf{U}}^{\kappa}\ \psi]$
  - $\mathsf{A}[\varphi\ \underline{\mathsf{B}}^{\kappa}\ \psi] \equiv \neg\mathsf{E}[(\neg\varphi)\ \mathsf{U}^{\kappa}\ \psi]$

- *left-constrained operators:*
  - $\mathsf{E}[\varphi\ {}^{\kappa}\mathsf{U}\ \psi] \equiv \mathsf{E}[\varphi\ {}^{\kappa}\underline{\mathsf{U}}\ \psi] \vee \mathsf{EG}^{\kappa}\varphi$
  - $\mathsf{E}[\varphi\ {}^{\kappa}\underline{\mathsf{U}}\ \psi] \equiv \neg\mathsf{A}[(\neg\psi)\ \mathsf{U}^{\kappa}\ (\neg\varphi \wedge \neg\psi)]$
  - $\mathsf{A}[\varphi\ {}^{\kappa}\mathsf{U}\ \psi] \equiv \neg\mathsf{E}[(\neg\psi)\ \underline{\mathsf{U}}^{\kappa}\ (\neg\varphi \wedge \neg\psi)]$
  - $\mathsf{A}[\varphi\ {}^{\kappa}\underline{\mathsf{U}}\ \psi] \equiv \neg\mathsf{E}[(\neg\psi)\ \underline{\mathsf{U}}^{\kappa}\ (\neg\varphi \wedge \neg\psi)] \wedge \neg\mathsf{EG}\neg\psi$
  - $\mathsf{E}[\varphi\ {}^{\kappa}\mathsf{B}\ \psi] \equiv \neg\mathsf{A}[(\neg\varphi)\ {}^{\kappa}\underline{\mathsf{U}}\ \psi]$
  - $\mathsf{E}[\varphi\ {}^{\kappa}\underline{\mathsf{B}}\ \psi] \equiv \neg\mathsf{A}[(\neg\varphi)\ {}^{\kappa}\mathsf{U}\ \psi]$
  - $\mathsf{A}[\varphi\ {}^{\kappa}\mathsf{B}\ \psi] \equiv \neg\mathsf{E}[(\neg\varphi)\ {}^{\kappa}\underline{\mathsf{U}}\ \psi]$
  - $\mathsf{A}[\varphi\ {}^{\kappa}\underline{\mathsf{B}}\ \psi] \equiv \neg\mathsf{E}[(\neg\varphi)\ {}^{\kappa}\mathsf{U}\ \psi]$

As can be seen, the $\mathsf{EG}^{[a,b]}$ operator states that some property holds for *all* states that can be reached within $[a, b]$, while $\mathsf{E}[\cdot\ \underline{\mathsf{U}}^{[a,b]}\ \cdot]$ states a property for *some* point of time in that interval. $\mathsf{EF}^{\kappa}\varphi$ holds in state s iff a state can be reached where φ holds and that state can be reached within a time that satisfi es the time constraint κ.

$\mathsf{E}[\varphi\ \underline{\mathsf{B}}^{\kappa}\ \psi]$ means that there must be a path π and numbers $i, j \in \mathbb{N}$ with $j < i$ such that φ and ψ hold on $\pi^{(j)}$ and $\pi^{(i)}$, respectively, and the time required to reach $\pi^{(j)}$ satisfies the time constraint κ (hence, φ must hold *before* ψ).

JCTL has also weak variants $\mathsf{E}[\cdot\ \mathsf{U}\ \cdot]$ and $\mathsf{E}[\cdot\ \mathsf{B}\ \cdot]$ of $\mathsf{E}[\cdot\ \underline{\mathsf{U}}\ \cdot]$ and $\mathsf{E}[\cdot\ \underline{\mathsf{B}}\ \cdot]$, respectively: $\mathsf{E}[\varphi\ \mathsf{U}^{\kappa}\ \psi]$ means that there must be a path π such that for all $i \in \mathbb{N}$ the following must hold: whenever a state $\pi^{(i)}$ satisfying ψ can be reached in time t that satisfies κ, then φ must hold for all states $\pi^{(0)}, \ldots, \pi^{(i-1)}$. $\mathsf{E}[\cdot\ \mathsf{B}\ \cdot]$ is defined in a similar way.

Finally, the versions with the universal (A) path quantifier are defined such that the corresponding path property must hold for all paths leaving the state.

As next example, consider now $[\varphi\ {}^{\kappa}\mathsf{B}\ \psi]$. We first prove the validity of the formula on a single path (without path quantifiers), abstracting of the time constraint to an ordinary signal. The equation that is the basis for the definition used in CTL is the LTL theorem $[\varphi\ \mathsf{B}\ \psi] \equiv \neg\,[(\neg\varphi)\ \underline{\mathsf{U}}\ \psi]$. We have

$$\begin{aligned}
&[\varphi\ {}^{\kappa}\mathsf{B}\ \psi] \equiv \\
&\quad [(\kappa \wedge \varphi)\ \mathsf{B}\ \psi] \equiv \neg\,[(\neg(\kappa \wedge \varphi))\ \underline{\mathsf{U}}\ \psi] \equiv \neg\,[(\neg\kappa \vee \neg\varphi))\ \underline{\mathsf{U}}\ \psi] \equiv \neg\,[(\kappa \rightarrow (\neg\varphi))\ \underline{\mathsf{U}}\ \psi] \\
&\equiv \neg\,[(\neg\varphi)\ {}^{\kappa}\underline{\mathsf{U}}\ \psi]
\end{aligned}$$

Adding now path quantifiers to both sides of the equation, is a straightforward task:

- $\mathsf{E}[\varphi\ {}^{\kappa}\mathsf{B}\ \psi] \equiv \neg\mathsf{A}[(\neg\varphi)\ {}^{\kappa}\underline{\mathsf{U}}\ \psi]$
- $\mathsf{A}[\varphi\ {}^{\kappa}\mathsf{B}\ \psi] \equiv \neg\mathsf{E}[(\neg\varphi)\ {}^{\kappa}\underline{\mathsf{U}}\ \psi]$

Adding path quantifiers is however not easily possible when the right hand side of an equation contains nested time-constrained temporal operators, like $\mathsf{AG}^{\leq a}\,(\varphi \wedge \mathsf{AXA}[\varphi\ \mathsf{U}\ \psi])$, since the innermost one must then be reset whenever the operator is evaluated. Nevertheless, path quantifiers can be shifted over temporal operators as done for $\mathsf{LeftCTL}^*$ in [110]. To consider the mentioned example, for $\Phi \in \mathsf{JCTL}$, the law $\mathsf{AG}^{\leq b}\Phi = \mathsf{AG}^{\leq b}\mathsf{A}\Phi$ holds, which is proved as follows:

$$\begin{aligned}
(\mathcal{K}, s) \models \mathsf{AG}^{\leq b}\Phi \quad &\Leftrightarrow \forall\pi \in \mathsf{Paths}_{\mathcal{K}}(s).(\mathcal{K}, \pi^{(0)}) \models \mathsf{G}^{\leq b}\Phi \\
&\Leftrightarrow \forall\pi \in \mathsf{Paths}_{\mathcal{K}}(s).\forall i \in \mathbb{N}.\ \textstyle\sum_{k=0}^{i-1} \tau_{\pi}^{(k)} \leq b \rightarrow (\mathcal{K}, \pi^{(i)}) \models \Phi \\
&\Leftrightarrow \forall\pi \in \mathsf{Paths}_{\mathcal{K}}(s).\forall i \in \mathbb{N}. \\
&\qquad\quad \textstyle\sum_{k=0}^{i-1} \tau_{\pi}^{(k)} \leq b \rightarrow \forall\varrho \in \mathsf{Paths}_{\mathcal{K}}(\pi^{(i)}).(\mathcal{K}, \varrho^{(0)}) \models \Phi \\
&\Leftrightarrow \forall\pi \in \mathsf{Paths}_{\mathcal{K}}(s).\forall i \in \mathbb{N}.\ \textstyle\sum_{k=0}^{i-1} \tau_{\pi}^{(k)} \leq b \rightarrow (\mathcal{K}, \pi^{(i)}) \models \mathsf{A}\Phi \\
&\Leftrightarrow \forall\pi \in \mathsf{Paths}_{\mathcal{K}}(s).(\mathcal{K}, \pi^{(0)}) \models \mathsf{G}^{\leq b}\mathsf{A}\Phi \\
&\Leftrightarrow (\mathcal{K}, s) \models \mathsf{AG}^{\leq b}\mathsf{A}\Phi
\end{aligned}$$

Finally, consider $\mathsf{A}\left[\varphi\ \mathsf{U}^{\leq b}\ \psi\right]$. Let the time constraint be $c \leq b$. Then the following transformations that are clearly valid:

$$\begin{aligned}
&(\mathcal{K}, s) \models \mathsf{A}\big[\varphi\ \mathsf{U}^{\leq b}\ \psi\big] \\
&\qquad \Leftrightarrow (\mathcal{K}, s) \models \mathsf{A}[\varphi\ \mathsf{U}\ \ ((c \leq b) \wedge \psi)] \\
&\qquad \Leftrightarrow \forall \pi \in \mathsf{Paths}_{\mathcal{K}}(s).(\mathcal{K}, \pi^{(0)}) \models [\varphi\ \mathsf{U}\ \ ((c \leq b) \wedge \psi)] \\
&\qquad \Leftrightarrow \forall \pi \in \mathsf{Paths}_{\mathcal{K}}(s).(\mathcal{K}, \pi^{(0)}) \models [\varphi\ \underline{\mathsf{U}}\ \ ((c \leq b) \wedge \psi)] \text{ or } (\mathcal{K}, \pi^{(0)}) \models \mathsf{G}\varphi \\
&\qquad \Leftrightarrow \forall \pi \in \mathsf{Paths}_{\mathcal{K}}(s).(\mathcal{K}, \pi^{(0)}) \models \big[\varphi\ \underline{\mathsf{U}}^{\leq b}\ \psi\big] \text{ or } (\mathcal{K}, \pi^{(0)}) \models \mathsf{G}\varphi \\
&\qquad \overset{(*)}{\Leftrightarrow} \forall \pi \in \mathsf{Paths}_{\mathcal{K}}(s).(\mathcal{K}, \pi^{(0)}) \models \big[\varphi\ \underline{\mathsf{U}}^{\leq b}\ \psi\big] \text{ or } (\mathcal{K}, \pi^{(0)}) \models \big[\varphi\ \underline{\mathsf{U}}^{\leq b}\ \mathsf{AG}\varphi\big] \\
&\qquad \Leftrightarrow \forall \pi \in \mathsf{Paths}_{\mathcal{K}}(s).(\mathcal{K}, \pi^{(0)}) \models \big[\varphi\ \underline{\mathsf{U}}^{\leq b}\ \ (\psi \vee \mathsf{AG}\varphi)\big] \\
&\qquad \Leftrightarrow (\mathcal{K}, s) \models \mathsf{A}\big[\varphi\ \underline{\mathsf{U}}^{\leq b}\ \ (\psi \vee \mathsf{AG}\varphi)\big]
\end{aligned}$$

The only critical equivalence is the one that is marked with $(*)$, so we consider this one in more detail. The direction $\Leftarrow$ is trivial, so it only remains to prove the $\Rightarrow$ direction. For this reason, we have to prove that for any arbitrary path $\pi \in \mathsf{Paths}_{\mathcal{K}}(s)$ the propositions (1) and (2) together imply (3):

$(1)\ \ \forall \varrho \in \mathsf{Paths}_{\mathcal{K}}(s).(\mathcal{K}, \varrho^{(0)}) \models \big[\varphi\ \underline{\mathsf{U}}^{\leq b}\ \psi\big]$ or $(\mathcal{K}, \varrho^{(0)}) \models \mathsf{G}\varphi$

$(2)\ \ (\mathcal{K}, \pi^{(0)}) \models \neg\big[\varphi\ \underline{\mathsf{U}}^{\leq b}\ \psi\big]$

$(3)\ \ (\mathcal{K}, \pi^{(0)}) \models \big[\varphi\ \underline{\mathsf{U}}^{\leq b}\ \mathsf{AG}\varphi\big]$

Instantiating $\varrho := \pi$ in (1) allows to derive with (2) by modus ponens that also

$(4)\ (\mathcal{K}, \pi^{(0)}) \models \mathsf{G}\varphi$

must hold for our path $\pi \in \mathsf{Paths}_{\mathcal{K}}(s)$. Therefore, we conclude from (4) and (2) that

$(5)\ (\mathcal{K}, \pi^{(0)}) \models \mathsf{G}^{\leq b}\neg\psi$

must hold for π. Let now t_b be the greatest position of π where the time required to reach $\pi^{(t_b)}$ is $\leq b$. We now prove

$(6)\ (\mathcal{K}, \pi^{(t_b)}) \models \mathsf{AG}\varphi.$

To this end, let furthermore be $\xi \in \mathsf{Paths}_{\mathcal{K}}(s)$ such that $\forall t \leq t_b.\xi^{(t)} = \xi^{(t)}$ holds, i.e. ξ shares the prefi x $\xi^{(0)}, \ldots, \xi^{(t_b)}$. For this reason, it also follows that

$(7)\ (\mathcal{K}, \xi^{(0)}) \models \neg\big[\varphi\ \underline{\mathsf{U}}^{\leq b}\ \psi\big]$

holds due to (2). But now, (3) follows: clearly, for $b > 0$ there is a position on π where the time required to reach this position is $\leq b$ (the equation is trivial for $b = 0$), we select the greatest such position, which is t_b. Now, (3) follows simply from (4) and (6).

3.2.3 Reducing Basic JCTL Operators

As already mentioned in section 3.2, the basic operators $\mathsf{EG}^{[0,b]}$ and $\mathsf{EG}^{\geq a}$ are used as reference, in order to simplify formulae notions, since they can be also defined as abbreviations. For the $\mathsf{EG}^{\geq a}$ operator we have:

$$\begin{aligned}
\mathsf{EG}^{\geq a+1}\varphi &= \neg\mathsf{AF}^{\geq a+1}\neg\varphi\\
&= \neg\mathsf{A}\left[1\ \underline{\mathsf{U}}^{\geq a+1}\ \neg\varphi\right]\\
&= \neg\mathsf{AG}^{\leq a}\,(1 \wedge \mathsf{AXA}[1\ \underline{\mathsf{U}}\ \neg\varphi])\\
&= \neg\mathsf{AG}^{\leq a}\mathsf{AXAF}\neg\varphi\\
&= \mathsf{EF}^{\leq a}\mathsf{EXEG}\varphi
\end{aligned}$$

Furthermore, the following lemma holds, which shows that also the $\mathsf{EG}^{[0,b]}(=\mathsf{EG}^{\leq b})$ is a somehow hybrid operator that makes both a universal and an existential statement (compare G_2 and G_3 in the following lemma). This is due to the fact, that the equation $\mathsf{EG}^{\leq b}\varphi = \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{>b}\ 1]$ is valid:

Lemma 1 (Semantics of $\mathsf{EG}^{\leq b}\varphi$) *Given a* JCTL *formula* φ *and a number* $b \in \mathbb{N}$. *Then, the following properties are equivalent for any TKS* $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$:

(G_1) $(\mathcal{K}, s) \models \mathsf{EG}^{\leq b}\varphi$

(G_2) *there is a path* $\pi \in \mathsf{Paths}_{\mathcal{K}}(s)$ *starting in state* s, *such that for all numbers* $i \in \mathbb{N}$, *we have*
$$\left(\sum_{j=0}^{i-1} \tau_\pi^{(j)} \leq b\right) \rightarrow \left((\mathcal{K}, \pi^{(i)}) \models \varphi\right)$$

(G_3) *there is a path* $\pi \in \mathsf{Paths}_{\mathcal{K}}(s)$ *starting in state* s *and a number* $i \in \mathbb{N}$ *such that*
$$\left(\sum_{j=0}^{i-1} \tau_\pi^{(j)} \leq b < \sum_{j=0}^{i} \tau_\pi^{(j)}\right) \wedge \left(\forall j \leq i.\ (\mathcal{K}, \pi^{(j)}) \models \varphi\right)$$

(G_4) *there is a path* $\pi \in \mathsf{Paths}_{\mathcal{K}}(s)$ *starting in state* s *and a number* $i \in \mathbb{N}$ *such that*
$$\left(\sum_{j=0}^{i-1} \tau_\pi^{(j)} > b\right) \wedge \left(\forall j < i.\ (\mathcal{K}, \pi^{(j)}) \models \varphi\right)$$

(G_5) $(\mathcal{K}, s) \models \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{>b}\ 1]$

The proof of the above lemma is not very difficult. We just make use of the well-ordering of natural numbers, i.e., if there is a number with some property, then there is also a least number with the same property.

It follows that the $\mathsf{EG}^{[0,b]}$ and $\mathsf{EG}^{\geq a}$ operators can be expressed as abbreviations of the qualitative-only CTL operators EX and EG and the JCTL operator $\mathsf{E}[\cdot\ \underline{\mathsf{U}}^{\kappa}\ \cdot]$:

- $\mathsf{EG}^{\leq b}\varphi \equiv \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{>b}\ 1]$
- $\mathsf{EG}^{\geq a+1}\varphi \equiv \mathsf{EF}^{\leq a}\mathsf{EXEG}\varphi$
- $\mathsf{EG}^{\geq 0}\varphi \equiv \mathsf{EG}\varphi$

To summarize, we see that also basic JCTL operators can be expressed as abbreviations. Nevertheless, in order to simplify formulae notions, but also to be able to directly compare JCTL with other existing real-time logics (cf. section 4) it is sensible to use the notion and the semantics of the JCTL operators as given in defi nitions 9 and 11.

Chapter 4

Problems of Previous Approaches

(Οὐ) τοῦ δοκεῖν μοι, τῆς δ' ἀληθείας μέλει. [1]
ΑΣΤΥΔΑΜΑΣ (4ος αἰών π.Χ.)
(Αλκμέων, απόσπ. 1, Nauck)

In this section we discuss the mentioned deficiencies (cf. section 1.1.4) of existing related approaches in more detail. In particular, we consider problems that occur when timed systems are expanded to ‘corresponding’ unit delay systems. In general, we show that these expansions yield in misleading and even wrong results. Authors of previous approaches have defined their logics so that these results match with their semantics. As outlined in section 1.1.4, we will however see in more detail, that all of these approaches to fix these problems yield in further problems. Moreover, we will show in section 4.3 that their results are in contrast to the results of other established approaches, like timed automata.

4.1 Time-Models of Previous Approaches

In order to precisely point out the deficiencies of the previously mentioned approaches we must first consider their time-models together with the notions of the *expansion of a timed Kripke structure* and the *track of a state*.

All previous approaches that use Kripke-structure-based formalisms are defined according to a *Stuttering Interpretation* I_S of timed transition systems:

Stuttering Interpretation I_S**:** A transition from state s_1 to state s_2 with label $k > 1$ is seen as abbreviation for a stuttering sequence $s_1 \xrightarrow{1} s_{1,1} \xrightarrow{1} \ldots \xrightarrow{1} s_{1,k-1} \xrightarrow{1} s_2$ where s_1 is repeated, according to the time consumption, so all the states $s_{1,i}$ have the same variable assignment as state s_1.

[1] I don’t care about different opinions, only about the truth.
ASTYDAMAS (4th cent. BC)

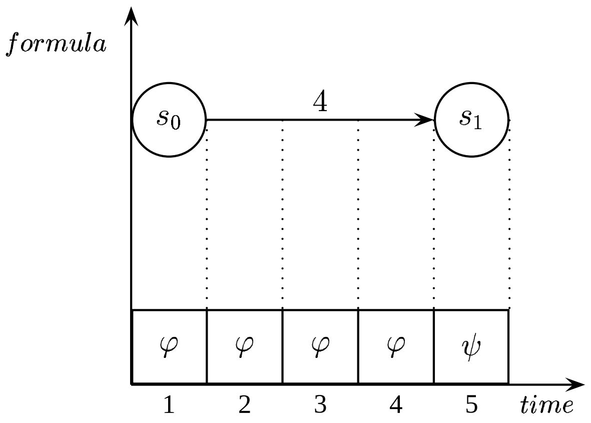


Figure 4.1: A Timed Transition According to Stuttering Interpretation I_S

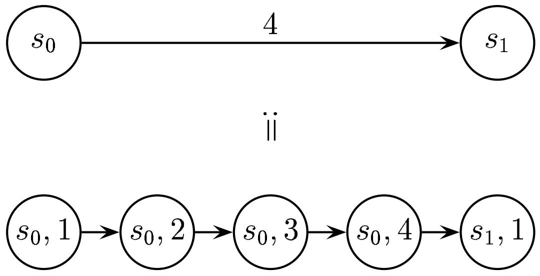


Figure 4.2: Interpretation I_S Represents Stuttering Sequences of Untimed Transitions

As example, consider Figure 4.1, which shows a timed transition according to stuttering interpretation I_S. At the point of time $t = 1$ we are in state s_0 where the formula φ is valid. The transition from s_0 to s_1 requires 4 time units. The formula φ is valid at the points of time $t = 2, t = 3, t = 4$, between the states s_0 and s_1. At the point of time $t = 5$, state s_1 is reached, where the formula ψ is valid.

As can be seen in Figure 4.2 the stuttering interpretation I_S represents transitions in timed systems as abbreviations for stuttering sequences of single untimed transitions. Hence, it cannot handle *atomic timed transitions*. In this work, to distinguish between timed Kripke structures that are based on I_J and other timed transition systems that are based on I_S, we call the latter ones *stuttering Kripke structures (SKSs)*.

Stuttering Kripke structures (SKSs) can be *expanded* into unit-delay structures by means of certain expansion techniques that work according to different *expansion semantics*. A possible expansion is defi ned in Figure 4.3.

```
**function** expand($\mathcal{I}, \mathcal{S}, \mathcal{R}$)
    $\mathcal{S}_e := \{(s, 1) \mid s \in \mathcal{S}\}$;
    $\mathcal{R}_e := \{\}$;
    **for** $(s, t, s') \in \mathcal{R}$ **do**
        **for** $i := 2$ **to** $t$ **do**
            $\mathcal{S}_e := \mathcal{S}_e \cup \{(s, i)\}$;
            $\mathcal{R}_e := \mathcal{R}_e \cup \{((s, i-1), (s, i))\}$;
        **end for**;
    $\mathcal{R}_e := \mathcal{R}_e \cup \{((s, t), (s', 1))\}$;
    **end for**;
    $\mathcal{I}_e := \{(s, 1) \mid s \in \mathcal{I}\}$;
    **return** $(\mathcal{I}_e, \mathcal{S}_e, \mathcal{R}_e)$;
**end function**
```

Figure 4.3: Expansion of a Timed Transition System

Definition 14 (Expansion of SKSs) *Given a SKS $\mathcal{K}$ = ($\mathcal{I}$, $\mathcal{S}$, $\mathcal{R}$, $\mathcal{L}$), we compute $(\mathcal{I}_e, \mathcal{S}_e, \mathcal{R}_e) =$ expand$(\mathcal{I}, \mathcal{S}, \mathcal{R})$ with the function* expand *as defined in Figure 4.3. Moreover, we define for any $(s, u) \in \mathcal{S}_e$ the label function $\mathcal{L}_e((s, u)) := \mathcal{L}(s)$. The expansion of $\mathcal{K}$ is then the unit delay structure $\mathcal{K}_e$:= ($\mathcal{I}_e$, $\mathcal{S}_e$, $\mathcal{R}_e$, $\mathcal{L}_e$).*

As can be seen, the expansion relies on interpretation I_S, since we defi ned $\mathcal{L}_e((s, u)) := \mathcal{L}(s)$, i.e., the states of $\mathcal{K}_e$ have the same variable assignments as the corresponding states of $\mathcal{K}$. For conciseness, we use the following defi nition:

Definition 15 (Tracks of a State) *Given a SKS $\mathcal{K}$ = ($\mathcal{I}$, $\mathcal{S}$, $\mathcal{R}$, $\mathcal{L}$), its expanded structure $\mathcal{K}_e$:= ($\mathcal{I}_e$, $\mathcal{S}_e$, $\mathcal{R}_e$, $\mathcal{L}_e$), and a state $s \in \mathcal{S}$. Then, we define $\mathsf{Tracks}_{\mathcal{K}}(s) = \{(s', u) \in \mathcal{S}_e \mid s = s'\}$. A track in an expanded structure is a tuple (s, k), where $s \in \mathcal{S}, k \in \mathbb{N}$. Furthermore, we call the tuple $(s, 1)$ with $s \in \mathcal{S}$, a main track in an expanded structure.*

Generally, SKSs are abbreviations of UDSs, where sets of states are merged into timed transitions. This is one of the main reasons for the problems described in the next sections: merging sets of states into timed transitions clearly leads into loss of the internal node-connections on the merged paths.

For many purposes like model-checking, composition, etc., a SKS must be expanded into a UDS. We emphasize however, that expansions of SKSs can be performed in many different ways that are not equivalent to each other (cf. Figure 4.7). We will discuss this issue in more detail in section 4.2.2. In contrast to this, as explained in 3.1, UDSs are special cases of TKS that are obtained by restricting TKSs so that $(s, t, s') \in \mathcal{R}$ implies $t = 1$.

4.2 Description of the Problems

4.2.1 Overlapping Time Constraints

In [57], it has been observed that timed transitions may be differently interpreted as already explained. However, [57] did neither decide to use I_J nor to use I_S, and instead mixed both interpretations in that different temporal operators interpret transitions differently, i.e., according to either I_J or I_S. For example, [57] defines a temporal operator $\widetilde{\mathsf{EF}}^k \varphi$ as follows [2]:

$(\mathcal{K}, s) \models \widetilde{\mathsf{EF}}^k \varphi$ iff there is a path $\pi \in \mathsf{Paths}_{\mathcal{K}}(s)$ with associated duration function τ_π and a $i \in \mathbb{N}$, so that $\big((\mathcal{K}, \pi^{(i)}) \models \varphi\big) \wedge \sum_{j=0}^{i} \tau_\pi^{(j)} > k$ holds.

Intuitively, $\widetilde{\mathsf{EF}}^k \varphi$ amounts to say that there is a path where φ holds after the time k has been consumed. However, this is not the case, which can be shown by the following example.

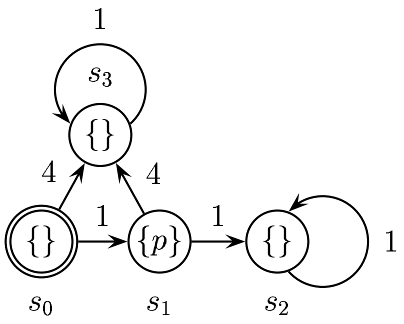


Figure 4.4: Problematic Example for [57]

Consider the state s_0 of the structure given in Figure 4.4. For comparison, note first that $\mathsf{EF}^{>3} p$ means that there must be a path starting in state s_0 where we reach some state s in time > 3 where p holds. This is not the case for any of the states in Figure 4.4, and hence, we have $[\![\mathsf{EF}^{>3} p]\!]_{\mathcal{K}} = \{\}$.

However, we have $\left[\!\!\left[\widetilde{\mathsf{EF}}^3 p\right]\!\!\right]_{\mathcal{K}} = \{s_0, s_1\}$!

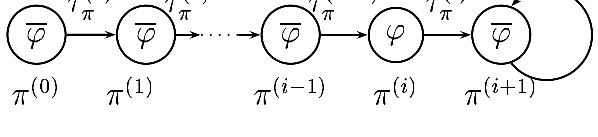


Figure 4.5: $\mathsf{EF}^{>k}\varphi$ and $\widetilde{\mathsf{EF}}^k \varphi$

[2]In [57], the notation $\mathsf{EXW}[k]\varphi$ (weak next) has been chosen, but in analogy to the JCTL EF operator, we prefer the notation above.

The difference between $\widetilde{\mathsf{EF}}^{k}$ and the JCTL $\mathsf{EF}^{>k}$ is best seen by the transition system in Figure 4.5 that contains a single path π, where φ holds only on state $\pi^{(i)}$. Furthermore, assume that $\sum_{j=0}^{i-1}\tau_\pi^{(j)} \leq k < \sum_{j=0}^{i}\tau_\pi^{(j)}$ holds. Then, the formula $\widetilde{\mathsf{EF}}^{k}\varphi$ will be satisfi ed on π, while $\mathsf{EF}^{>k}\varphi$ will not be satisfi ed. The reason is that [57] interpret$\widetilde{\mathsf{EF}}^{k}\varphi$ with interpretation I_S so that it will be satisfi ed by some of the intermediate states between $\pi^{(i)}$ and $\pi^{(i+1)}$, while JCTL operators do not consider any intermediate states at all.

We are not aware of any reasonable application, where additional time constraints of actions are to be considered that are not even taken. But even more severe, the coexistence of both interpretations of timed transitions makes it impossible to defi ne operations on TKS like composition of structures or simulation preorders.

4.2.2 Branching Problems

Becoming aware of the misleading mixture of semantics used in [57], [109] decided to only use interpretation I_S. However, another problem with the approach in [109] is that different expansion techniques yield in different results.

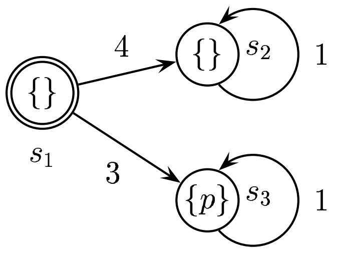


Figure 4.6: A Timed Transition System

To see this, consider the SKS given in Figure 4.6 and two different expansions of it that are given in Figure 4.7. In the expansion $\mathcal{K}_1$ (upper structure), we have added to every timed transition intermediate states according to the time delay. This is in accordance with the technique of [27]. In the second expansion $\mathcal{K}_2$ (lower structure), we have shared the intermediate states as long as possible which is the expansion technique of Figure 4.3 that is currently also preferred by [109].

The two structures $\mathcal{K}_1$ and $\mathcal{K}_2$ are not equivalent. To see this, consider the formula

$\Phi := \mathsf{AF}^{\leq 1}\mathsf{EF}^{\leq 2}p.$

In $\mathcal{K}_1$, we obtain $[\![\Phi]\!] = [\![\mathsf{EF}^{\leq 2}p]\!] = \{(s_1', 2), (s_1', 3), (s_3, 1)\}$.

In $\mathcal{K}_2$, we however obtain $[\![\mathsf{EF}^{\leq 2}p]\!] = \{(s_1, 2), (s_1, 3), (s_3, 1)\}$, and therefore $[\![\Phi]\!] = \{(s_1, 1),(s_1, 2), (s_1, 3), (s_3, 1)\}$.

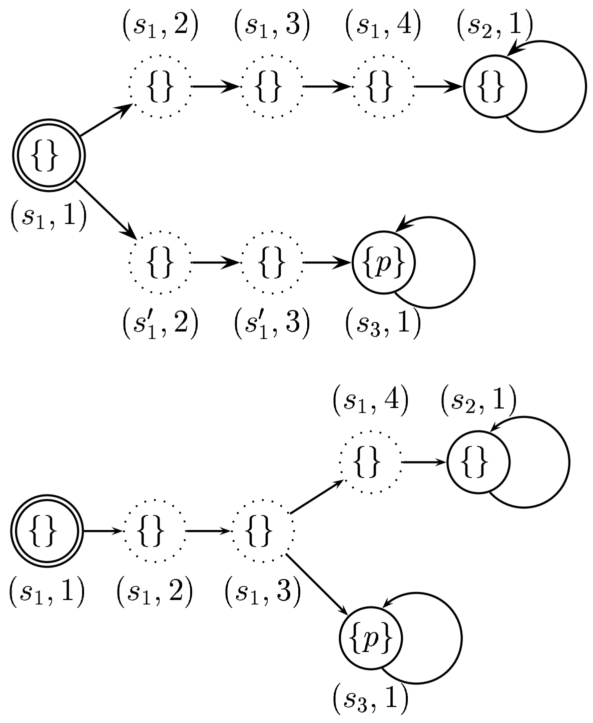


Figure 4.7: Different Expansions of the SKS in Figure 4.6

Hence, the semantics of [109] and [27] depend on the chosen expansion. As a consequence, the approaches followed in [109] and [27] are also not equivalent to each other. Moreover, as the real-time temporal logics used in [109] and [27] both use SKSs as models, it is unsatisfactory to use something else but the SKS to define the semantics.

As the results of the expanded structure moreover seems to have nothing to do with the original problem, i.e., the results obtained for an expanded structure can not be directly transferred back to the timed transition system, we may consider [109] in principle as an approach based on unit-delay Kripke structures.

Furthermore, it is erroneously argued in [109] that the semantics as given in [57] is not intuitive, since in general, we have $\mathsf{EF}^{\leq 6}p \neq \mathsf{EF}^{\leq 3}\mathsf{EF}^{\leq 3}p$. However, both formulae express different things: $\mathsf{EF}^{\leq 3}\mathsf{EF}^{\leq 3}p$ means that we can reach some state in at least 3 units of time where in further 3 units of time, a possibly different state is reached where p holds. This is obviously not the same as $\mathsf{EF}^{\leq 6}p$, which means that we can reach a state where p holds in at least 6 units of time. In general, we only have $\mathsf{EF}^{\leq 3}\mathsf{EF}^{\leq 3}p \rightarrow \mathsf{EF}^{\leq 6}p$, but not vice versa.

4.2.3 Problems by Nesting Operators

One might think that we can simply expand SKSs to obtain corresponding unit-delay structures in order to use the traditional CTL model checking algorithms. However, we show by the following example, that this is not the case.

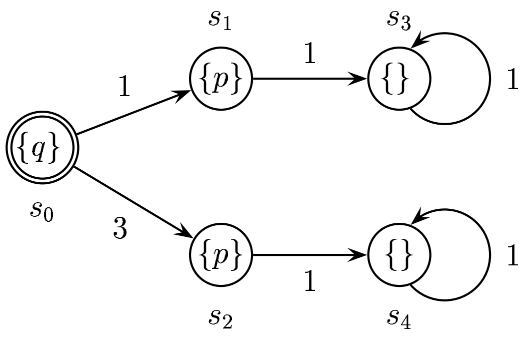


Figure 4.8: Example for the Expansion Problem

Consider fi rst the timed transition system given in Figure 4.8 as a TKS and the formula

$$\Phi_E := \mathsf{EG}^{\leq 2}\mathsf{EF}^{\leq 1}p$$

To evaluate this formula, according to JCTL semantics we successively obtain the following sets of states:

$[\![p]\!] = \{s_1, s_2\}$, $[\![\mathsf{EF}^{\leq 1}p]\!] = \{s_0, s_1, s_2\}$, and thus $[\![\mathsf{EG}^{\leq 2}\mathsf{EF}^{\leq 1}p]\!] = \{s_0\}$.

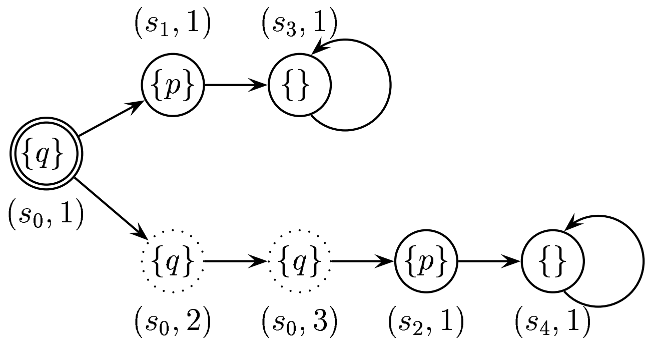


Figure 4.9: Expansion of the Timed Transition System of Figure 4.8

Now, consider again the timed transition system given in Figure 4.8 as a SKS and the formula Φ_E. In order to evaluate Φ_E on a SKS, we must consider the expanded structure as given in Figure 4.9. Please note, that the expanded structure of this system is identical for both expansion semantics used by [109] and [27].

Evaluating again the formula $\mathsf{EG}^{\leq 2}\mathsf{EF}^{\leq 1}p$ using the real-time logics of [109] and [27] we obtain:

$[\![p]\!] = \{(s_1, 1), (s_2, 1)\}$ and $[\![\mathsf{EF}^{\leq 1}p]\!] = \{(s_1, 1), (s_0, 1), (s_2, 1), (s_0, 3)\}$.

To evaluate $[\![\mathsf{EG}^{\leq 2}\mathsf{EF}^{\leq 1}p]\!]$, we must now check which of the expansion states have a path so that we are in $[\![\mathsf{EF}^{\leq 1}p]\!]$ as long as the time consumption is ≤ 2. It is not difficult to see that there is no such state, and hence, we now have $[\![\mathsf{EG}^{\leq 2}\mathsf{EF}^{\leq 1}p]\!] = \{\}$.
Hence, we see by the above example, that SKSs and their expanded UDSs are not bisimilar. The example moreover shows an intrinsic problem of approaches like [27, 109] that are based on I_S:

The evaluation of a formula φ may yield in a set of tracks $\mathcal{T}_\varphi$, where $(s, i) \in \mathcal{T}_\varphi$ holds for one track (s, i), but $(s, j) \in \mathcal{T}_\varphi$ does not hold for another track (s, j). As such 'inconsistent' sets of tracks may occur (although the tracks of atomic formulae are consistent) it follows that the approaches based on I_S can not define in which states of the timed transition system of Figure 4.8 a formula like $\mathsf{EF}^{\leq 1}p$ holds. This yields in problems when operators are nested, as demonstrated by the above example. Therefore, [109] performs all computations on the expanded structure 4.9 to compute the set of tracks $\mathcal{T}_\varphi$ where a formula φ holds, and then checks whether $\mathcal{I} \times \{1\} \subseteq \mathcal{T}_\varphi$ holds.

4.3 A Comparison to Timed Automata

The authors of [109, 27] have defined their logics so that their results match with their semantics. However, to obtain a solid picture about these approaches, one should test the 'compatibility' of their results against other, popular and established approaches. The best paradigm for such an approach is the formalism of timed automata and the real-time logic TCTL.

For this purpose, we have described the timed transition system shown in Figure 4.8 by means of timed automata and have used the tool Kronos [40] to verify the formula $\Phi_E := \mathsf{EG}^{\leq 2}\mathsf{EF}^{\leq 1}p$. The results are shown in Figure 4.10. Kronos clearly states that the formula Φ_E is true at the initial states init of the system, which is s_0.

These results agree with the results of our logic JCTL and are in contrast to the approaches of [27] and [109], as shown in section 4.2.3.

Formal Model:

```
#states 5
#trans 6
#clocks 1 CLK

state : 0
prop : Q
invar : TRUE
trans :
CLK = 1 =>; CLK := 0; goto1
CLK = 3 =>; CLK := 0; goto3

state : 1
prop : P
invar : TRUE
trans :
CLK = 1 =>; CLK := 0; goto2

state : 2
prop : DONTCARE1
invar : TRUE
trans :
CLK = 1 =>; CLK := 0; goto2

state : 3
prop : P
invar : TRUE
trans :
CLK = 1 =>; CLK := 0; goto4

state : 4
prop : DONTCARE2
invar : TRUE
trans :
CLK = 1 =>; CLK := 0; goto4
```

Specification:

```
init impl eb{<= 2} (ed{<= 1} P)
```

Verification:

```
true
```

Figure 4.10: Counterexample Described by Timed Automata and Verifi ed by Kronos

Chapter 5

Real-Time Symbolic Model Checking

Ἄγει δέ πρός φῶς τήν ἀλήθειαν χρόνος. [1]

ΜΕΝΑΝΔΡΟΣ (342 - 291 π.Χ.)

(Γν. μον. 11, Meinecke)

5.1 Real Time Model Checking on Timed Kripke Structures

In this section, we present model checking algorithms for the real-time temporal logic JCTL. The essential idea is to take advantage of successful CTL symbolic techniques by extending their operation for real-time constraints: the algorithms are applied to timed Kripke structures and are based on fi xpoint calculations that operate with respect to time constraints.

Note also that we consider timed Kripke structures that do not contain fi nite paths. Moreover, it is advisable to perform a reachability analysis in advance, since this can be easily done on qualitative-only level and hence release the model checking algorithms from complex calculations on timed paths of unreachable states. The qualitative-only transition system (a UDS) is always given in advance – before the TKS construction (cf. sections 6.1 and 6.2), so there are no additional operations required in order to obtain it.

Given a transition relation $\mathcal{U}$ of a UDS $\mathcal{K}_{\mathcal{U}}$, the function Deadends shown in Figure 5.1 eliminates all fi nite paths contained in $\mathcal{U}$ and returns a fi nite-path-free transition relation. This is done by iteratively computing all states in $\mathcal{K}_{\mathcal{U}}$ that have at least one successor state.

The function Reach clearly computes the set of states that are reachable from the states $\mathcal{S}_{\varphi}$ by the transitions of $\mathcal{U}$.

The key idea to reason about the real-time constraints is to move fronts of tracks on a TKS, which is virtually expanded like a SKS. However, we emphasize that *we do never expand the structure*. Furthermore, we do not run into semantic problems since the result of any evaluation

[1] Time brings truth into light.
MENANDROS (342 - 291 BC)

of a logical operator is *a set of states* instead of a set of tracks. *Hence, all calculations are independent of the virtual expansion.* This is achieved by abstraction of the set of tracks, so that the semantics of the evaluated temporal operator is respected. The algorithms are able to correctly translate the model checking results obtained after a fi xpoint iteration on any virtually expanded structure to the abstract structure so that the interpretation I_J is respected.

In general, there are two possibilities: On the one hand, a state s belongs to the result if its main track $(s, 1)$ belongs to the track set, on the other hand it may be suffi cient if anyone of its tracks (s, t) belongs to the track set. The choice between the two possibilities depends on the semantics of the considered temporal operator. The underlying algorithms are given in Figures 5.3, 5.4, 5.5 and 5.6.

```
function Deadends($\mathcal{U}$)
 $\mathcal{U}_{inf} := \mathcal{U}$;
  repeat
   $\mathcal{U}_{next} := \mathcal{U}_{inf}$;
   $\mathcal{S}_{next} := \{s \in \mathcal{S} \mid \exists s' \in \mathcal{S}.(s, s') \in \mathcal{U}_{next}\}$;
   $\mathcal{U}_{inf} := \mathcal{U}_{inf} \cap (\mathcal{S} \times \mathcal{S}_{next})$;
  until $\mathcal{U}_{inf} = \mathcal{U}_{next}$;
  return $\mathcal{U}_{inf}$;
end function

function Reach($\mathcal{U}, \mathcal{S}_\varphi$)
 $\mathcal{S}_{reach} := \mathcal{S}_\varphi$
  repeat
   $\mathcal{S}_{old} := \mathcal{S}_{reach}$;
   $\mathcal{S}_{next} := \{s' \in \mathcal{S} \mid (s, s') \in \mathcal{U} \wedge s \in \mathcal{S}_{reach}\}$;
   $\mathcal{S}_{reach} := \mathcal{S}_{reach} \cup \mathcal{S}_{next}$;
  until $\mathcal{S}_{reach} = \mathcal{S}_{old}$;
  return $\mathcal{S}_{reach}$;
end function
```

Figure 5.1: Algorithms for Finite Paths Elimination and Reachability Analysis

The main function for the evaluation of all real-time constraints is the function MoveFront. Given a set of tracks $\mathcal{T}_\psi$ and a set of states $\mathcal{S}_\varphi$, this function computes the set of tracks that have a path of a certain length through the tracks $\mathcal{S}_\varphi \times \mathbb{N}$. The precise specifi cation is as follows:

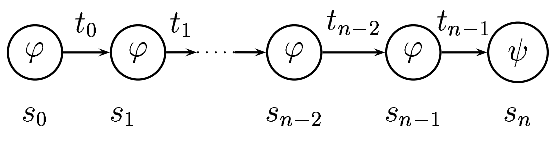


Figure 5.2: Correctness of MoveFront

Lemma 2 (Correctness of $\mathsf{MoveFront}$**)** *Given a TKS $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$, a set of states $\mathcal{S}_\varphi$, and a set of tracks $\mathcal{T}_\psi$, the function* $\mathsf{MoveFront}$ *as given in Figure 5.4 satisfies the following equations for $\sim \in \{strong, weak\}$ (cf. Figure 5.2):*

$$(s_0, t) \in \mathsf{MoveFront}(\sim, k, \mathcal{S}_\varphi, \mathcal{T}_\psi) \quad \Leftrightarrow \quad \left(\begin{array}{l} \exists s_1, \ldots, s_{n-1} \in \mathcal{S}_\varphi . \exists (s_n, d) \in \mathcal{T}_\psi . \\ \quad \exists t_0, \ldots, t_{n-1} \in \mathbb{N}. \\ \quad\quad \bigwedge_{i=0}^{n-1} (s_i, t_i, s_{i+1}) \in \mathcal{R} \land \\ \quad\quad k \geq \left(\sum_{i=0}^{n-1} t_i \right) + d - t \land \\ \quad\quad t \leq t_0 \end{array} \right)$$

Hence, $\mathsf{MoveFront}(\sim, k, \mathcal{S}_\varphi, \mathcal{T}_\psi)$ *computes the set of tracks that have a path through any expanded structure of length ℓ with $\ell \geq k$ to a track in $\mathcal{T}_\psi$ which runs only through tracks of $\mathcal{S}_\varphi \times \mathbb{N}$.*

Proof: The correctness easily follows by induction on k, when we observe that our algorithm and the right hand sides of the above equivalence both satisfy the following recursion equations (note that primitive recursive defi nitions are uniquely determined):

- $\mathsf{MoveFront}(\sim, 0, \mathcal{S}_\varphi, \mathcal{T}_\psi) = \mathcal{T}_\psi$
- $\mathsf{MoveFront}(strong, k{+}1, \mathcal{S}_\varphi, \mathcal{T}_\psi) = (\mathcal{S}_\varphi \times \mathbb{N}) \cap \mathsf{preTracks}(\mathsf{MoveFront}(strong, k, \mathcal{S}_\varphi, \mathcal{T}_\psi))$
- $\mathsf{MoveFront}(weak, k+1, \mathcal{S}_\varphi, \mathcal{T}_\psi) = \left(\begin{array}{l} \mathsf{let}\ \mathcal{T}_0 := \mathsf{MoveFront}(weak, k, \mathcal{S}_\varphi, \mathcal{T}_\psi) \\ \mathsf{in}\ \ \mathcal{T}_0 \cup ((\mathcal{S}_\varphi \times \mathbb{N}) \cap \mathsf{preTracks}(\mathcal{T}_0)) \end{array} \right)$

Using these equations, we can easily prove that $\mathcal{T}_0 = \mathsf{MoveFront}(\sim, i, \mathcal{S}_\varphi, \mathcal{T}_\psi)$ is an invariant of the loop in the algorithm given in Figure 5.4 for $\mathsf{MoveFront}$. This directly implies the correctness of the above lemma. ■

Please note that in the algorithm given in Figure 5.4 to implement the function $\mathsf{MoveFront}$, we use $(i \neq k) \land ((\Lambda \cap (\mathcal{T}_0 \times \mathbb{N})) \neq \{\}$ and $(i \neq k) \land (\mathcal{T}_0 \neq \mathcal{T}_1)$ as conditions of the two loops instead of $(i \neq k)$ only (which would also be correct). The reason for this is that whenever $(\Lambda \cap (\mathcal{T}_0 \times \mathbb{N})) \neq \{\}$ or $\mathcal{T}_0 = \mathcal{T}_1$ hold for an iteration $i < k$, then it follows that all fronts $\mathsf{MoveFront}(\sim, i, \mathcal{S}_\varphi, \mathcal{T}_\psi)$, ..., $\mathsf{MoveFront}(\sim, k, \mathcal{S}_\varphi, \mathcal{T}_\psi)$ would be identical, so that we already have the result in this case.

Now consider the function $\mathsf{StatesEU}^{[a,b]}$. We fi rst compute the set of tracks $\mathcal{T}_0$ that can reach a track of $\mathcal{S}_\psi \times \{1\}$ in exactly a steps (where only tracks of the states $\mathcal{S}_\varphi$ are traversed). After

```
function preStates($\mathcal{S}_0$)
  $\mathcal{S}_1 := \left\{ s \in \mathcal{S} \,\middle|\, \begin{array}{r} \exists s' \in \mathcal{S}_0.\exists t \in \mathbb{N}. \\ (s,t,s') \in \mathcal{R} \end{array} \right\}$;
  return $\mathcal{S}_1$;
end function

function preTracks($\mathcal{T}$)
  $\mathcal{T}_1 := \{(s,t) \mid (s,t+1) \in \mathcal{T}\}$;
  $\mathcal{T}_2 := \{(s,t) \mid \exists (s',1) \in \mathcal{T}.(s,t,s') \in \mathcal{R}\}$;
  return $\mathcal{T}_1 \cup \mathcal{T}_2$;
end function
```

Figure 5.3: Symbolic Algorithms for Traversing on a TKS $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$

this, we move the tracks $\mathcal{T}_0$ by further $b - a$ unit delay steps through the tracks of the states $\mathcal{S}_\varphi$. By the above lemma, we then obtain

$$(s_0', t') \in \mathcal{T}_1 \Leftrightarrow \left(\begin{array}{l} \exists s_1', \ldots, s_{m-1}' \in \mathcal{S}_\varphi.\exists t_0', \ldots, t_{m-1}' \in \mathbb{N}. \\ \exists s_0, \ldots, s_{n-1} \in \mathcal{S}_\varphi.\exists t_0, \ldots, t_{n-1} \in \mathbb{N}. \\ \exists s_n \in \mathcal{S}_\psi.\exists t \in \mathbb{N}. \\ \quad \bigwedge_{i=0}^{m-1} (s_i', t_i', s_{i+1}') \in \mathcal{R} \wedge \\ \quad (s_m' = s_0) \in \mathcal{R} \wedge \\ \quad \bigwedge_{i=0}^{n-1} (s_i, t_i, s_{i+1}) \in \mathcal{R} \wedge \\ \quad a = \left(\sum_{i=0}^{n-1} t_i\right) + 1 - t \wedge \\ \quad b \geq \left(\sum_{i=0}^{m-1} t_i'\right) + \left(\sum_{i=0}^{n-1} t_i\right) + 1 - t' \wedge \\ \quad t \leq t_0 \wedge t' \leq t_0' \end{array} \right)$$

Note that $\left(\sum_{i=0}^{m-1} t_i'\right) + \left(\sum_{i=0}^{n-1} t_i\right) + 1 - t'$ is the number of unit delay steps that are required to reach track $(s_n, 1) \in \mathcal{S}_\psi \times \{1\}$ from track (s_0', t'). By the above result, it follows that this time is in the interval $[a, b]$ (consider the case $m = 0$, where the second MoveFront went along a single transition, and the case where $m > 0$ holds). The fi nal step is to translate this result (given for tracks) to sets of states. If $(s_0', 1) \in \mathcal{T}_1$ holds, then we clearly see that state s_0 belongs to $\llbracket \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[a,b]}\ \psi] \rrbracket_{\mathcal{K}}$. If, on the other hand, $(s_0', t') \in \mathcal{T}_1$ holds for some $t' > 1$, but $(s_0', 1)$ is not included in $\mathcal{T}_1$, then it follows by the above formula that the time to reach $(s_n, 1)$ from $(s_0', 1)$ is larger than b, so that $s_0' \notin \llbracket \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[a,b]}\ \psi] \rrbracket_{\mathcal{K}}$.

The correctness of the function to evaluate $\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\geq a}\ \psi]$ is proved in a similar way.

The correctness of the function $\mathsf{EG}^{[a,b]}\varphi$ can be seen as follows:

```
**function** MoveFront($\sim, k, \mathcal{S}_\varphi, \mathcal{T}_\psi$)
  $\mathcal{T}_\varphi := \mathcal{S}_\varphi \times \mathbb{N}$;
  $\mathcal{T}_0 := \mathcal{T}_\psi$;
  $\mathcal{T}_1 := \{\}$;
  $\Lambda := \{\}$;
  $i := 0$;
  **if** equal($\sim, strong$) **then**
    **while** $(i \neq k) \wedge ((\Lambda \cap (\mathcal{T}_0 \times \mathbb{N})) \neq \{\})$ **do**
      $\mathcal{T}_1 := \mathcal{T}_0$;
      $\mathcal{T}_0 := \mathcal{T}_\varphi \cap \mathsf{preTracks}(\mathcal{T}_1)$;
      $\Lambda := \Lambda \cup (\mathcal{T}_1, i+1)$;
      $i := i+1$;
    **end**;
    **if** $(i = k)$ **then return** $\mathcal{T}_0$;
    **else**
      $\lambda := \Lambda \cap (\mathcal{T}_0 \times \mathbb{N})$;
      $\{cnt_{break}\} := \{j \in \mathbb{N} \mid (s,t,j) = \lambda\}$;
      $cnt_{out} := ((k-i)\mathsf{mod}(i - cnt_{break})) + cnt_{break} - 1$;
      $\lambda_{out} := \Lambda \cap (\mathcal{S} \times \mathbb{N} \times \{cnt_{out}\})$;
      $\{\mathcal{T}_{out}\} := \{\mathcal{T} \mid (\mathcal{T}, j) = \lambda_{out}\}$;
    **return** $\mathcal{T}_{out}$;
  **else** $//if\mathsf{equal}(\sim, weak)$
    **while** $(i \neq k) \wedge (\mathcal{T}_0 \neq \mathcal{T}_1)$ **do**
      $\mathcal{T}_1 := \mathcal{T}_0$;
      $\mathcal{T}_0 := \mathcal{T}_\varphi \cap \mathsf{preTracks}(\mathcal{T}_1)$;
      $\mathcal{T}_0 := \mathcal{T}_1 \cup \mathcal{T}_0$;
      $i := i+1$;
    **end**;
    **return** $\mathcal{T}_0$;
**end function**
```

Figure 5.4: Symbolic Algorithms for Track-Exploration on a TKS $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$

```
function $\mathsf{StatesEU}^{[a,b]}(\mathcal{S}_\varphi, \mathcal{S}_\psi)$
  $\mathcal{T}_0 := \mathsf{MoveFront}(strong, a, \mathcal{S}_\varphi, \mathcal{S}_\psi \times \{1\})$;
  $\mathcal{T}_1 := \mathsf{MoveFront}(weak, b-a, \mathcal{S}_\varphi, \mathcal{T}_0)$;
  return $\{s \in \mathcal{S} \mid (s,1) \in \mathcal{T}_1\}$;
end function

function $\mathsf{StatesEU}^{\geq a}(\mathcal{S}_\varphi, \mathcal{S}_\psi)$
  $\mathcal{T}_0 := \mathsf{MoveFront}(strong, a, \mathcal{S}_\varphi, \mathcal{S}_\psi \times \{1\})$;
  $\mathcal{S}_0 := \{s \in \mathcal{S} \mid \exists t \in \mathbb{N}.(s,t) \in \mathcal{T}_0\}$;
  repeat
    $\mathcal{S}_1 := \mathcal{S}_0$;
    $\mathcal{S}_0 := \mathcal{S}_0 \cup (\mathcal{S}_\varphi \cap \mathsf{preStates}(\mathcal{S}_1))$ ;
  until $(\mathcal{S}_0 = \mathcal{S}_1)$;
  return $\mathcal{S}_0$;
end function

function $\mathsf{StatesEG}^{[a,b]}(\mathcal{S}_\varphi)$
  $\mathcal{R}_{run} := \mathcal{R}$;
  $\mathcal{R}_{\overline{X\varphi}} := \{(s,t,s') \mid s \in \mathcal{S} \wedge t \in \mathbb{N} \wedge s' \notin \mathcal{S}_\varphi\}$;
  $i = j := 0$;
  while $(i \leq b) \wedge (\mathcal{R}_{run} \neq \{\})$ do
    $t_{min} := min\{t \in \mathbb{N} \mid (s,t,s') \in \mathcal{R}_{run}\}$;
    $\mathcal{R}_{min} := \{(s,t,s') \in \mathcal{R}_{run} \mid t = t_{min}\}$;
    $j := i + t_{min}$;
    if $(b - i) < t_{min}$ then
      return $\{s \mid \exists s' \in \mathcal{S}.\exists t \in \mathbb{N}.(s,t,s') \in \mathcal{R}_{run}\}$;
    else
      $\mathcal{R}_{gt} := \{(s,t,s') \mid (s, t+t_{min}, s') \in \mathcal{R}_{run} \setminus \mathcal{R}_{min}\}$;
      if $(j \geq a) \wedge (j \leq b)$ then $\mathcal{R}_{min} := \mathcal{R}_{min} \setminus \mathcal{R}_{\overline{X\varphi}}$ endif;
      $\mathcal{S}_{succ} := \{s' \mid \exists s \in \mathcal{S}.\exists t \in \mathbb{N}.(s,t,s') \in \mathcal{R}_{min}\}$;
      $\mathcal{R}_{succ} := \mathcal{R} \cap \{(s,t,s') \mid s \in \mathcal{S}_{succ} \wedge t \in \mathbb{N} \wedge s' \in \mathcal{S}\}$;
      $\mathcal{R}_{new} := \{(s,t,s') \mid \exists s_1 \in \mathcal{S}.\exists t' \in \mathbb{N}.(s,t',s_1) \in \mathcal{R}_{min} \wedge (s_1,t,s') \in \mathcal{R}_{succ}\}$;
      $\mathcal{R}_{run} := \mathcal{R}_{gt} \cup \mathcal{R}_{new}$;
      $i := i + t_{min}$;
    end
  end
  return $\{s \mid \exists s' \in \mathcal{S}.\exists t \in \mathbb{N}.(s,t,s') \in \mathcal{R}_{run}\}$;
end function
```

Figure 5.5: Symbolic Algorithms for the Basic JCTL Operators

```
function States($\Phi$)
  case $\Phi$ of
    is_var($\Phi$)  : return $\{s \in \mathcal{S} \mid \Phi \in \mathcal{L}(s)\}$;
    $\neg\varphi$            : $\mathcal{S}_\varphi := \mathsf{States}(\varphi)$;
                      return $\mathcal{S} \setminus \mathcal{S}_\varphi$;
    $\varphi \wedge \psi$          : $\mathcal{S}_\varphi := \mathsf{States}(\varphi)$; $\mathcal{S}_\psi := \mathsf{States}(\psi)$;
                      return $\mathcal{S}_\varphi \cap \mathcal{S}_\psi$;
    $\mathsf{E}\underline{\mathsf{X}}^{[a+1,b]}\varphi$ : $\mathcal{S}_\varphi := \mathsf{States}(\varphi)$;
                      $\mathcal{S}_0 := \left\{ s \in \mathcal{S} \;\middle|\; \begin{array}{c} \exists s' \in \mathcal{S}_\varphi . \exists t \in [a+1, b]. \\ (s, t, s') \in \mathcal{R} \end{array} \right\}$;
                      return $\mathcal{S}_0$;
    $\mathsf{E}\underline{\mathsf{X}}^{\geq a+1}\varphi$  : $\mathcal{S}_\varphi := \mathsf{States}(\varphi)$;
                      $\mathcal{S}_0 := \left\{ s \in \mathcal{S} \;\middle|\; \begin{array}{c} \exists s' \in \mathcal{S}_\varphi . \exists t \geq a+1. \\ (s, t, s') \in \mathcal{R} \end{array} \right\}$;
                      return $\mathcal{S}_0$;
    $\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[a,b]}\ \psi]$ $\mathcal{S}_\varphi := \mathsf{States}(\varphi)$; $\mathcal{S}_\psi := \mathsf{States}(\psi)$;
                      return $\mathsf{StatesEU}^{[a,b]}(\mathcal{S}_\varphi, \mathcal{S}_\psi)$;
    $\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\geq a}\ \psi]$  $\mathcal{S}_\varphi := \mathsf{States}(\varphi)$; $\mathcal{S}_\psi := \mathsf{States}(\psi)$;
                      return $\mathsf{StatesEU}^{\geq a}(\mathcal{S}_\varphi, \mathcal{S}_\psi)$;
    $\mathsf{EG}^{[a,b]}\varphi$     : $\mathcal{S}_\varphi := \mathsf{States}(\varphi)$;
                      return $\mathsf{StatesEG}^{[a,b]}(\mathcal{S}_\varphi)$;
end function
```

Figure 5.6: Model Checking of JCTL Formulae on a TKS $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$

The algorithm starts considering all transitions $\mathcal{R}$ of the system and traverses forward through the state space determining the successor transitions $\mathcal{R}_{succ}$ of $\mathcal{R}$ and collecting them in $\mathcal{R}_{run}$. In each loop iteration and if the time is within the interval $[a, b]$, all transitions $\mathcal{R}_{\overline{X\varphi}}$ that lead to some states $\notin S_\varphi$ are being removed from $\mathcal{R}_{run}$ and their successors are not being collected further. This guaranties the integrity of φ in $[a, b]$. When the time is outside the interval $[a, b]$, all transitions are collected without restrictions, since the validity of their states is there trivially given (cf. defi nition 11). The algorithm terminates either when the time has reached the value b, or when the set of valid states has been emptied.

Hence, we obtain the following correctness result:

Theorem 1 (Correctness of Function States**)** *For any TKS $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ and any* JCTL *formula φ, the function* States *given in Figure 5.6 satisfies the equation* $\mathsf{States}(\varphi) = [\![\varphi]\!]_{\mathcal{K}}$.

5.2 Complexity of JCTL

In this section, we analyze the complexity of the presented model checking algorithm for JCTL. For this reason, we first note that there is a model checking procedure for CTL, e.g. the one given in [33], that runs in time $O(|\varphi|\,(|\mathcal{R}| + |\mathcal{S}|))$. This procedure is able to evaluate any CTL operator in time $O(|\varphi|\,(|\mathcal{R}| + |\mathcal{S}|))$.

It is easily seen that if we virtually expand a TKS to a UDS, the number of states and transition is multiplied with the maximum delay time $\hat{\tau}_{\mathcal{K}}$ that appears in $\mathcal{K}$. However, the runtime of the JCTL model checking procedure is not in $O(\hat{\tau}_{\mathcal{K}}\,|\varphi|\,(|\mathcal{R}|+|\mathcal{S}|))$, since the number of iterations does also depend on the time constraints of the temporal operators that may enforce more than $\hat{\tau}_{\mathcal{K}}\,|\mathcal{S}|$ iterations. The crucial part of our complexity analysis is the complexity of the MoveFront function. To this end, we note that the following holds:

Lemma 3 (Complexity of MoveFront **(I))** *Given a TKS $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$, a set of states $\mathcal{S}_\varphi$, and a set of tracks $\mathcal{T}_\psi$. Let moreover be $\mathcal{K}_e = (\mathcal{I}_e, \mathcal{S}_e, \mathcal{R}_e, \mathcal{L}_e)$ the virtually expanded structure of $\mathcal{K}$ according to definition 14 (cf. section 4.1), and φ and ψ formulae so that the equations $\mathcal{S}_\varphi \times \mathbb{N} \cap \mathcal{S}_e = [\![\varphi]\!]_{\mathcal{K}_e}$ and $\mathcal{T}_\psi = [\![\psi]\!]_{\mathcal{K}_e}$ hold. Then, we have*

$$\mathsf{MoveFront}(\sim, k, \mathcal{S}_\varphi, \mathcal{T}_\psi) = [\![\Phi(\sim, k, \varphi, \psi)]\!]_{\mathcal{K}_e}\,,$$

where the formula $\Phi(\sim, k, \varphi, \psi)$ is recursively defined as follows:

- $\Phi(\sim, 0, \varphi, \psi) = \psi$
- $\Phi(strong, k+1, \varphi, \psi) = \varphi \wedge \mathsf{EX}\Phi(strong, k, \varphi, \psi)$
- $\Phi(weak, k+1, \varphi, \psi) = \left(\begin{array}{l}\mathsf{let}\ \ y = \Phi(weak, k, \varphi, \psi)\\ \ \ \mathsf{in}\ \ y \vee (\varphi \wedge \mathsf{EX}y)\\ \mathsf{end}\end{array}\right)$

Moreover, if common subformulae are shared, it is easily seen that $|\Phi(\sim, k, \varphi, \psi)| \in O(k)$ holds. Hence, MoveFront$(strong, k, \mathcal{S}_\varphi, \mathcal{T}_\psi)$ *can be computed in time $O(min\{k, 2^{|\mathcal{S}_e|}\}(|\mathcal{R}_e| + |\mathcal{S}_e|))$, and* MoveFront$(weak, k, \mathcal{S}_\varphi, \mathcal{T}_\psi)$ *even in time $O(\min\{k, |\mathcal{S}_e|\}(|\mathcal{R}_e| + |\mathcal{S}_e|))$.*

The complexity can also be directly derived from the implementation of MoveFront: Note that the value of $\mathcal{T}_0$ monotonically grows in function calls of MoveFront$(weak, k, \ldots)$, but not for calls of MoveFront$(strong, k, \ldots)$. Therefore, MoveFront$(weak, k, \ldots)$ runs in time $O(\min\{k, |\mathcal{S}_e|\}(|\mathcal{R}_e| + |\mathcal{S}_e|))$, while MoveFront$(strong, k, \ldots)$ must check for disjunct processed fronts that can be at most $2^{|\mathcal{S}_e|}$ and hence requires time $O(min\{k, 2^{|\mathcal{S}_e|}\}(|\mathcal{R}_e| + |\mathcal{S}_e|))$. For this reason, we have the following result:

Theorem 2 (Complexity of JCTL**)** *For any TKS $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ and any* JCTL *formula φ, the function* States *given in Figure 5.6 runs in time $O(\hat{k}_\varphi\,|\varphi|\,\hat{\tau}_{\mathcal{K}}(|\mathcal{R}| + |\mathcal{S}|))$, where $\hat{\tau}_{\mathcal{K}} := \max\{t \mid \exists s, s'.(s, t, s') \in \mathcal{R}\}$ is the maximum delay time of $\mathcal{K}$, and $\hat{k}_\varphi$ is the maximal number used in time constraints in φ.*

The proof can be obtained by induction along the JCTL formulae. The induction steps are thereby obtained by the following facts, where $\mathsf{Time}(f)$ denotes the runtime of function f:

- $\mathsf{Time}(\mathsf{preStates}) \in O\big(\,|\mathcal{R}| + |\mathcal{S}|\,\big)$
- $\mathsf{Time}(\mathsf{preTracks}) \in O\big(\hat{\tau}_{\mathcal{K}}(|\mathcal{R}| + |\mathcal{S}|)\big)$
- $\mathsf{Time}(\mathsf{StatesEU}^{[a,b]}) \in O\big(k\hat{\tau}_{\mathcal{K}}(|\mathcal{R}| + |\mathcal{S}|)\big)$,
  where $k := \max\big\{\min\{a, 2^{\hat{\tau}_{\mathcal{K}}|\mathcal{S}|}\}, \min\{b-a, \hat{\tau}_{\mathcal{K}}\,|\mathcal{S}|\}\big\} \leq b$
- $\mathsf{Time}(\mathsf{StatesEU}^{\geq a}) \in O\big(\min\{a, 2^{\hat{\tau}_{\mathcal{K}}|\mathcal{S}|}\}\hat{\tau}_{\mathcal{K}}(|\mathcal{R}| + |\mathcal{S}|)\big)$
- $\mathsf{Time}(\mathsf{StatesEG}^{[a,b]}) \in O(k\hat{\tau}_{\mathcal{K}}(|\mathcal{R}| + |\mathcal{S}|))$, where $k \leq b$

Hence, all operators can be evaluated in time $O(\hat{k}_\varphi\hat{\tau}_{\mathcal{K}}(|\mathcal{R}| + |\mathcal{S}|))$. As a formula φ may contain $|\varphi|$ operators, the above theorem follows.

As we can see, one key to defi ne a more effi cient fragment of JCTL is to avoid calls of the form $\mathsf{MoveFront}(strong, k, \ldots)$, since this is not as effi cient as $\mathsf{MoveFront}(weak, k, \ldots)$. Using a specialized algorithm similar to the one given in [33], we can even improve the complexity for computing $\mathsf{MoveFront}(weak, k, \ldots)$:

Lemma 4 (Complexity of MoveFront **(II))** *Given a TKS $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$, a set of states $\mathcal{S}_\varphi$, and a set of tracks $\mathcal{T}_\psi$. Let moreover be $\mathcal{K}_e = (\mathcal{I}_e, \mathcal{S}_e, \mathcal{R}_e, \mathcal{L}_e)$ the virtually expanded structure of $\mathcal{K}$ according to definition 14 (cf. section 4.1), and φ and ψ formulae so that the equations $\mathcal{S}_\varphi \times \mathbb{N} \cap \mathcal{S}_e = [\![\varphi]\!]_{\mathcal{K}_e}$ and $\mathcal{T}_\psi = [\![\psi]\!]_{\mathcal{K}_e}$ hold. Then, there is an algorithm to compute* $\mathsf{MoveFront}(weak, k, \mathcal{S}_\varphi, \mathcal{T}_\psi)$ *in time $O(|\mathcal{R}_e| + |\mathcal{S}_e|)$.*

The specialized algorithm is similar to the one given in [33], but it needs to additionally take care of the lengths of the paths that have reached a certain track. For this reason, we maintain for any track $s_i \in \mathcal{S}_e$ a list $T_i = [s_{i,1},\ldots, s_{i,m_i}]$ such that $(s_{i,j}, s_i)$ is a transition in $\mathcal{R}_e$. Any track s_i will be marked with a number m_i later on that is the minimal length of a path to reach the set $\mathcal{T}_\psi$ from s_i. Furthermore, we create lists L_ℓ during the computations that contain the tracks that are marked with the number ℓ. The algorithm performs then the following steps:

Step 1**:** We eliminate all $s_{i,j}$ in each list T_i that do not belong to $\mathcal{T}_\psi$. This can be done in time $O(|\mathcal{R}_e|)$, since we look at each transition once.

Step 2**:** We mark the tracks $\mathcal{T}_\psi$ with the number 0, and list them in our fi rst list L_0. Let $\ell := 0$. This step is performed in time $O(|\mathcal{S}_e|)$, since we look at each track at most once.

Step 3**:** For each track s_i in L_ℓ, and each track $s_{i,j}$ of T_i, we mark $s_{i,j}$ with $\ell + 1$ if it is not already marked, and put $s_{i,j}$ in list $L_{\ell+1}$ in this case. We eliminate $s_{i,j}$ from T_i. We then increment ℓ, and repeat this step until $\ell = k$ holds or no transitions are left in the lists T_i.

Clearly, the repeated execution of step 3 does look at each transition at most once, so that $O(|\mathcal{R}_e|)$ is an upper bound for its complexity. Hence, we see that the entire algorithm runs in time $O(|\mathcal{R}_e| + |\mathcal{S}_e|)$.

Note that the above sketched algorithm works with a depth-fi rst search and therefore performs – in theory – better than breadth-fi rst searches like those used by symbolic model checking based on BDDs. However, in practice, the latter approaches are in general superior.

The next step is to defi ne a subset of JCTL that can be computed without calls of MoveFront $(strong, k, \cdot, \cdot)$. To this end, the interval constraints must be avoided, and therefore the logic must be restricted to the basic operators $\mathsf{EX}^{[a+1,b]}$, $\mathsf{EX}^{\geq a+1}$, $\mathsf{E}[\cdot\,\underline{\mathsf{U}}^{[0,a]}\,\cdot]$ and $\mathsf{EG}^{\geq a}$ only. Of course, we can still use all macro operators that can be defi ned from these basic operators like, for example, $\mathsf{E}[\varphi\,\underline{\mathsf{U}}^{\leq a}\,\psi] := \mathsf{E}[\varphi\,\underline{\mathsf{U}}^{[0,a]}\,\psi]$ or $\mathsf{EF}^{\leq a}\varphi := \mathsf{E}[1\,\underline{\mathsf{U}}^{\leq a}\,\varphi]$.

We defi ne the subset $\mathsf{JCTL}^{\leq}$ of JCTL as follows:

Definition 16 (The Fragment $\mathsf{JCTL}^{\leq}$ of JCTL) *Given a set of variables $\mathcal{V}$, the set of $\mathsf{JCTL}^{\leq}$ formulae is the least set satisfying the following rules, where φ and ψ denote arbitrary $\mathsf{JCTL}^{\leq}$ formulae, and $a, b \in \mathbb{N}$ are arbitrary natural numbers, and $\sim \in \{<, \leq, =, \geq, >\}$:*

- $\mathcal{V} \subseteq \mathsf{JCTL}^{\leq}$, *i.e, any variable is a $\mathsf{JCTL}^{\leq}$ formula*
- $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi \in \mathsf{JCTL}^{\leq}$
- $\mathsf{EX}^{[a+1,b]}\varphi \in \mathsf{JCTL}^{\leq}$
- $\mathsf{EX}^{\sim a+1}\varphi \in \mathsf{JCTL}^{\leq}$
- $\mathsf{EG}^{\geq a}\varphi \in \mathsf{JCTL}^{\leq}$ *and* $\mathsf{EG}^{> a}\varphi \in \mathsf{JCTL}^{\leq}$
- $\mathsf{E}[\varphi\,\underline{\mathsf{U}}^{<a}\,\psi] \in \mathsf{JCTL}^{\leq}$ *and* $\mathsf{E}[\varphi\,\underline{\mathsf{U}}^{\leq a}\,\psi] \in \mathsf{JCTL}^{\leq}$
- $\mathsf{EF}^{<a}\varphi \in \mathsf{JCTL}^{\leq}$ *and* $\mathsf{EF}^{\leq a}\varphi \in \mathsf{JCTL}^{\leq}$

As each temporal operator of $\mathsf{JCTL}^{\leq}$ can be evaluated in time $O(|\mathcal{R}_e| + |\mathcal{S}_e|)$, i.e., $O(\hat{\tau}_{\mathcal{K}}(|\mathcal{R}| + |\mathcal{S}|))$, we immediately have the following result:

Theorem 3 (Complexity of $\mathsf{JCTL}^{\leq}$) *For any TKS $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ and any $\mathsf{JCTL}^{\leq}$ formula φ, there is an algorithm to compute $[\![\varphi]\!]_{\mathcal{K}}$ in time $O(|\varphi|\,\hat{\tau}_{\mathcal{K}}(|\mathcal{R}| + |\mathcal{S}|))$, where $\hat{\tau}_{\mathcal{K}}$ is defined as in theorem 2.*

In comparison to the complexity of JCTL as given in theorem 2, the factor $\hat{k}_{\varphi}$ disappeared. This is due to the fact that the temporal operators that belong to $\mathsf{JCTL}^{\leq}$ are all monotonic, so that the iterations can stop at least when all tracks have been visited once.

Chapter 6

Translating Synchronous Programs to Real-Time Models

Χρόνος δέ φευγέτω σε μηδέ εἷς ἀργός. [1]

ΙΠΠΩΝΑΚΤΟΣ (περί τό 550 π.Χ.)

(Ανϑ. Στοβ. ΚΘ, 42)

In this section, we present techniques that allow the translation of synchronous programs to timed Kripke structures. The obtained models can then directly be used for architecture-dependent and -independent real-time verification purposes, as well as for further analysis purposes like WCET and BCET, using techniques like the ones presented in section 7.

To this end, we take advantage of the distinction between micro and macro steps, offered by the usage of synchronous languages (cf. sections 1.1.2 and 2.2). This distinction simplifies the analysis of the runtime behavior of programs, since the macro steps can be directly used as ‘building blocks’. Building blocks are thereby parts of the program with a fixed runtime (independent of particular inputs). For this reason, building blocks must not contain data-dependent loops or conditional statements. While it is sufficient for sequential code to simply take the greatest substatements that do not contain data-dependent loops or conditional statements, this becomes more difficult for programs with concurrency and mechanisms like process preemption and suspension. The important advantage of synchronous languages for our purposes is therefore that compilers do automatically determine the building blocks in terms of macro steps.

In early design phases, if the complete realization of a system is not yet known, one can not argue about physical time, since this depends on the hardware chosen for the realization. Nevertheless, using synchronous languages, one can achieve realization independent descriptions of the system, which makes possible to *reason about time at a logical level*.

Of course, it is also of essential importance to guarantee that the real-time specifications are met with respect to *physical time*. Reasoning about physical time however, requires to postpone

[1] Don’t let time escape from you being unalert.
IPPONAKTOS (fl 550 BC)

the analysis to later design phases where the hardware/software partitioning and the choice of the used microprocessors are made. The first consequence when considering architecture-dependent execution time of systems described by means of synchronous programs, is that, in contrast to the model at logical level, the finite number of micro steps included in each macro step of the architecture-dependent model *will consume time*, which is not always equal.
It is viewed as a good programming style when the actual runtime of the macro steps is balanced, but generally, the macro steps will not require the same amount of physical time.

As a natural order, we will consider in the next sections first the high-level- and then the low-level approach. It is important to note that our methods allow in both cases the generation of real-time models without having the need of parallel composition of single submodels, which is one of the most important drawbacks of other approaches, as described in sections 1.1.4 and 1.1.5. Parallel composition is already performed during compilation of the synchronous program, before the generation of the single real-time model.

6.1 Timed Kripke Structures with Logical Time

In order to translate synchronous programs to high-level real-time models, we introduce in this section an extension of synchronous languages together with translation techniques to obtain abstract real-time models (in the sense of definition 7). This extension allows the programmer to declare irrelevant program locations and hence perform abstractions. After the usual compilation of a program $\mathcal{P}$ into a UDS $\mathcal{K}_{\mathcal{U}}$, certain states are thus to be ignored. For this purpose, an efficient algorithm is proposed in section 6.1.2 to generate a *single TKS* as a *unique* real-time model $\mathcal{K}_{\mathcal{R}}$, by ignoring the irrelevant states, while retaining the quantitative information. In section 6.1.3, we then present the relationship between the TKS $\mathcal{K}_{\mathcal{R}}$ and the corresponding UDS $\mathcal{K}_{\mathcal{U}}$. The algorithms presented in section 5.1 can then be used to check quantitative temporal properties of the generated TKS.

Our goal is to show how abstractions can be incorporated in synchronous programs to obtain abstract high-level real-time models that retain the quantitative temporal information. In particular, the programmer can generate abstract real-time models without having the need of special knowledge about verification techniques. A well-known problem of all approaches based on abstraction techniques is that the chosen abstraction might be too coarse. In this case, our technique is able to detect the problematic program locations.

6.1.1 Extending Synchronous Languages

Using the available algorithms for compiling Esterel and Quartz, one can compile every program into an equivalent sequential program. If only finite data types were used, then additionally hardware circuits and UDSs can be generated. This is sufficient for code generation and for the verification of temporal properties.

To enhance the efficiency of the verification, it is often advantageous to omit irrelevant details so that the generated formal models are as small as possible. In the opinion of the author, the choice between relevant and irrelevant program locations must be left to the programmer. For this reason, we propose a new statement to explicitly mark these locations by emitting a special signal δ. To this end, we introduce a new macro statement of the form **abstract** S **end** that can be defined as follows:

$$\textbf{abstract } S \textbf{ end} :\equiv \left(\begin{array}{l}\textbf{local } t \textbf{ in}\\ \quad S; \textbf{emit } t\\ \quad \|\\ \quad \textbf{abort}\\ \quad\quad \textbf{loop}\\ \quad\quad\quad \ell : \textbf{pause};\\ \quad\quad\quad \textbf{emit } \delta\\ \quad\quad \textbf{end loop}\\ \quad \textbf{when immediate } t\\ \textbf{end local}\end{array}\right)$$

Hence, **abstract** S **end** behaves like S, but additionally emits the variable δ whenever the control flow moves inside S. The entering and termination transitions (from and to S) do not emit δ. The above definition has however the drawback that an additional **pause** statement and an additional local signal t are used. This additional overhead can be circumvented by the following alternative definitions:

- $\mathsf{inside}\,(\textbf{abstract } S \textbf{ end}) :\equiv \mathsf{inside}\,(S)$
- $\mathsf{instant}\,(\textbf{abstract } S \textbf{ end}) :\equiv \mathsf{instant}\,(S)$
- $\mathsf{enter}\,(\textbf{abstract } S \textbf{ end}) :\equiv \mathsf{enter}\,(S)$
- $\mathsf{terminate}\,(\textbf{abstract } S \textbf{ end}) :\equiv \mathsf{terminate}\,(S)$
- $\mathsf{move}\,(\textbf{abstract } S \textbf{ end}) :\equiv \mathsf{move}\,(S)$
- $\mathsf{guardcmd}\,(\varphi, \textbf{abstract } S \textbf{ end})$
  $:\equiv \mathsf{guardcmd}\,(\varphi, S) \cup \{(\mathsf{inside}\,(S) \wedge \neg\mathsf{terminate}\,(S), \textbf{emit } \delta)\}$

Hence, the control flows of **abstract** S **end** and S are the same, and the data flow differs only in that **abstract** S **end** additionally emits the variable δ whenever the control flow moves inside S. Using this direct definition via the semantics given in [112, 114] instead of the above macro expansion, we circumvent the use of the additional local variable t and the additional **pause** statement.

Using our new statement **abstract** S **end**, it is easily seen that the following statements were equivalent for any $n \in \mathbb{N}$ and any Boolean condition σ:

- $\textbf{abstract } \ell : \textbf{await } \sigma \textbf{ end} \equiv \left(\begin{array}{l}\textbf{do}\\ \quad \ell : \textbf{pause};\\ \quad \textbf{if } \neg\sigma \textbf{ then emit } \delta \textbf{ end}\\ \textbf{while } \neg\sigma\end{array}\right)$

- **abstract** ℓ : **await** n **end** $\equiv$

```
(  local c in
     c := 0;
     do
       c := delayed c + 1;
       ℓ : pause;
       if c ≠ n then
         emit δ
       end
     while c ≠ n
   end local  )
```

The abstract **await** statements are important to model delays. As can be seen, these statements can be easily defined in terms of existing Esterel/Quartz statements so that existing tools can be used for their compilation. Hence, using the available compilers, we obtain a transition system where each irrelevant state is marked with the special variable δ. Note that **await** n will definitely terminate after n macro steps, while the termination of **await** σ is not guaranteed unless we could guarantee that σ will eventually hold. The termination of abstract statements must be considered in the generation of TKSs as described in the next section.

6.1.2 Generating the High-Level Model

We now consider the generation of a TKS from a given Quartz program. For this purpose, we assume that we already have a function QuartzCompileUDS that computes an equivalent unit delay structure (UDS) $\mathcal{K}_{\mathcal{U}}$ of a given Quartz program $\mathcal{P}$. Such a function is essentially implemented by any compiler, like the one described in [111, 112, 114, 116]. To finally obtain an equivalent TKS $\mathcal{K}_{\mathcal{R}}$, it is therefore sufficient to be able to compute a corresponding TKS from a given UDS where certain states are marked to be irrelevant. In the following, these irrelevant states are labeled by the variable δ.

The overall idea is to replace finite paths $s_0 \to s_1 \to \ldots \to s_{n-1} \to s_n$ of states where $s_1, \ldots, s_{n-1}$ were labeled with δ, but s_0 and s_n were not labeled with δ, by single transitions $s_0 \xrightarrow{n} s_n$. This will generate a TKS, where the quantitative properties of the UDS are preserved.

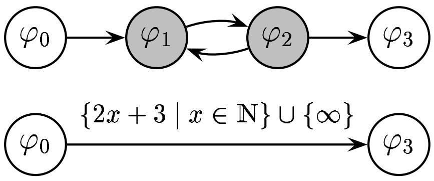


Figure 6.1: Impossibility of TKS Generation

However, if a cycle of states labeled with δ is reachable, then the generation of the corresponding TKS transition can not be performed: If such a cycle is between two states s and s' that are not labeled with δ, then this means that there would exist infinitely many transitions in the TKS between these states (cf. Figure 6.1). Such cycles arise when too large parts of the program are embraced within an **abstract** statement.

For this reason, we have to check for reachable cycles of states that are labeled with δ. To be concise in the following, we call states labeled with δ simply 'δ-states', and cycles consisting of δ-states 'δ-cycles'.

Given a transition relation $\mathcal{U}$ of a UDS $\mathcal{K}_{\mathcal{U}}$, the function Deadends shown in Figure 5.1 eliminates all finite paths contained in $\mathcal{U}$ and returns a finite-path-free transition relation. This is done by computing all states in $\mathcal{K}_{\mathcal{U}}$ that have at least one successor state.

The function Reach clearly computes the set of states that are reachable from the states $\mathcal{S}_{\varphi}$ by the transitions of $\mathcal{U}$.

Consider now the overall translation of a Quartz program $\mathcal{P}$ to an equivalent TKS, as given in Figure 6.2. We first compute the corresponding UDS $\mathcal{U}$ and their initial states $\mathcal{I}$ with the function QuartzCompileUDS as given in [111, 112, 114, 116]. Our next task is to check whether there is a reachable δ-cycle. For this reason, we restrict the transitions to δ-states, thus obtaining $\mathcal{U}_{\delta}$, and compute the transition relation $\mathcal{U}_{\delta-cycles}$ that has only infinite paths of δ-states. If $\mathcal{U}_{\delta-cycles} = \{\}$ holds, then we have no δ-cycles at all, since all δ-states have only finite paths. In this case, we can compute the desired TKS by calling $\mathsf{Chronos}(\mathcal{U}, \mathcal{S}_{\delta})$. We will discuss that function below.

On the other hand, if $\mathcal{U}_{\delta-cycles} \neq \{\}$ holds, then in $\mathcal{U}_{\delta}$ occurs at least one δ-cycle. There is still a chance to generate a TKS, namely if none of these states is reachable. For this reason, we next compute the set of reachable states $\mathcal{S}_{reach}$, and check if a δ-cycle is included in $\mathcal{S}_{reach}$. If no δ-cycle is reachable, then we can generate the TKS by calling Chronos with the transitions restricted to reachable states.

Finally, if there is a reachable δ-cycle, then the construction of some TKS transitions is not possible (cf. Figure 6.1), and therefore an exception is raised where the transition relation $\mathcal{U}_{\delta-cycles}$ of the reachable δ-cycles is returned. With this information, the programmer can identify the program locations that lead to the too coarse abstraction. In this case, there are two possible solutions: either the programmer proceeds with the verification at the UDS level, or the programmer has to weaken the abstraction. We consider the first case in section 6.1.3 in more detail. In the second case, one must consider that the specifications must also be adapted.

Now, consider how Chronos works. Recall, that we may assume that the transition relation $\mathcal{U}$ does not contain any δ-cycle, when Chronos is called. We first compute the transitions between all states s and s' that are not labeled with δ. These transitions are labeled with time duration 1. To compute the further transitions, we have to compute the transitions $\mathcal{U}_{out}$ that leave a δ-sequence. In the following loop, we have as an invariant that the timed transitions with duration $< \tau$ have already been computed, and that $\mathcal{U}_{out}$ contains transitions (s, s') such that s' is not a δ-state, and that there will be a timed transition leading to s' with duration $\geq \tau$.

```
function Chronos($\mathcal{U}, \mathcal{S}_\delta$)
  $\mathcal{R} := \{(s, 1, s') \mid (s, s') \in \mathcal{U} \wedge \{s, s'\} \cap \mathcal{S}_\delta = \{\}\}$;
  $\mathcal{U}_{out} := \{(s, s') \in \mathcal{U} \mid s \in \mathcal{S}_\delta \wedge s' \notin \mathcal{S}_\delta\}$;
  $\tau := 1$;
  repeat
    $\tau := \tau + 1$;
    $\mathcal{U}_0 := \{(s, s') \mid \exists s_1.(s, s_1) \in \mathcal{U} \wedge (s_1, s') \in \mathcal{U}_{out}\}$;
    $\mathcal{R} := \mathcal{R} \cup \{(s, \tau, s') \mid (s, s') \in \mathcal{U}_0 \wedge s \notin \mathcal{S}_\delta\}$;
    $\mathcal{U}_{out} := \{(s, s') \in \mathcal{U}_0 \mid s \in \mathcal{S}_\delta\}$;
  until $\mathcal{U}_{out} = \{\}$;
  return $\mathcal{R}$;
end function

function QuartzCompileHTKS($\mathcal{P}$)
  $(\mathcal{I}, \mathcal{S}, \mathcal{U})$ := QuartzCompileUDS($\mathcal{P}$);
  $\mathcal{S}_\delta := \{s \in \mathcal{S} \mid \delta \in \mathcal{L}(s)\}$;
  $\mathcal{U}_\delta := \mathcal{U} \cap (\mathcal{S}_\delta \times \mathcal{S})$;
  $\mathcal{U}_{\delta-cycles}$ := Deadends($\mathcal{U}_\delta$);
  if $\mathcal{U}_{\delta-cycles} = \{\}$ then
    return Chronos($\mathcal{U}, \mathcal{S}_\delta$)
  else
    $\mathcal{S}_{reach}$ := Reach($\mathcal{U}, \mathcal{I}$);
    $\mathcal{U}_{\delta-cycles} := \mathcal{U}_{\delta-cycles} \cap (\mathcal{S}_{reach} \times \mathcal{S})$;
    if $\mathcal{U}_{\delta-cycles} = \{\}$ then
      $\mathcal{S}_\delta := \mathcal{S}_\delta \cap \mathcal{S}_{reach}$;
      $\mathcal{U}_{reach} := \{(s, s') \in \mathcal{U} \mid s \in \mathcal{S}_{reach}\}$;
      return Chronos($\mathcal{U}_{reach}, \mathcal{S}_\delta$)
    else
      raise exception AbstractionTooCoarse($\mathcal{U}_{\delta-cycles}$);
    end
  end
end function
```

Figure 6.2: Algorithms for Generation of High-Level TKSs

To compute the timed transitions with duration $\tau + 1$, we consider the δ-states s that are connected by a $\mathcal{U}$-transition to a state s_1, such that $(s_1, s') \in \mathcal{U}_{out}$ holds. Then, the transition (s, τ, s') is added to $\mathcal{R}$. If, on the other hand, s is labeled with δ, then this transition will be extended in a later loop iteration until no δ-state remains.

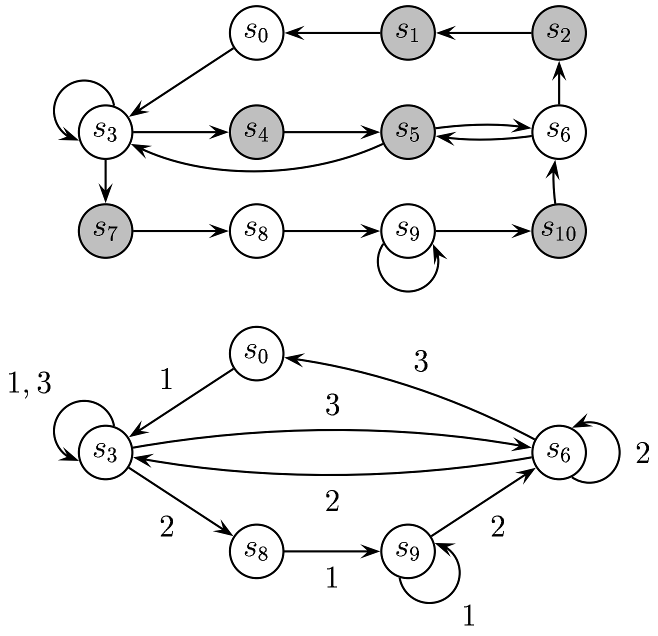


Figure 6.3: A UDS with its Corresponding TKS

It is easily seen that the loop in Chronos will be repeated at most d_{len} times, where d_{len} is the maximal length of a fi nite sequence of δ-states. An example for the execution of Chronos is given in Figure 6.3. The different loop iterations for the UDS given in the upper half of Figure 6.3 are as follows:

- $\mathcal{U}^{(1)}_{out} = \{(s_1, s_0), (s_5, s_3), (s_5, s_6), (s_7, s_8), (s_{10}, s_6)\}$
- $\mathcal{R}^{(1)} = \{(s_3, 1, s_3), (s_8, 1, s_9), (s_9, 1, s_9), (s_0, 1, s_3)\}$
- $\mathcal{U}^{(2)}_{out} = \{(s_2, s_0), (s_4, s_3), (s_4, s_6),$
  $(s_6, s_6), (s_3, s_8), (s_9, s_6), (s_6, s_3)\}$
- $\mathcal{R}^{(2)} = \{(s_6, 2, s_6), (s_3, 2, s_8), (s_9, 2, s_6), (s_6, 2, s_3)\}$
- $\mathcal{U}^{(3)}_{out} = \{(s_6, s_0), (s_3, s_6), (s_3, s_3)\}$
- $\mathcal{R}^{(3)} = \{(s_6, 3, s_0), (s_3, 3, s_6), (s_3, 3, s_3)\}$

At the end, we obtain the TKS given in the lower part of Figure 6.3.

6.1.3 Verifying Real-Time Properties at UDS Level

Recall that our task is to check for a given program $\mathcal{P}$ whether a temporal property Φ holds. To describe real-time temporal properties, we use the CTL real-time extension JCTL (cf. section 3) that is defined on TKSs.

Using traditional compilers, it is possible to compute for a given program $\mathcal{P}$ the corresponding UDS $\mathcal{K}_{\mathcal{U}}$, so that temporal properties can be checked for the program. The algorithm given in the previous section allows furthermore to compute a TKS $\mathcal{K}_{\mathcal{R}}$ for $\mathcal{P}$ to increase the efficiency of the verification. Clearly, the structures $\mathcal{K}_{\mathcal{U}}$ and $\mathcal{K}_{\mathcal{R}}$ are different and therefore satisfy different formulae. In this section, we explain the relationship between the verification at the TKS- and at the UDS level. For this purpose, we define a function Θ_δ that computes for a JCTL formula Φ a corresponding JCTL formula $\Theta_\delta(\Phi)$ that holds on the UDS $\mathcal{K}_{\mathcal{U}}$ iff Φ holds on the TKS $\mathcal{K}_{\mathcal{R}}$.

Definition 17 *Given a* JCTL *formula* Φ *and a variable* δ*, we define a corresponding* JCTL *formula* $\Theta_\delta(\Phi)$ *as follows:*

- $\Theta_\delta(x) :\equiv x$ *for variables* x
- $\Theta_\delta(\neg\varphi) :\equiv \neg\Theta_\delta(\varphi)$
- $\Theta_\delta(\varphi \wedge \psi) :\equiv \Theta_\delta(\varphi) \wedge \Theta_\delta(\psi)$
- $\Theta_\delta(\varphi \vee \psi) :\equiv \Theta_\delta(\varphi) \vee \Theta_\delta(\psi)$
- $\Theta_\delta(\mathsf{E}\underline{\mathsf{X}}^\kappa\varphi) :\equiv \mathsf{E}[\Theta_\delta(\varphi)\ \underline{\mathsf{XW}}^\kappa\ (\neg\delta)]$
- $\Theta_\delta(\mathsf{A}\underline{\mathsf{X}}^\kappa\varphi) :\equiv \mathsf{A}[\Theta_\delta(\varphi)\ \underline{\mathsf{XW}}^\kappa\ (\neg\delta)]$
- $\Theta_\delta(\mathsf{E}[\varphi\ \underline{\mathsf{U}}^\kappa\ \psi]) :\equiv \mathsf{E}[(\delta \vee \Theta_\delta(\varphi))\ \underline{\mathsf{U}}^\kappa\ (\neg\delta \wedge \Theta_\delta(\psi))]$
- $\Theta_\delta(\mathsf{E}[\varphi\ \mathsf{U}^\kappa\ \psi]) :\equiv \mathsf{E}[(\delta \vee \Theta_\delta(\varphi))\ \mathsf{U}^\kappa\ (\neg\delta \wedge \Theta_\delta(\psi))]$
- $\Theta_\delta(\mathsf{A}[\varphi\ \underline{\mathsf{U}}^\kappa\ \psi]) :\equiv \mathsf{A}[(\delta \vee \Theta_\delta(\varphi))\ \underline{\mathsf{U}}^\kappa\ (\neg\delta \wedge \Theta_\delta(\psi))]$
- $\Theta_\delta(\mathsf{A}[\varphi\ \mathsf{U}^\kappa\ \psi]) :\equiv \mathsf{A}[(\delta \vee \Theta_\delta(\varphi))\ \mathsf{U}^\kappa\ (\neg\delta \wedge \Theta_\delta(\psi))]$

The semantics is given in section 3, except for $\mathsf{E}[\varphi\ \underline{\mathsf{XW}}^\kappa\ \psi]$. $\mathsf{E}[\varphi\ \underline{\mathsf{XW}}^\kappa\ \psi]$ holds in a state s iff there is a path starting in s such that at some position (different from the starting one) of the path ψ holds, and at the first such position (different from the starting one) φ holds, and the time required to reach this first position satisfies the time constraint κ.

$\Theta_\delta(\Phi)$ is of length $O(|\Phi|)$, i.e., there is only a linear blow-up. The above definition is used to transform a given JCTL Φ to another corresponding JCTL formula $\Theta_\delta(\Phi)$ such that only states not labeled with δ are considered for evaluation of $\Theta_\delta(\Phi)$. If no state is labeled with δ, then it is easily seen that Φ and $\Theta_\delta(\Phi)$ were equivalent. The following theorem reveals the entire relationship between Φ and $\Theta_\delta(\Phi)$:

Theorem 4 (Relationship between UDS and TKS) *Given a UDS* $\mathcal{K}_{\mathcal{U}}$ *such that the corresponding TKS* $\mathcal{K}_{\mathcal{R}}$ *exists, then we have for any* JCTL *formula* Φ *and any state* s *of* $\mathcal{K}_{\mathcal{R}}$ *the following relationship:*

$$(\mathcal{K}_{\mathcal{U}}, s) \models \Theta_\delta(\Phi) \quad \textit{iff} \quad (\mathcal{K}_{\mathcal{R}}, s) \models \Phi$$

The theorem is easily proved by an induction on the structure of Φ. The essential property for the induction steps is thereby that (by construction of $\mathcal{K}_{\mathcal{R}}$) a transition $s \xrightarrow{t} s'$ in $\mathcal{K}_{\mathcal{R}}$ corresponds to a sequence of transitions $s \to s_1 \to \ldots \to s_{n-1} \to s'$ of states where $s_1, \ldots, s_{n-1}$ are labeled with δ (of course, s and s' are not labeled with δ, since they belong to $\mathcal{K}_{\mathcal{R}}$).

The above theorem precisely states that if an equivalent TKS $\mathcal{K}_{\mathcal{R}}$ exists, then we have the choice between checking $(\mathcal{K}_{\mathcal{U}}, s) \models \Theta_{\delta}(\Phi)$ or $(\mathcal{K}_{\mathcal{R}}, s) \models \Phi$. Both model checking problems are equivalent to each other.

6.2 Timed Kripke Structures with Physical Time

In section 6.1 it has been shown how macro steps can be combined to larger steps in that parts of synchronous programs are marked as 'uninteresting' for the system's verifi cation. This means that the macro steps of these parts are combined into a building block and that the numbers of macro steps to execute these blocks are counted. This extension does not affect the code generation, i.e., the processes still run in lockstep with respect to their macro steps. Thus, small abstract transition systems can be obtained from synchronous programs to verify required real-time specifi cations.

In this section, we present a new technique that performs an exact low-level (architecture-dependent) runtime analysis of synchronous programs and generates low-level timed Kripke structures, whose transitions are labeled with numbers denoting the exact execution times of the macro steps that are related to these transitions. These TKSs can then be directly used for architecture-dependent real-time verifi cation, as well as for further analysis purposes like WCET and BCET, using techniques like the ones presented in section 7.

Our goal is to directly generate executable code out of synchronous programs, and develop techniques to construct low-level timed Kripke structures with respect to the execution times required for the steps of the executable code. Note that the executable code can already have been verifi ed at a logical level.

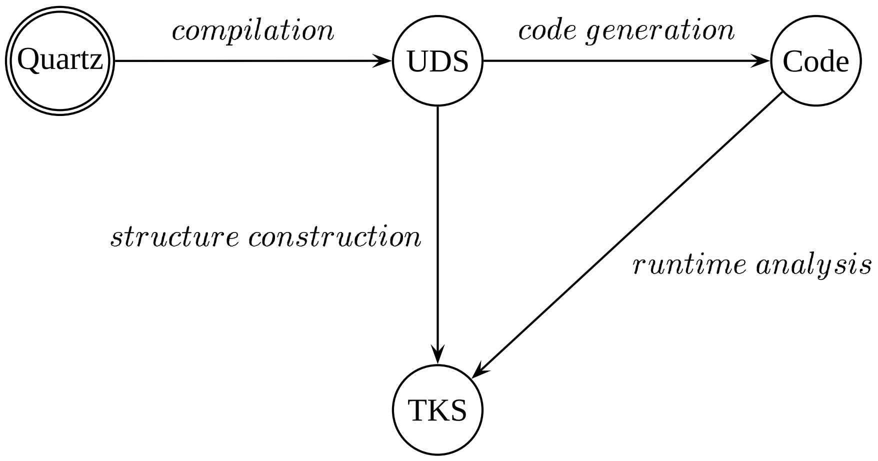


Figure 6.4: Generation of Low-Level Timed Kripke Structures

For this purpose, we fi rst construct a transition system (as UDS) and obtain executable code for the given synchronous program (cf. Fig. 6.4). The generated code is then being embedded in an environment in order to perform exact and detailed low-level runtime analysis, i.e. to determine and capture the execution times required for *all actions* of the code. This is done during the execution of the code. Simultaneously, our method constructs a low-level timed Kripke structure by labeling the transitions of the system by the determined execution times of the corresponding code actions for the given microprocessor.

Our approach takes advantage of established symbolic techniques to effi ciently manipulate large fi nite state transition systems by means of binary decision diagrams (BDDs) [17].

Note further that the generated transition systems have timed transitions that correspond to *non-interruptible atomic actions*. For verifi cation purposes of such systems, the real-time temporal logic JCTL (cf. section 3) can be considered.

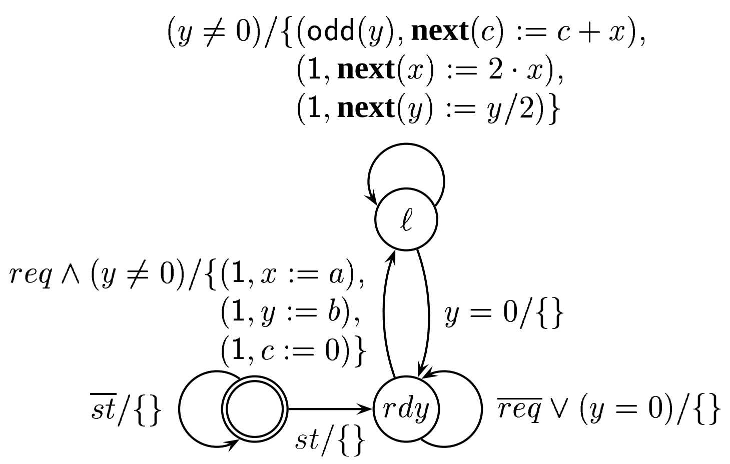
In the next sections we fi rst discuss the generation of executable code. Then we give techniques to effi ciently perform exact low-level runtime analysis consider also native CPU-instructions in order to obtain low-level real-time formal models. The overall flow is shown in Fig. 6.4.

6.2.1 Exact Low-Level Runtime Analysis: Code Generation

Our method to generate code for synchronous programs considers *equation systems based on hardware synthesis*, i.e. the automaton representation obtained from a synchronous program [112]. The method is based on the encoding of the states with Boolean state variables.

```
module RussMult :
  input $req, a : \mathbb{I}[n], b : \mathbb{I}[n]$;
  output $c : \mathbb{I}[n]$;
  local $x : \mathbb{I}[n], y : \mathbb{I}[n]$
  label $rdy$;
    loop
      $rdy$ : await $req$;
      $x := a; y := b; c := 0$;
      while $y \neq 0$ do
        if $\mathsf{odd}(y)$ then next$(c) := c + x$ end;
        next$(x) := 2 \cdot x$;
        next$(y) := y/2$;
        $\ell$ : pause
      end while
    end loop
end module
```

Figure 6.5: Russian Multiplication



$(y \neq 0)/\{(\mathsf{odd}(y), \textbf{next}(c) := c + x),$
$(1, \textbf{next}(x) := 2 \cdot x),$
$(1, \textbf{next}(y) := y/2)\}$
ℓ
$req \wedge (y \neq 0)/\{(1, x := a),$
$(1, y := b),$
$(1, c := 0)\}$
$y = 0/\{\}$
$\overline{st}/\{\}$
rdy
$st/\{\}$
$\overline{req} \vee (y = 0)/\{\}$

Figure 6.6: Semantics of Module RussMult

As an example, consider the Quartz program given in Figure 6.5 (it implements a Russian multiplication algorithm). The semantics is the transition system given in Figure 6.6.

The three states correspond to the situations where the control flow is either outside the program or at one of the locations labeled with ℓ or rdy. The labels of the transitions are of the form $\Phi/\{(\gamma_1, \alpha_1), \ldots, (\gamma_n, \alpha_n)\}$ with the following meaning: the transition can be taken iff the condition Φ holds at that point of time. Taking the transition means that those assignments or signal emissions α_i are executed whose guard γ_i holds at that point of time.

The automaton of the previous example yields state transition equations like the ones shown in Figure 6.7.

$$
\begin{aligned}
&\textbf{next}(rdy) := \left(\begin{array}{l} \neg rdy \wedge \neg \ell \wedge st \vee \\ rdy \wedge (\neg req \vee (y = 0)) \vee \\ \ell \wedge (y = 0) \end{array}\right) \\
&\textbf{next}(\ell) := (rdy \wedge req \vee \ell) \wedge (y \neq 0) \\
&\textbf{next}(x) := \left(\begin{array}{l} \textbf{if } \ell \wedge (y = 0) \wedge \mathsf{odd}(y) \textbf{ then } 2 \cdot x \\ \vdots \end{array}\right) \\
&\textbf{next}(y) := \ldots \\
&\textbf{next}(c) := \ldots
\end{aligned}
$$

Figure 6.7: Code Generation Using Equation Systems Based on Hardware Synthesis

It is straightforward to generate sequential code (e.g. C-code) from the above state transition equations. We simply put the assignments in a nonterminating loop (and use the C-syntax, of course). Some problems of synchronous languages like causality have to be checked here, but these problems have already found good solutions [15, 9], so we do not consider this issue here. The size of the generated code is very small (it is in practice linear in terms of the given synchronous program). This is an important advantage in praxis, in particular for applications involved in embedded systems, where the memory size is usually limited.

The exact runtime analysis of single instructions used in a sequential program is a complicated task due to the complex interaction of different cache hierarchies: The execution time depends not only on its operands but also on the fact that the needed data might either be available in caches, or have to be requested through slower channels. We handle this problem as follows:

Our generated program is one static block which is executed in an endless loop. The writing to cache data and also the execution of instructions is performed in the same order in each loop. By executing this block several times for a given input we obtain a cache-configuration which is very similar to the configuration in a real environment. A runtime-analysis for this block can then be directly performed by measuring the time for the execution of the static block without a deeper analysis of the cache-structure. Furthermore, there already exist successful methods like [65] in order to handle this problem. Techniques like the ones presented in [65] can be easily endowed in our tool and are part of our current implementation work.

6.2.2 Exact Low-Level Runtime Analysis: Extending Synchronous Languages to Consider Native CPU Instructions

Model-checking algorithms used for formal verification purposes usually consider combinations of boolean operators for expressing mathematical calculations, i.e. arithmetic operations are mapped to the boolean level and are implemented as bit-operations on the microprocessors. Unfortunately, this has the disadvantage, that the information of the mathematical function is lost in the resulting transition system.

For the generation of executable code which benefits of the native mathematical operators of the target CPU, it is necessary to avoid the conversion of mathematical operators to their logical equivalents. For this purpose, we introduce a new technique that constructs a UDS, which preserves mathematical operations as atomic actions. After that, the transitions can be easily labeled by their physical execution times required by the target CPU.

For this purpose, we introduce an extension of synchronous languages in order to represent atomic execution of specific program locations. Similar to the extension proposed in section 6.1.1, we introduce a new macro statement of the form **exclusive** ζ **in** S **end** that can be defined as follows:

$$\textbf{exclusive}\ \zeta\ \textbf{in}\ S\ \textbf{end} :\equiv \left(\begin{array}{l}\textbf{local}\ t\ \textbf{in}\\ \quad S; \textbf{emit}\ t\\ \quad \|\\ \quad \textbf{abort}\\ \quad\quad \textbf{loop}\\ \quad\quad\quad \ell : \textbf{pause};\\ \quad\quad\quad \textbf{emit}\ \zeta\\ \quad\quad \textbf{end loop}\\ \quad \textbf{when immediate}\ t\\ \textbf{end local}\end{array}\right)$$

The key idea of this extension is to remove the parallelism which can't exist on a target microprocessor due to the fact that arithmetic operations are executed as non interruptible atomic actions on a target machine. Similar to section 6.1.1, our technique leads to 'hyper steps' that correspond to one transition of the transition system, but contain a fi nite number of intermediate macro steps, which are marked, so that a post-processing step can replace them by transitions.

Hence, **exclusive** ζ **in** S **end** behaves like S, but additionally emits the variable ζ whenever the control flow moves inside S. The entering and termination transitions (from and to S) do not emit ζ. The above defi nition has however the drawback that an additional **pause** statement and an additional local signal t are used. This additional overhead can be circumvented by the following alternative defi nitions:

- $\mathsf{inside}\,(\textbf{exclusive}\ \zeta\ \textbf{in}\ S\ \textbf{end}) :\equiv \mathsf{inside}\,(S)$
- $\mathsf{instant}\,(\textbf{exclusive}\ \zeta\ \textbf{in}\ S\ \textbf{end}) :\equiv \mathsf{instant}\,(S)$
- $\mathsf{enter}\,(\textbf{exclusive}\ \zeta\ \textbf{in}\ S\ \textbf{end}) :\equiv \mathsf{enter}\,(S)$
- $\mathsf{terminate}\,(\textbf{exclusive}\ \zeta\ \textbf{in}\ S\ \textbf{end}) :\equiv \mathsf{terminate}\,(S)$
- $\mathsf{move}\,(\textbf{exclusive}\ \zeta\ \textbf{in}\ S\ \textbf{end}) :\equiv \mathsf{move}\,(S)$
- $\mathsf{guardcmd}\,(\varphi, \textbf{exclusive}\ \zeta\ \textbf{in}\ S\ \textbf{end})$
  $:\equiv \mathsf{guardcmd}\,(\varphi, S) \cup \{(\mathsf{inside}\,(S) \wedge \neg\mathsf{terminate}\,(S)\,, \textbf{emit}\ c)\}$

Hence, the control flows of **exclusive** ζ **in** S **end** and S are the same, and the data flow differs only in that **exclusive** ζ **in** S **end** additionally emits the variable ζ whenever the control flow moves inside S. Using this direct defi nition via the semantics given in [112] instead of the above macro expansion, we circumvent the use of the additional local variable t and the additional **pause** statement.

The main idea is to replace all **pause** statements in program locations which are not included in **exclusive** statements, by **await** $\neg\zeta$. This will suspend all threads which are running in parallel to blocks marked by **exclusive**. Outside **exclusive**-blocks, the control flow returns to its normal status, since ζ is not emitted.

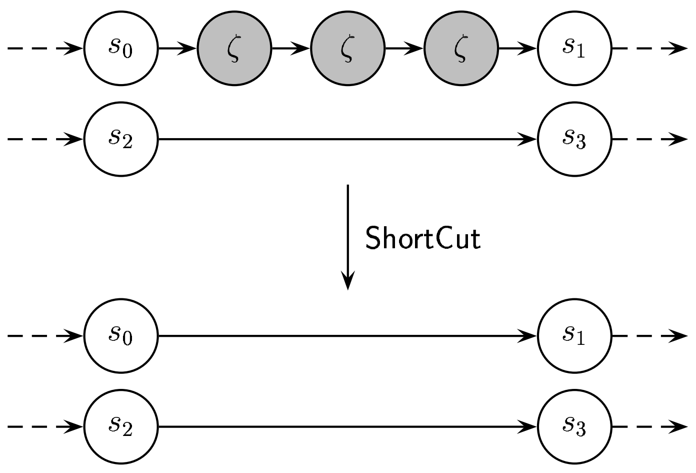


Figure 6.8: Generating Kripke Structures Preserving Native CPU-Instructions

The technique is shown in Figure 6.8. In the fi rst place (upper part of Fig. 6.8), a UDS is generated, where all states contained in **exclusive** locations are marked by the variable ζ, while threads running in parallel are suspended until the end of ζ- emissions.

The approach allows a low level runtime analysis considering native processor instructions, which can only be stated as a sequence of macro steps in synchronous languages, like divisions. Consider a microprocessor which implements for example a division instruction $instr_{div}$ (as an atomic one) for a given data type d_{div}. Usually, such operations are translated into sequences of boolean calculations. In order to consider $instr_{div}$ as an atomic instruction we have to merge these sequences of boolean calculations into single transitions and obtain the UDS shown in the lower part of Fig. 6.8.

This means that the state of the machine and the state of the formal model will be identical before and after the execution of an arithmetic instruction. Hence, we have removed the parallelism which can't exist on a target microprocessor due to the fact that arithmetic operations are executed as non interruptible atomic actions on a target machine.

To enable code generation with support for native CPU instructions, we need also information about the kind of the native instructions and the point of time of their execution. Applying the **exclusive** statement leads to a formal model, where $\mathcal{S}$ is reduced to the states terminate (S). These states correspond to the termination of the native instruction on the microprocessor. To give also the information about the kind of the native instructions, we simply use a labeling function $N(\mathcal{S}) \rightarrow \mathcal{A}$, where $\mathcal{S}$ is the set of states and $\mathcal{A}$ is the set of arithmetic instructions.

Finally, the resulting transition relation is constructed by the algorithm ShortCut shown in Figure 6.9, which transforms the selected paths into atomic transitions. ShortCut works according to a similar principle as the Chronos algorithm of Figure 6.2, explained in section 6.1.2.

```
function ShortCut($\mathcal{U}, \mathcal{S}_\varphi$)
  $\mathcal{U}_{short} := \{(s,s') \in \mathcal{U} \mid \{s,s'\} \cap \mathcal{S}_\varphi = \{\}\}$;
  $\mathcal{U}_{out} := \{(s,s') \in \mathcal{U} \mid s \in \mathcal{S}_\varphi \wedge s' \notin \mathcal{S}_\varphi\}$;
  repeat
    $\mathcal{U}_0 := \{(s,s') \mid \exists s_1.(s,s_1) \in \mathcal{U} \wedge (s_1,s') \in \mathcal{U}_{out}\}$;
    $\mathcal{U}_{short} := \mathcal{U}_{short} \cup \{(s,s') \mid (s,s') \in \mathcal{U}_0 \wedge s \notin \mathcal{S}_\varphi\}$;
    $\mathcal{U}_{out} := \{(s,s') \in \mathcal{U}_0 \mid s \in \mathcal{S}_\varphi\}$;
  until $\mathcal{U}_{out} = \{\}$;
  return $\mathcal{U}_{short}$;
end function
```

Figure 6.9: ShortCut Algorithm

Recall, that we may assume that all computation paths starting at the beginning of an arithmetic instruction will finally reach the end of the operation when ShortCut is called. We first compute the transitions between all states s and s' that are not marked by the variable ζ. To compute the further transitions, we have to compute first the transitions $\mathcal{U}_{out}$ that exit from a marked sequence. In the following loop, the algorithm performs a short cut of all predecessor transitions of $\mathcal{U}_{out}$.

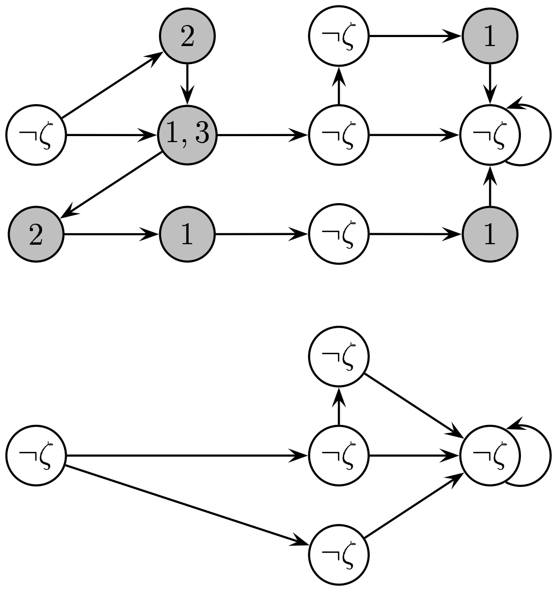


Figure 6.10: Function of the ShortCut algorithm

Figure 6.10 shows an example for the function of the ShortCut algorithm. The algorithm computes first the transitions $\mathcal{U}_{out}$ that exit sequences of states marked with ζ (shadowed). In the upper part of Fig. 6.10 these exiting states are marked with $''1''$. In the following loop, the algorithm traverses backwards, replacing current ζ-states by edges and computing simultaneously

their predecessors. These predecessors are marked in the upper part of Fig. 6.10 with "2" (to be eliminated in the second iteration) and "3" (to be eliminated in the third iteration). The algorithm terminates when all ζ-states are eliminated, i.e. when no such states are contained in the computed predecessors, as shown in the lower part of Fig. 6.10.
It is easily seen that the loop in ShortCut will be repeated at most d_{len} times, where d_{len} is the maximal length of a finite sequence of states in a computation path representing an arithmetic operation.

6.2.3 Exact Low-Level Runtime Analysis: Generating the Low-Level Model

We assume that we already have a function QuartzCompileUDS for the code generation, that computes an equivalent unit delay structure (UDS) $\mathcal{K}_{\mathcal{U}}$ of a given Quartz program $\mathcal{P}$. The states of this structure correspond with the states of the program $\mathcal{P}$ and are labeled with Boolean variables. Such a function is essentially implemented by any compiler, like the ones described in [111, 112]. To finally obtain an equivalent TKS $\mathcal{K}_{\mathcal{R}}$, it is therefore sufficient to be able to compute a corresponding TKS from a given UDS where the transitions between the states are endowed by notions of physical time. These labels are obtained by measuring the runtime for generated platform-specific code for the micro steps that are related to the transitions.

We first use a function QuartzCompileC which translates the obtained UDS $\mathcal{K}_{\mathcal{U}}$ into C-Code according to the method described above. The TKS is constructed by the algorithms given in Figure 6.12. To explain these algorithms, we first want to emphasize, that our goal is to develop symbolic techniques that allow us to consider *sets of states together with their transitions in a single iteration*, instead of processing all transitions of the UDS one after the other. This makes it possible to perform the analysis in less than $|\mathcal{S}|^2$ transitions.

An important observation for this analysis is to consider the state transition equations that are generated by the translation of the Quartz program into an equation system. Our method takes advantage from the fact that some operations consume identical amount of time regardless of the values of their operands, while the time consumption of other operations depends on their operands, e.g. the multiplication and division instructions on a Motorola 68000. The key idea is to use efficient techniques in order to merge sets of transitions, so that we are able to consider the same execution time for all merged transitions instead of calculating the execution times for all possible transitions of the UDS. This enables the use of symbolic techniques for the construction of the TKS as follows:

The first step is to distinguish between variables that exclusively occur in statements, which always consume identical amount of time ($\mathcal{V}_{nonia}$), and other variables occurring in statements which consume variable amounts of time, according to their operands ($\mathcal{V}_{ia}$). Hence, the set of variables is partitioned as

$$\mathcal{V} := \mathcal{V}_{nonia} \cup \mathcal{V}_{ia}$$

The property of $\mathcal{V}_{ia}$ is dominant over the property of $\mathcal{V}_{nonia}$, so $\mathcal{V}_{nonia} \cap \mathcal{V}_{ia} = \{\}$. We also distinguish between these sets in the labeling function of the TKS, i.e., we have $\mathcal{L}_{nonia}$ and $\mathcal{L}_{ia}$ that correspond to $\mathcal{V}_{nonia}$ and $\mathcal{V}_{ia}$, respectively, so that

$$\mathcal{L}(s) = \mathcal{L}_{ia}(s) \wedge \mathcal{L}_{nonia}(s), \forall s \in \mathcal{S}.$$

Hence, if $\mathcal{V}_{ia} = \{\}$ then it follows that $\mathcal{L}_{ia}(s) = true, \forall s \in \mathcal{S}$ and also if $\mathcal{V}_{nonia} = \{\}$ then it follows that $\mathcal{L}_{nonia}(s) = true, \forall s \in \mathcal{S}$.

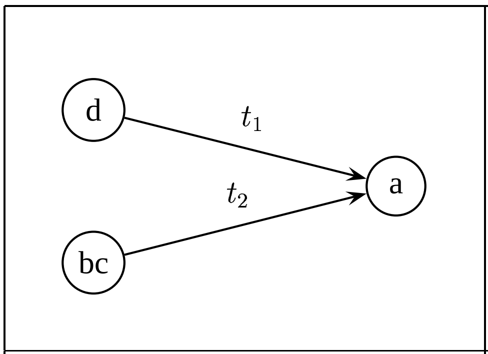
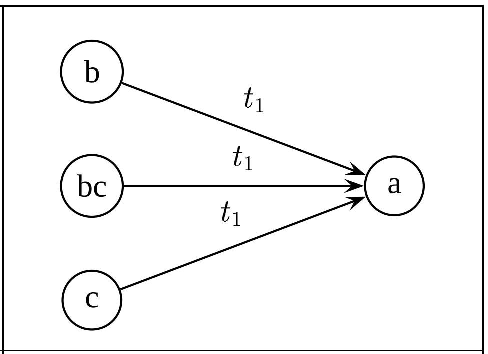
	
$next(a) = b?c : d$	$next(a) = b \vee c$

Figure 6.11: Examples transition equations

An example is shown in Fig. 6.11. Boolean operations consume identical amount of time on an ALU, regardless of the values of their operands. This is not the case for If-Then-Else-statements, which are often handled by microprocessors by means of jump-instructions. Their runtimes therefore depend on the confi guration of their arguments. Hence, the two transitions referring to the statement $next(a) = b?c : d$ consume different times t_1 and t_2. On the other hand the statement $next(a) = b \vee c$ of Fig. 6.11 consumes identical time t_1 for all three transitions since it is a boolean combination of three variables without any If-Then-Else-statement.

The second step is performed by the main algorithm QuartzCompileLTKS given in Fig. 6.12. QuartzCompileLTKS must consider all different variable confi gurations of $\alpha \in \mathcal{V}_{ia}$, but only one single confi guration $\beta \in \mathcal{V}_{nonia}$ and this only under the condition that there exist a reachable state meeting such a confi guration, i.e. $\exists s \in \mathcal{S}_{reach}.\mathcal{L}(s) = \alpha \wedge \beta$, where $\mathcal{S}_{reach} \subseteq \mathcal{S}$ is the set of reachable states (cf. Fig. 5.1). If there is no such β, then the represented state is not reachable and must hence not be considered in a runtime analysis.

After having determined a runtime t for a given confi guration of $\mathcal{V}_{ia} \wedge \mathcal{V}_{nonia}$, all transitions (the set $\mathcal{U}_{time}$) that are represented by all possible variable confi gurations of $\mathcal{V}_{nonia}$ are labeled by the physical time t. The function RuntimeC determines the runtime of the generated code for a given set of transitions. RuntimeC performs dynamic runtime measurements for given confi gurations of $\mathcal{V}_{ia} \wedge \mathcal{V}_{nonia}$ by means of so-called *profiling tools* like gprof [59] or VTune [72].

```
**function** QuartzCompileLTKS($\mathcal{P}$)
  $(\mathcal{I}, \mathcal{S}, \mathcal{U}) :=$ QuartzCompileUDS($\mathcal{P}$);
  $\mathcal{C} :=$ QuartzCompileC($\mathcal{U}$);
  $\mathcal{R} := \{\}$;
  **while** $\mathcal{U} \neq \{\}$ **do**
    $\mathcal{S}_{nonia} := \{s \in \mathcal{S} \mid \exists s' \in \mathcal{S}.(s, s') \in \mathcal{U} \wedge \mathcal{L}_{nonia}(s) \neq false\}$;
    $s_{time}$ := **choose any of** $\mathcal{S}_{nonia}$;
    $time :=$ RuntimeC($s_{time}, \mathcal{C}$);
    $\mathcal{U}_{time} := \{(s, s') \in \mathcal{U} \mid \exists s, s' \in \mathcal{S}.\mathcal{L}_{nonia}(s) = \mathcal{L}_{nonia}(s_{time})\}$;
    $\mathcal{U} := \mathcal{U} \setminus \mathcal{U}_{time}$;
    $\mathcal{R} := \mathcal{R} \cup \{\mathcal{U}_{time} \times \{time\}\}$;
  **end**;
  **return** $\mathcal{R}$;
**end function**

**function** RuntimeC($s, \mathcal{C}$)
  $time$ := **execution time**($\mathcal{C}(s)$);
  **return** $time$;
**end function**
```

Figure 6.12: Algorithms for Generation of Low-Level TKSs

The set $\mathcal{U}_{time}$ is removed from the UDS $\mathcal{U}$ since it is of no interest for further runtime analysis. This guaranties also that the time labeling of the edges is unique, i.e. for each transition $(s, s') \in \mathcal{U}$ exists exactly one $t \in \mathbb{N}$ such that $(s, t, s') \in \mathcal{R}$.

The algorithm terminates when all states of $\mathcal{U}$ were considered in the runtime analysis, i.e. $\mathcal{U} = \{\}$. Note that if $\mathcal{V}_{ia} = \{\}$, i.e., if there are no arithmetic instructions included in the code, $\mathcal{L}_{ia}$ returns $true$ for all possible inputs. In this case the generated code for the transitions contains only boolean operations which consume the same amount of time for all possible variable confi gurations, and hence the entire TKS $\mathcal{R}$ will labeled with the same time t.

As an example, Fig. 6.14 shows a TKS, obtained from the Quartz program shown in Fig. 6.13.

```
module QuartzExample :

input req;
output out;

          a1 : await (req);
          emit out;
          a2 : halt
end
```

Figure 6.13: An Example for a Quartz Program

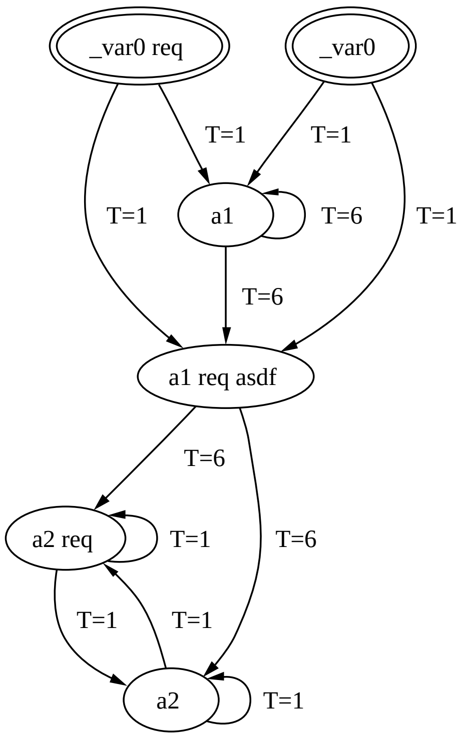


Figure 6.14: A TKS for the Quartz Program of Figure 6.13

Theorem 5 (QuartzCompileLTKS Iterations) *The algorithm* QuartzCompileLTKS *terminates in maximum* 2^n *Iterations, where* $n = |\mathcal{V}_{if}|$.

Proof: The theorem can be easily proved by induction on n:

$n = 0$:
if $\mathcal{V}_{ia} = \{\}$ and $\mathcal{V}_{nonia} \neq \{\}$, then according to the definition of $\mathcal{L}_{ia}$ we have $\mathcal{L}_{ia}(s) = true, \forall s \in \mathcal{S}$. But then $\mathcal{U}_{time} = \mathcal{U}$ holds trivially and hence also $\mathcal{U} \setminus \mathcal{U}_{time} = \{\}$, which terminates the algorithm after one iteration, i.e. $1 = 2^{|\mathcal{V}_{ia}|}$.

If we assume that for $n = |\mathcal{V}_{ia}|$ the algorithm QuartzCompileLTKS will terminate after maximum 2^n steps, then for $n + 1$ we have:

If $|\mathcal{V}_{ia}| = n + 1$, then for $x \in \mathcal{V}_{ia}$ we separate $\mathcal{U}$ as follows:

$$\begin{gathered}
\mathcal{U} := \mathcal{U}_x \cup \mathcal{U}_{\neg x}, \text{ where} \\
\mathcal{U}_x := \{(s, s') \in \mathcal{U}.x \in \mathcal{L}_{ia}(s)\}\ , \\
\mathcal{U}_{\neg x} := \{(s, s') \in \mathcal{U}.x \notin \mathcal{L}_{ia}(s)\} \\
\text{and } \mathcal{U}_x \cap \mathcal{U}_{\neg x} = \{\}.
\end{gathered}$$

If we apply QuartzCompileLTKS to $\mathcal{U}_x$ and $\mathcal{U}_{\neg x}$ separately, then we have

for $\mathcal{U}_x$: $x \in \mathcal{L}_{ia}(s), \forall s \in \mathcal{S}_{ia}$ and

for $\mathcal{U}_{\neg x}$: $x \notin \mathcal{L}_{ia}(s), \forall s \in \mathcal{S}_{ia}$.

In other words, x has fixed values in $\mathcal{U}_x$ and $\mathcal{U}_{\neg x}$ and hence, it must not be considered for determining the possible configurations of $\mathcal{V}_{ia}$-variables. The variable set of $\mathcal{V}_{ia}$ can then be expressed as $\mathcal{V}_{\neg x} = \mathcal{V}_{ia} \setminus \{x\}$, where $|\mathcal{V}_{\neg x}| = n$.

According to the induction's assumption, $\mathcal{U}_x$ and $\mathcal{U}_{\neg x}$ can then be computed in maximum $2^n = 2^{|\mathcal{V}_{\neg x}|}$ iterations and hence, for the entire problem we have a maximum of $2 \cdot 2^n = 2^{n+1} = 2^{|\mathcal{V}_{ia}|}$ iterations. ■

Chapter 7

Exact WCET and BCET Analysis

'Εξελέγχων μόνος ἀλάθειαν ἐτήτυμον χρόνος. [1]
ΠΙΝΔΑΡΟΣ (518 - 440 π.Χ.)
(Ολυμπ. Χ, 65)

As a direct result of the previous sections, we present in this section a new and completely automatic approach to perform *exact worst– and best case execution time (WCET and BCET)* analysis. Our approach overcomes the problems described in section 1.1.3. For a given program, we compute the exact best and worst runtimes in terms of macro steps *for all possible inputs at once*. We are even able to compute the input sequences that require these bounds. For this purpose, we have to assume that all data types were finite, so that the overall problem becomes decidable. For embedded systems, this restriction is not a severe one. Moreover, modelling integers with a finite, constant bitwidth is even more accurate and allows one to detect problems with overflows and underflows.

Nevertheless, checking all input sequences is still a highly complex task. A key idea of this approach is therefore to use symbolic state space exploration techniques [11, 22], developed for the verification of temporal properties of reactive systems. Beneath the symbolic state space traversal, the other essential key ingredient to our solution is the use of synchronous programming languages to achieve descriptions of the system.

Having constructed real-time models considering logical as well as physical time (cf. chapter 6), allows us to compute exact results, i.e., to overcome the known problem of computing only highly pessimistic results due to simply adding maximal bounds (cf. section 1.1.3). To this end, the WCET analysis problem is transfered into a symbolic state space exploration problem, which is performed on a given TKS. We emphasize, that our technique is *not dependent on the type (logical or physical) of the TKS's time notion*. Hence, it can directly be applied to both, low-level– and high-level WCET analysis. For this purpose, *we reduce both problems to the*

[1] In fact, time by itself reveals the truth.
PINDAROS (518 - 440 BC)

same UDS–symbolic state space exploration problem, as follows:

By the semantics of synchronous languages, there will be only finitely many micro steps in a macro step. If logical time is considered, then a given TKS was obtained by abstraction techniques like the ones introduced in section 6.1.2. This means that the notions of logical time correspond to macro steps that were declared to be irrelevant for verification purposes. Nevertheless, these abstracted macro steps must be considered for WCET and BCET analysis, since they consume time. Using the algorithm of Figure 4.3 given in section 4.1, we can obtain a UDS out of the given TKS. This is allowable due to the fact, that we do not consider the evaluation of any JCTL formulae, but compute only longest and shortest paths of the transition system. On the other hand, applying the algorithm of Figure 4.3 when physical time is considered, is valid due to the fact that physical time is running on a TKS equally fast on all transitions. In other words, the obtained UDS contain only transitions that are synchronized and consume one unit of physical time. Hence, they can be considered as synchronized macro steps.

Our procedure works as follows: We start with a synchronous program and translate this program as described in chapter 6 into a TKS. Clearly, if only high-level analysis is to be considered, one can avoid all TKS-generation procedures described in chapter 6 and directly use the formal model obtained from the compiler. The algorithm presented in Figure 7.1 of section 7.1.1 is then used to compute the minimal and maximal numbers of macro steps necessary to reach a set of control flow states $\mathcal{S}_\gamma$ from another set of control flow states $\mathcal{S}_\alpha$. Additionally, we can count the number of visits of a third set of states $\mathcal{S}_\beta$ while the control flow moves from $\mathcal{S}_\alpha$ to $\mathcal{S}_\gamma$. It is also possible to compute the input sequences that lead to these number of iterations. To specify sets $\mathcal{S}_\alpha$, $\mathcal{S}_\beta$, and $\mathcal{S}_\gamma$, it is convenient to make use of the control flow predicates given in section 2.2. Hence, we are able to compute the exact minimal and maximal reaction times in terms of macro steps. In particular, we can compute path information like loop bounds, the minimal/maximal number of macro steps required to reach a certain program location from another one, as well as infeasible paths.

There is some related work like [28] where similar algorithms for runtime analysis on transition systems have been considered. However, in contrast to our approach, [28] is not integrated in a design flow, i.e. it has no relationship to the program source, and hence, is not able to compute program related properties like the minimal/maximal numbers of loop iterations. The transition systems we analyze are obtained from synchronous programs that are used for automatic hardware and software generation. Hence, our approach is not restricted to logical steps, but enables also low-level WCET and BCET analysis.

7.1 WCET Analysis of Synchronous Programs

The presentation of the semantics in the form indicated in Figure 6.6 is the basis of our execution time analysis. However, the key problem in execution time analysis, namely to determine how many transitions can be taken from a state set $\mathcal{S}_\alpha$ to another state set $\mathcal{S}_\gamma$, is still an undecidable problem. Nevertheless, if we assume that all data types used in the program are finite, then we can compile the program to a classical finite state machine (fsm). Clearly, the obtained fsm will normally suffer from the enormous state explosion. For this reason, we use a symbolic representation in the sense of symbolic state space exploration [11, 22]. The key idea is thereby that sets are not explicitly stored; instead the characteristic function is represented as a Boolean formula that is itself stored in a canonical normal form (BDDs [17]).

7.1.1 Determining Path Information

In this section, we present techniques for WCET and BCET analysis of a given UDS that corresponds to a synchronous program. In particular, we explain how the lower and upper bounds of loop iterations can be efficiently calculated.

The essential task of high-level WCET and BCET analysis is then that for given sets of states $\mathcal{S}_\alpha$ and $\mathcal{S}_\gamma$, we have to compute the minimal and maximal numbers of transitions necessary to reach $\mathcal{S}_\gamma$ from $\mathcal{S}_\alpha$. $\mathcal{S}_\alpha$ and $\mathcal{S}_\gamma$ are thereby represented as formulae α and γ.

We can furthermore determine the minimal and maximal number of loop iterations in that we count the number of visits in a further set of states $\mathcal{S}_\beta$, while traversing from $\mathcal{S}_\alpha$ to $\mathcal{S}_\gamma$. For example, for a loop **do** S **while** σ, we can use the following formulae to compute the number of loop iterations:

- $\alpha :\equiv \neg\mathsf{inside}\,(S) \wedge \mathsf{enter}\,(S)$ describes all situations where the control flow is not yet inside the loop body S ($\neg\mathsf{inside}\,(S)$), but will enter the loop body right now ($\mathsf{enter}\,(S)$).
- $\beta :\equiv \mathsf{terminate}\,(S) \wedge \sigma$ describes all situations where the execution of the loop body S currently terminates ($\mathsf{terminate}\,(S)$) and the loop condition σ holds. Hence, the loop body is once more executed.
- $\gamma :\equiv \mathsf{terminate}\,(S) \wedge \neg\sigma$ describes all situations where the execution of the loop body S currently terminates ($\mathsf{terminate}\,(S)$) and the loop condition σ does not hold. Hence, the loop terminates.

Using symbolic representations of the properties α, β, and γ, it is straightforward to compute the corresponding sets of states $\mathcal{S}_\alpha$, $\mathcal{S}_\beta$, and $\mathcal{S}_\gamma$ of the Kripke structure, where α, β, and γ, respectively, holds.

In the first place, we must ensure that the given program is correctly implemented, i.e. that all computation paths starting at $\mathcal{S}_\alpha$ will finally reach $\mathcal{S}_\gamma$. This correctness property can be easily verified by checking the following CTL property, that states that all computation paths that start in $\mathcal{S}_\alpha$ must finally reach $\mathcal{S}_\gamma$:

$$\mathsf{AG}(\alpha \rightarrow \mathsf{AF}\gamma) \qquad (2)$$

For the above properties α, β, and γ this means that the loop will terminate for all inputs. The final WCET / BCET analysis together with the calculation of bounds for loop iterations is then

```
function EHLA($\mathcal{U}, \mathcal{S}_\alpha, \mathcal{S}_\beta, \mathcal{S}_\gamma$)
  $i = bound := 0$;
  $\mathcal{T} = \mathcal{S}_{\alpha\gamma} := \{\}$;
  repeat
    if $\mathcal{S}_\alpha \cap \mathcal{S}_\gamma \neq \{\}$ then
      $\mathcal{T} := \mathcal{T} \cup \{i\}$;
      $\mathcal{S}_\alpha := \mathcal{S}_\alpha \setminus \mathcal{S}_\gamma$;
    endif ;
    $\mathcal{S}_\alpha := \{s' \in \mathcal{S} \mid (s, s') \in \mathcal{U} \wedge s \in \mathcal{S}_\alpha\}$;
    $\mathcal{S}_{\alpha\gamma} := \mathcal{S}_{\alpha\gamma} \cup \mathcal{S}_\alpha$;
    $i := i + 1$;
  until $\mathcal{S}_\alpha := \{\}$;
  $\mathcal{S}_{\overline{\alpha\beta\gamma}} := \{s \in \mathcal{S} \mid s \notin \mathcal{S}_\alpha \cup \mathcal{S}_\beta \cup \mathcal{S}_\gamma\}$;
  $\mathcal{U}_{\alpha\gamma} := \{(s, s') \mid s \in \mathcal{S}_{\alpha\gamma} \wedge s' \in \mathcal{S}\}$;
  $\mathcal{U}_{short} :=$ ShortCut($\mathcal{U}_{\alpha\gamma}, \mathcal{S}_{\overline{\alpha\beta\gamma}}$);
  $\mathcal{U}_{\alpha\beta} := \{(s, s') \in \mathcal{U}_{short} \mid s \in \mathcal{S}_\alpha \wedge s' \in \mathcal{S}_\beta\}$;
  $\mathcal{S}_{next} := \{s \in \mathcal{S}_\beta \mid \exists s \in \mathcal{S}.(s, s') \in \mathcal{U}_{\alpha\beta}\}$;
  repeat
    $\mathcal{S}_{next} := \{s' \in \mathcal{S}_\beta \mid \exists s \in \mathcal{S}.(s, s') \in \mathcal{U}_{short} \cap (\mathcal{S}_{next} \times \mathcal{S})\}$;
    $bound := bound + 1$;
  until $\mathcal{S}_{next} := \{\}$;
  return $(\mathcal{T}, bound - 1)$;
end function
```

Figure 7.1: Determining Minimal and Maximal Computation Paths and Loop-Iterations

performed by the function EHLA *(*E*xact* H*igh– and* L*ow Level* A*nalysis)*, shown in Figure 7.1. Arguments of the algorithm are the transition relation $\mathcal{U}$ of the UDS, the source set of states $\mathcal{S}_\alpha$, the set of target states $\mathcal{S}_\gamma$, and the set of states $\mathcal{S}_\beta$. As an invariant, the algorithm stores in $\mathcal{S}_\alpha$ the set of states that can be reached within i steps from the originally given set $\mathcal{S}_\alpha$. Using a breadth fi rst search, the algorithm successively computes all successor states of the current set $\mathcal{S}_\alpha$. A variable i is used to count the number of macro steps taken so far, and a set $\mathcal{T} \subseteq \mathbb{N}$ stores the lengths of paths from $\mathcal{S}_\alpha$ to $\mathcal{S}_\gamma$.

For the calculation of $\mathcal{S}_\beta$-visits, we must consider the fact that paths of different length will reach $\mathcal{S}_\beta$, at different points of time. To avoid multiple counting of $\mathcal{S}_\beta$-visits occurring due to such time differences, we use the algorithm ShortCut introduced in section 6.2.2 (cf. Figure 6.9), in order to equalize fi rst the time needed to reach $\mathcal{S}_\beta$ from $\mathcal{S}_\alpha$. Using ShortCut, the algorithm EHLA simply eliminates all non-α, non-β and non-γ states between $\mathcal{S}_\alpha$, $\mathcal{S}_\beta$ and $\mathcal{S}_\gamma$, so that $\mathcal{S}_\beta$ can be reached equally fast from $\mathcal{S}_\alpha$, regardless of the followed path.

ShortCut works according to a similar principle as the Chronos algorithm of Figure 6.2, explained in section 6.1.2. To guarantee a cycle-free input needed for these kind of algorithms, the fi rst loop in EHLA collects in the set $\mathcal{S}_{\alpha\gamma}$ the states between $\mathcal{S}_\alpha$ and $\mathcal{S}_\gamma$. Only these states are then considered by ShortCut, which are guaranteed cycle-free, due to (2).

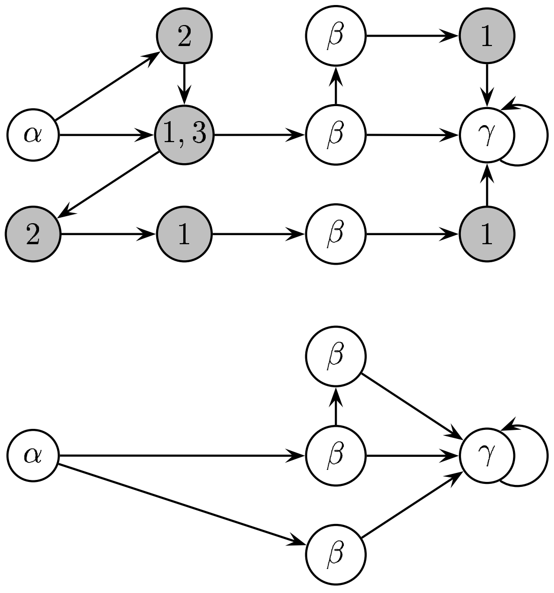


Figure 7.2: Function of the ShortCut algorithm in WCET Analysis

Fig. 7.2 shows an example for the function of the ShortCut algorithm for the WCET analysis approach. The algorithm computes fi rst the transitions $\mathcal{U}_{out}$ that exit sequences of non-α, non-β and non-γ states. In the upper part of Fig. 7.2 these states are marked with ″1″. In the following loop, the algorithm traverses backwards, replacing current non-α, non-β or non-γ states by edges and computing simultaneously their predecessors. These predecessors are marked in the upper part of Fig. 7.2 with ″2″ (to be eliminated in the second iteration) and ″3″ (to be eliminated in the third iteration). The algorithm terminates when all non-α, non-β and non-γ states are eliminated, i.e. when no such states are contained in the computed predecessors, as shown in the lower part of Fig. 7.2.

It is easily seen that the loop in ShortCut will be repeated at most d_{len} times, where d_{len} is the maximal length of a fi nite sequence of non-α, non-β and non-γ states.

Two main things must be checked within the EHLA-loops:

- If a path reaches the target set of states $\mathcal{S}_\gamma$ (checked by $\mathcal{S}_\alpha \cap \mathcal{S}_\gamma \neq \{\}$), then the current number of macro steps i is added to the set $\mathcal{T}$, and the path is removed from the current set $\mathcal{S}_\alpha$ (so it will not be considered further). The algorithm terminates when the set $\mathcal{S}_\alpha$ is empty, i.e. if no more paths are left. Then, $\mathcal{T}$ will contain lengths of all paths from $\mathcal{S}_\alpha$ to

$\mathcal{S}_\gamma$. The minimum and maximum of $\mathcal{T}$ are then the BCET and WCET, respectively. For compact storage, one can also easily represent $\mathcal{T}$ by means of a BDD.

- If some path reaches the set of states $\mathcal{S}_\beta$, then the loop bound variable $bound$ is incremented. At the end of the calculation, $bound - 1$ will deliver exactly how many times $\mathcal{S}_\beta$ can be visited by computations from $\mathcal{S}_\alpha$ to $\mathcal{S}_\gamma$.

The EHLA algorithm works on finite-state transition systems, but gathers runtime information about the program locations described by α, β, and γ. Note that the compiler used to generate executable C-code or hardware circuits is the same that is used to compute our finite state machines. Hence, the overall design work flow assures that our runtime analysis determines the correct information.

Chapter 8

Experimental Results and Discussion

Πρός γάρ τό τελευταῖον ἐκβάν ἕκαστον τῶν πρίν ὑπαρξάντων κρίνεται. [1]
ΔΗΜΟΣΘΕΝΗΣ (383 - 322 π.Χ.)
(Ολυνθιακός, Γ, 32)

In the previous sections we have presented techniques for the specification, modelling, verification and runtime analysis of real-time systems.

We have implemented these techniques in our tool framework Equinox. Equinox consists of the BDD-based tool JERRY, which is used for model-checking and runtime analysis purposes and a compiler for the extended synchronous language Quartz. According to Fig. 1.2, Fig. 8.1 shows the flow of the techniques presented in this work. Having experimented with many BDD-tools available, we consider the CUDD-package [120] as a reliable BDD-package, offering a great number of useful features. Consequently all our tools use the CUDD-package for BDD-manipulation.

In this section, we present experimental results that we have obtained with Equinox, considering widely used benchmarks. Focusing on the different techniques presented in the previous sections, we consider the different stages of the Equinox framework as follows:

- *generation of high-level timed Kripke structures:* in order to directly obtain high-level real-time formal models out of industrially-used programming languages and simultaneously allow the use of abstractions, we have developed a real-time extension of the synchronous language Quartz together with its translation to timed Kripke structures. Our extension allows the programmer to declare program locations that are irrelevant for verification. The model is generated in two steps, which we consider separately:
  - first the program is translated into a unit-delay structure (UDS)
  - then the Chronos algorithm generates a TKS by ignoring the irrelevant states, while retaining the quantitative information.

[1] Based on the last event's result, all prior existing ones are judged.
DEMOSTHENES (383 - 322 BC)

These techniques are implemented in JERRY. They directly generate a single real-time transition system, thus overcoming the known problem of composing several real-time models.

- *generation of low-level timed Kripke structures by means of runtime analysis:* we have extended JERRY to an exact and detailed low-level runtime analysis in order to perform low-level real-time formal verifi cation. We presented a technique for analyzing the execution times of all single transitions of a synchronous program. This allows JERRY to generate low-level timed Kripke structures out of unit-delay structures: the transitions are labeled with the physical times required to execute the code on the transitions. The generated transition systems have timed transitions that correspond to *non-interruptible atomic actions* and can be verifi ed by means of the real-time temporal logic JCTL.
- *high-level and low-level formal verification:* we have shown that many existing approaches to real-time extensions of CTL are misleading. For this reason, we have developed a new real-time temporal logic JCTL which is directly defi ned on timed Kripke structures using interpretation I_J. We presented effi cient symbolic model checking algorithms for JCTL, so that we are able to benefi t from established improvements of symbolic state space traversal. These real-time model checking techniques are implemented in JERRY, together with standard qualitative-only CTL model checking algorithms.
- *WCET and BCET analysis:* we have presented a novel approach to analyze execution times of synchronous programs, which is able to compute for all input sequences the number of macro steps that are executed between given program locations α, β and γ. In particular, using control flow predicates, we can determine α, β and γ such that the algorithm can be used to compute exact bounds of loop iterations. The overall approach is implemented in JERRY.

Our results clearly show the advantages of Equinox′s and JERRY′s modular structure: having the possibility to store and reuse all models obtained in the intermediate steps of the analysis, allows us to individually start an analysis at each desired step, without having to calculate all previous steps from the beginning. This is very important for formal verifi cation purposes, since one of the main time– and memory-consuming factor is the construction of the fi rst unit-delay structure.

Note that the runtime and memory consumption is always dependent on the initial variable order. For the obtained results we have used the sifting variable reordering of the CUDD-package.

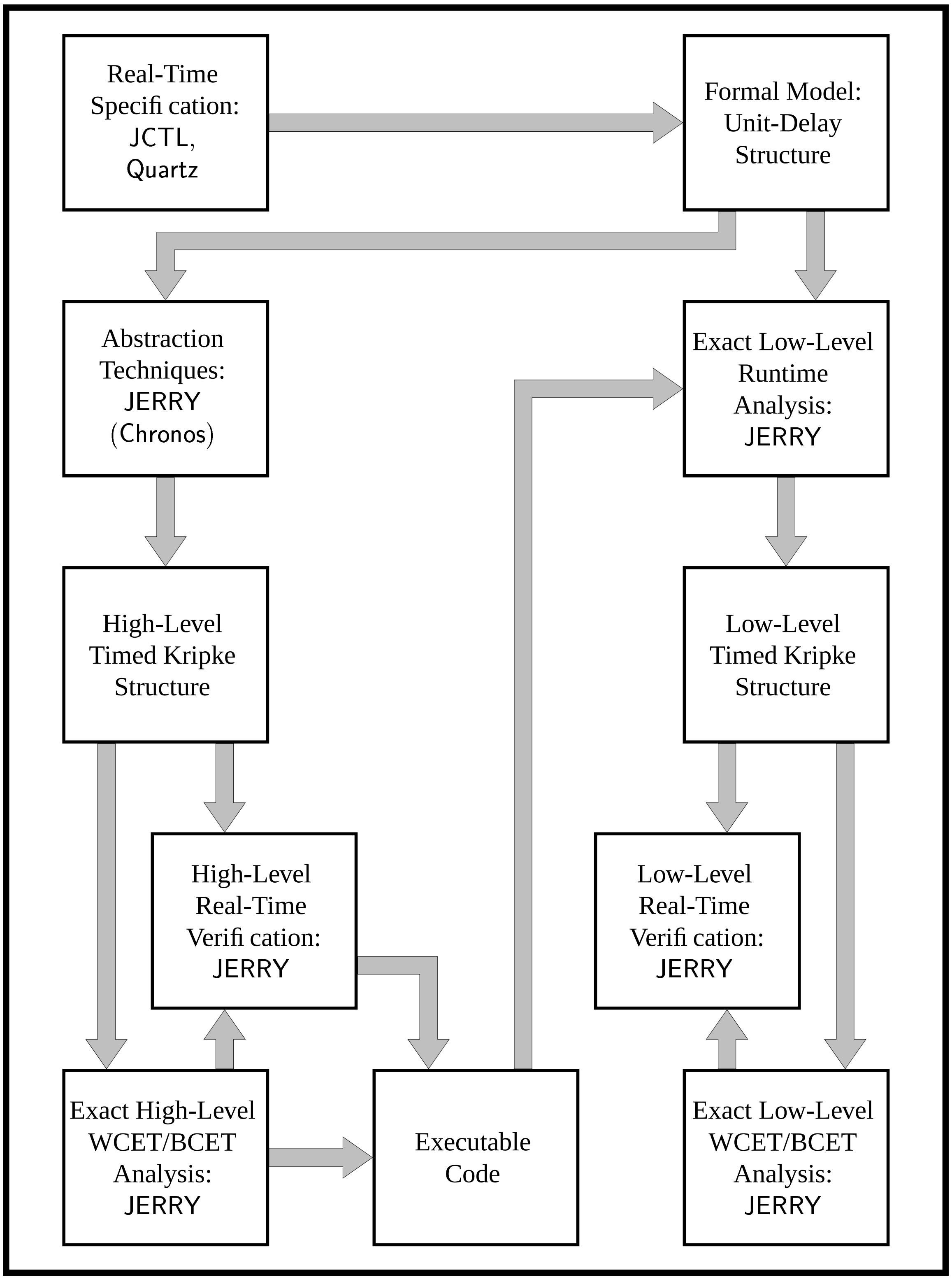

Real-Time Specifi cation: JCTL, Quartz
Formal Model: Unit-Delay Structure
Abstraction Techniques: JERRY (Chronos)
Exact Low-Level Runtime Analysis: JERRY
High-Level Timed Kripke Structure
Low-Level Timed Kripke Structure
High-Level Real-Time Verifi cation: JERRY
Low-Level Real-Time Verifi cation: JERRY
Exact High-Level WCET/BCET Analysis: JERRY
Executable Code
Exact Low-Level WCET/BCET Analysis: JERRY

Figure 8.1: Equinox: A Formal Framework for the Specifi cation, Modelling, Verifi cation and Runtime Analysis of Real-Time Systems

8.1 Fischer’s Mutual Exclusion Protocol

It is well-known that program sections of concurrent processes that modify some shared variables have to be protected to avoid inconsistencies of the data structures. The mutual exclusion in systems with concurrent processes is a mechanism, in order to allow different processes exclusive access on resources. Should any process be interrupted by another process during his access on this resources, loss of data or other unwanted effects/events could occur. The area, in which the process is not allowed to be interrupted, is called *critical section*.

To overcome this problem, several solutions have been suggested. Perhaps the simplest possible algorithm is the one suggested by Michael Fischer [77] and is known as *Fischer’s Mutual Exclusion Protocol*. Figure 8.2 gives some pseudo-code for the protocol: The protocol is used to protect critical sections for δ processes. For this purpose, a global lock variable λ of type $\{0, ..., \delta\}$ is used. The role of λ consists of holding the index of the process that is allowed to enter its critical section. The basic idea of the protocol is roughly as follows (cf. the program code in Figure 8.2):

If $\lambda = 0$ holds, the critical region is currently not owned by a process, so that a process that wants to enter the section can try to obtain access to the region. It therefore will then assign λ its own process id (line s_1). After this, the process will be inactivated for δ units of time so that the other δ processes have the chance to write their process ids to λ. If after that time, λ still contains the process id of the considered process, this process is allowed to enter the critical section and after that, it will release the section by resetting λ to zero.

```
s_init :  **repeat**
s_0 :         **await** $\lambda = 0$;
s_1 :         $\lambda := i$;
s_2 :         **sleep** $\delta$;
s_3 :     **until** $\lambda = i$;
s_4 :     //*critical section*
s_5 :     $\lambda := 0$;
```

Figure 8.2: Fischer’s Mutual Exclusion Protocol

The disadvantage of Fischer’s mutex protocol is that it requires for n processes in each process a delay time δ of order $O(n)$. In the meantime, other solutions have been presented that do not suffer from this disadvantage (cf. [77]). Nevertheless, Fischer’s mutex protocol is an excellent example which has been widely considered by many researchers as a benchmark.

The benchmark consist of n processes that execute the code given in Figure 8.3. There is a critical section between the locations *cs*$_1$ and *cs*$_2$, i.e. at most one of the processes is allowed to be in between these locations. The access to the critical region is controlled by a shared variable x: when $x = 0$ holds, the region is free, $x \in \{1, \ldots, n\}$ means that the process with the identifi er x is granted access to the region. A process tries to assign its process identifi er *pid* to a shared variable x. As the processes are executed in an interleaved manner, there will be no write conflict in doing so, and it may be the case that after having written x, it may no longer have the value *pid* when the process reaches the location *wait*$_2$.

```
module FischerProcess :
  input running, pid : integer;
  output x : integer;
  suspend
    do
      wait_1 : await (x = 0);
      next(x) := pid;
      wait_2 : pause
    while (x ≠ pid);
    cs_1 : pause;
    /* critical section */
    cs_2 : pause;
    x := 0;
  wait_3 : halt
  when running
end
```

Figure 8.3: A Process in Fischer's Protocol

We have used JERRY to model, analyze and verify Fischer's protocol up to 30 processes. The runtime and memory requirements obtained on an Intel Pentium 3 platform with 1 GHz and 1 GByte of main memory that runs under Linux are given in the next sections.

8.1.1 UDS Generation

The first part of the automatic formal analysis by means of Equinox is always the generation of the unit-delay structure. Table 8.1 shows the results for Fischer's UDS generation for n processes.

The columns of the table are as follows: The first column denotes the instantiation of the benchmark's parameter (number of processes). The second column shows how many Boolean variables were necessary to encode the state transition diagram, which means that the system has 2^{var} reachable states. Column three shows the determined runtimes for the UDS generation. Columns four and five show memory consumption, expressed in required BDD nodes and kBytes of memory respectively.

Table 8.1 shows that JERRY has the ability to handle very large systems. The generated model for 30 Fischer processes requires 186 boolean variables, which means that it includes 2^{186} reachable states. To be able to directly compare our results with other tools, we have generated all smaller models between 2 - 10 processes and proceeded in 5-steps for models above 10 processes, which could not be handled by some tools.

Note that the runtimes include also the time needed for exporting and storing the formal model and all other needed data on hard disk.

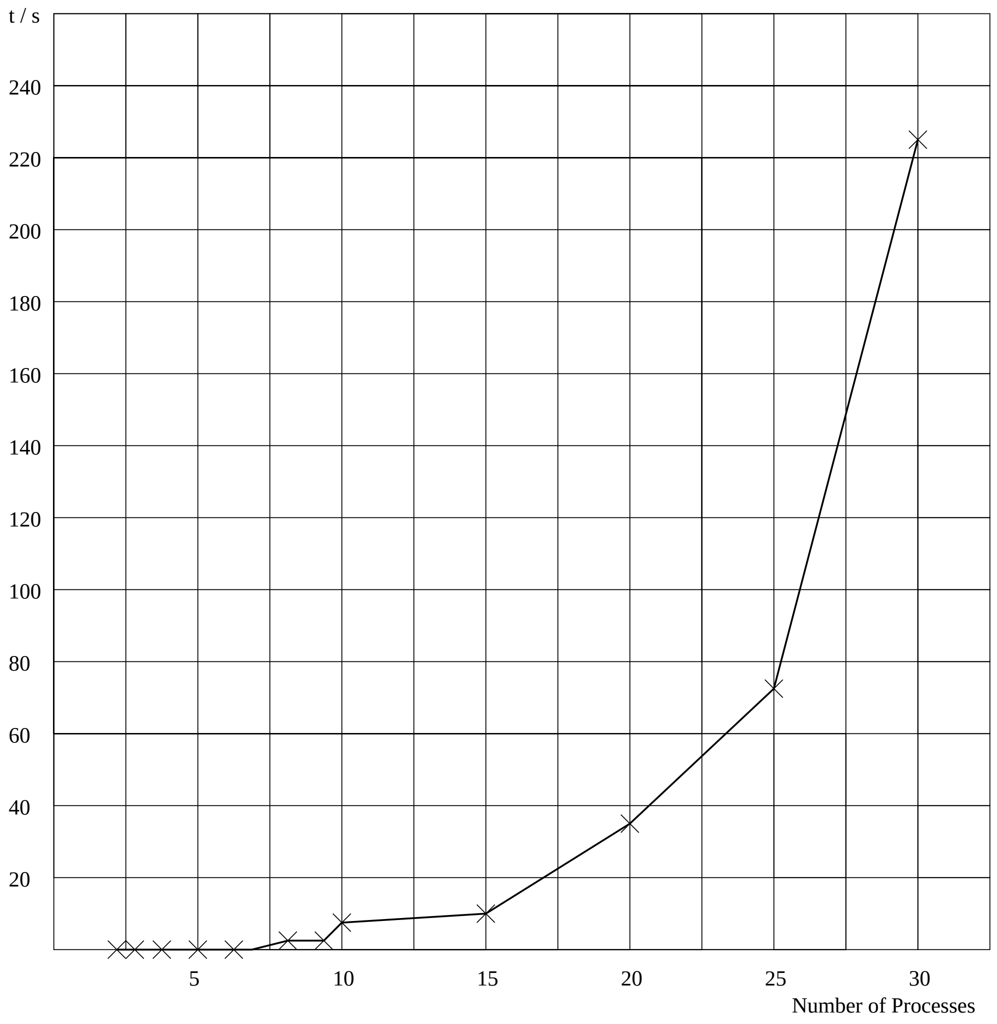


Figure 8.4: Fischer: UDS Generation Time

Fischer: UDS generation				
Number of Processes	Variables states	Time h:m:s	BDD nodes	Memory kB
2	15	0.15	230	4765
3	21	0.27	424	4770
4	28	0.40	711	4855
5	34	0.77	1156	5084
6	40	0.93	1523	5206
7	46	2.23	1979	5462
8	53	2.37	2712	5472
10	65	6.47	3346	5635
15	95	10.67	6501	6962
20	126	34.36	8714	14567
25	156	1:13.92	17527	16430
30	186	3:48.28	26531	27842
Pentium 3, 1GHz, 1GB				

Table 8.1: UDS Generation for Fischer's Mutex Protocol

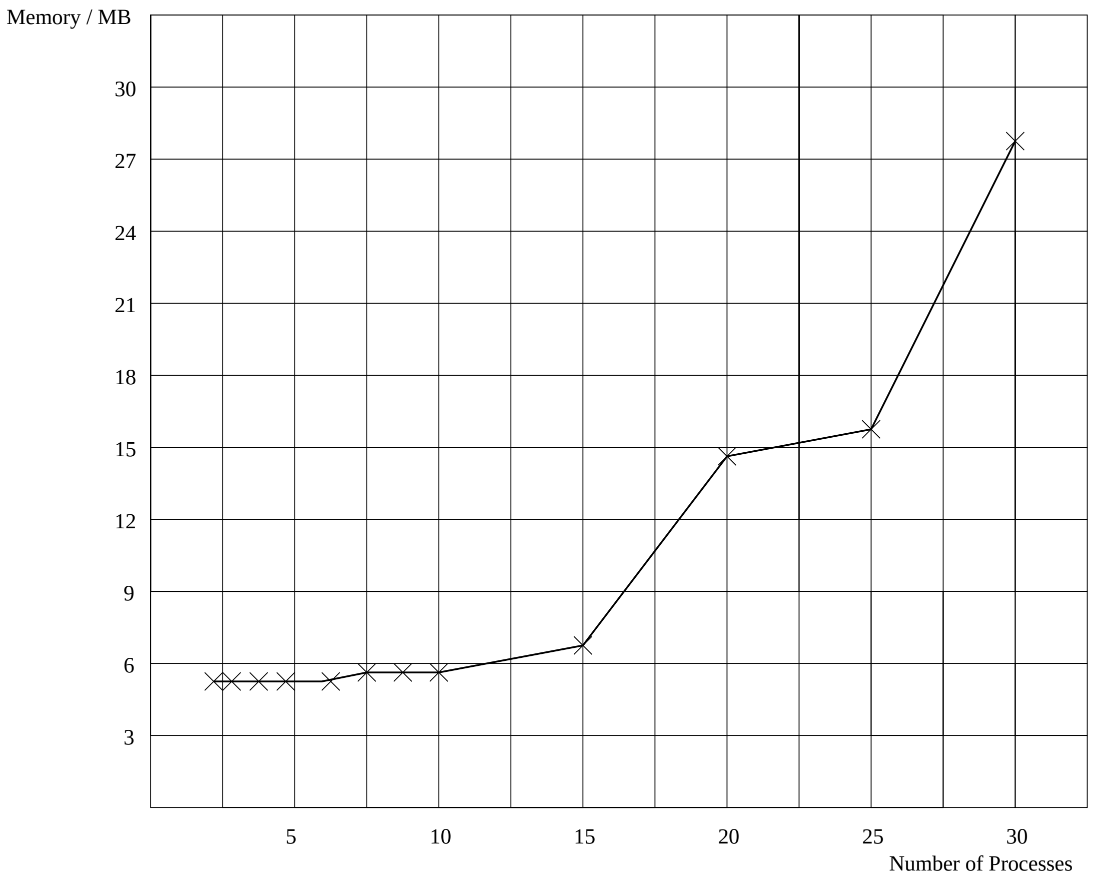


Figure 8.5: Fischer: UDS Generation Memory

8.1.2 High-Level TKS Generation - Chronos

The generation of the high-level TKS uses the algorithm Chronos, explained in section 6.1.2. Compared to other benchmarks discussed in the next sections, the chosen abstraction for the Fischer benchmark is small, since a coarser abstraction would probably ignore important details. Table 8.2 shows the obtained results. All experiments were run on a Pentium 3 with 1GHz and 1GByte of main memory.

The columns of the table are as follows: The first column denotes the instantiation of the benchmark's parameter (number of processes). The second column shows how many Boolean variables were necessary to encode the state transition diagram and the logical times on the transitions. This means that the system has 2^{s_var} reachable states and the longest transition of the system is not exceeding 2^{t_var} logical time units. Column three shows the determined runtimes for the high-level TKS generation. Columns four and five show memory consumption, expressed in required BDD nodes and kBytes of memory respectively. The required BDD nodes are given for both, the generated TKS and the obtained UDS (the TKS with untimed transitions), which is then used by the algorithms for all qualitative computations of the JCTL-operators. Column six finally, shows the minimum and maximum duration (in logical time units) of the transitions contained in the high-level TKS.

Fischer: High-Level TKS Generation - Chronos								
Number of	Variables		Time	BDD nodes		Memory	Transition	
Processes	states	time	h:m:s	UDS	TKS	kB	min	max
5	34	2	0.53	636	666	4876	1	2
10	65	2	4.42	2222	2299	6085	1	2
15	95	2	12.99	3856	4097	6988	1	2
20	126	2	36.01	8230	8636	9704	1	2
25	156	2	45.63	9708	9731	10847	1	2
30	186	2	1:21.30	16764	17061	14031	1	2
Pentium 3, 1GHz, 1GB								

Table 8.2: High-Level TKS Generation for Fischer's Mutex Protocol

The generation of the high-level model clearly shows how effective the use of abstractions can be in praxis. Even small abstractions, like the one used here, can lead to impressive results. Here, only 2 variables are required to represent time. Nevertheless, as can be seen, the size in BDD nodes of the generated TKS is significantly smaller (approx. 68% at average) than the size of the first generated UDS (cf. Table 8.1). Moreover, the use of time variables in order to keep the time-information stored, only adds a very small amount of BDD nodes (approx. 3.5% at average) to the abstract UDS. This is of essential importance for later verification purposes, where the runtimes and memory consumption will depend on the size of the formal model.

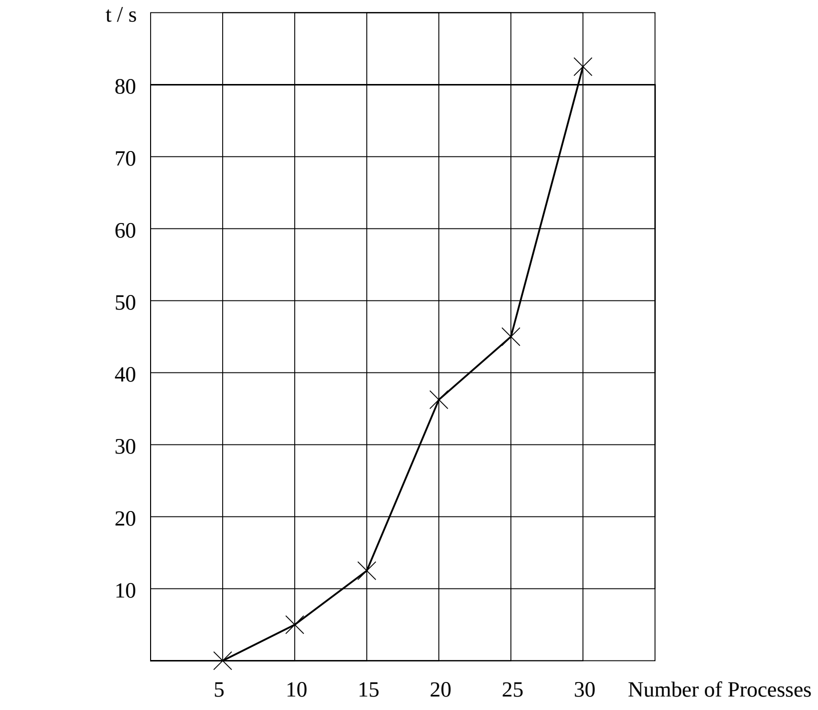


Figure 8.6: Fischer: High-Level TKS Generation Time

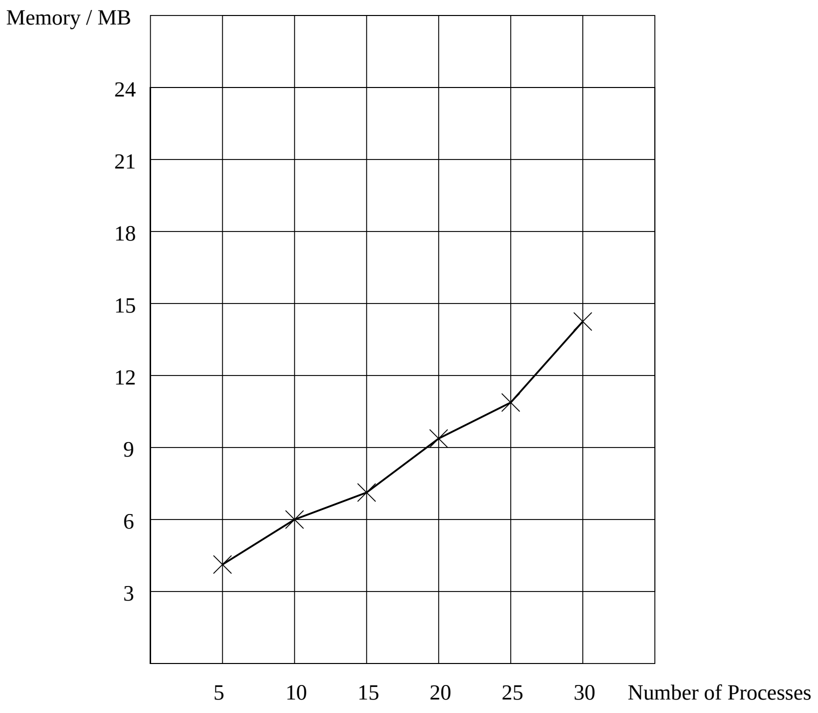


Figure 8.7: Fischer: High-Level TKS Generation Memory

The symbolic breadth-fi rst search traversing of the algorithm Chronos, makes it possible to easily compute large systems, like the one for 30 Fischer processes, which requires 186 boolean variables. The runtimes and the memory consumption show a quadratic growth to the number of processes. The results clearly depend on the variable ordering of the BDDs. For these experiments we used sifting as reordering method, which is a good average solution. Using other reordering methods it is generally possible to obtain also smaller TKSs, but this might also result in worse runtimes.

JERRY offers many variants of initial variable orderings, but also many options in order to benefi t from the variety of functions offered by the CUDD BDD package.

8.1.3 Low-Level TKS Generation - Runtime Analysis

For the generation of low-level TKSs, an exact runtime analysis for the appropriate architecture is necessary (cf. section 6.2). We have performed this for three different architectures:

- Pentium 3, 1GHz, 1GB
- Ultrasparc III, 750MHz, 512MB
- Pentium 4, 2GHz, 512MB

The results for the obtained low-level TKSs and the runtime analysis are given in tables 8.3, 8.5 and 8.4.

The columns of the tables are as follows: The fi rst column denotes the instantiation of the benchmark's parameter (number of processes). The second column shows how many Boolean variables were necessary to encode the state transition diagram and the physical times on the transitions. This means that the system has 2^{s_var} reachable states and the longest transition of the system is not exceeding $2^{t\_var} \times 10^{-5}$ seconds. Column three shows the determined runtimes for the runtime analysis and the low-level TKS generation, which are executed parallel.

Columns four and fi ve show memory consumption, expressed in required BDD nodes and kBytes of memory respectively. The required BDD nodes are given for both, the generated TKS and the obtained UDS (the TKS with untimed transitions), which is then used by the algorithms for all qualitative computations of the JCTL-operators. The last column fi nally, shows the determined runtimes for the minimal and maximal macro steps (in seconds $\times 10^{-5}$) on the target machines.

Note that each transition within the system holds exactly the time, which is needed for its own execution and this time lies always between the minimal and the maximal value. It's interesting to see that, on the Pentium 3, the executable code for four concurrent processes consumes more time (longest maximum transition) than the code for fi ve processes. This shows the necessity of an exact analysis, since an estimation of runtime, based on a few example-data only, can be very misleading.

Fischer: Low-Level TKS Generation - Runtime Analysis								
Number of	Variables		Time	BDD nodes		Memory	Transition [s]$\cdot 10^{-5}$	
Processes	states	time	h:m:s	UDS	TKS	kB	min	max
2	15	2	3.96	268	317	4733	1	3
3	21	3	4.27	418	441	4925	2	4
4	28	4	8.17	559	581	6780	4	8
5	34	3	47.62	693	776	6635	5	7
6	40	4	7:23.99	1068	1167	7824	6	11
7	46	4	1:10:35.62	1487	1635	14752	7	11
Pentium 3, 1GHz, 1GB								

Table 8.3: Low-Level TKS Generation for Fischer's Protocol on Pentium 3 Architecture

Fischer: Low-Level TKS Generation - Runtime Analysis								
Number of	Variables		Time	BDD nodes		Memory	Transition [s]$\cdot 10^{-5}$	
Processes	states	time	h:m:s	UDS	TKS	kB	min	max
2	15	1	2.63	268	270	4733	1	1
3	21	2	2.88	418	449	4925	1	2
4	28	3	4.95	560	618	6780	2	4
5	34	3	29.06	676	771	6751	2	4
6	40	3	4:36.66	1119	1244	7976	3	6
7	46	3	46:59.29	1544	1721	7778	4	6
Pentium 4, 2GHz, 512MB								

Table 8.4: Low-Level TKS Generation for Fischer's Protocol on Pentium 4 Architecture

Fischer: Low-Level TKS Generation - Runtime Analysis								
Number of	Variables		Time	BDD nodes		Memory	Transition [s]$\cdot 10^{-5}$	
Processes	states	time	h:m:s	UDS	TKS	kB	min	max
2	15	4	5.69	268	330	4733	5	8
3	21	4	5.35	326	365	4925	8	12
4	28	5	12.89	577	624	6764	14	21
5	34	5	1:28.20	745	1016	6735	17	27
6	40	6	14:41.80	1110	1306	7904	21	32
7	46	6	2:30:27.29	1506	1668	10400	25	38
Ultrasparc III, 750MHz, 512MB								

Table 8.5: Low-Level TKS Generation for Fischer's Protocol on Ultrasparc III Architecture

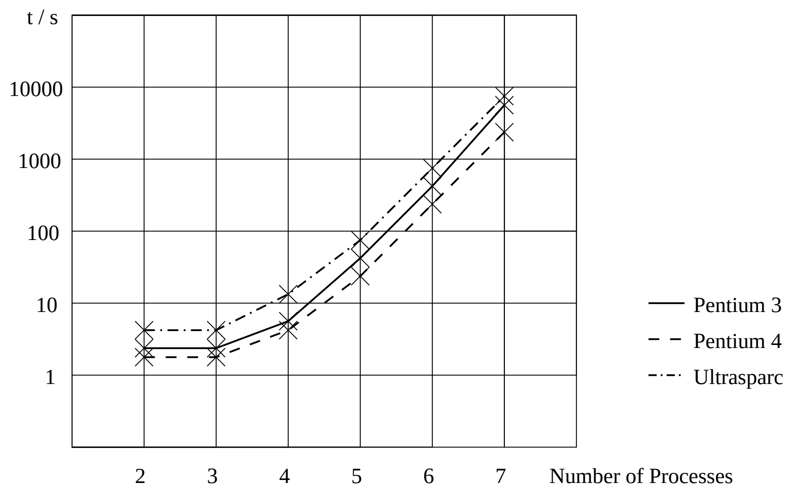


Figure 8.8: Fischer: Low-Level TKS Generation Time - Log. Scale

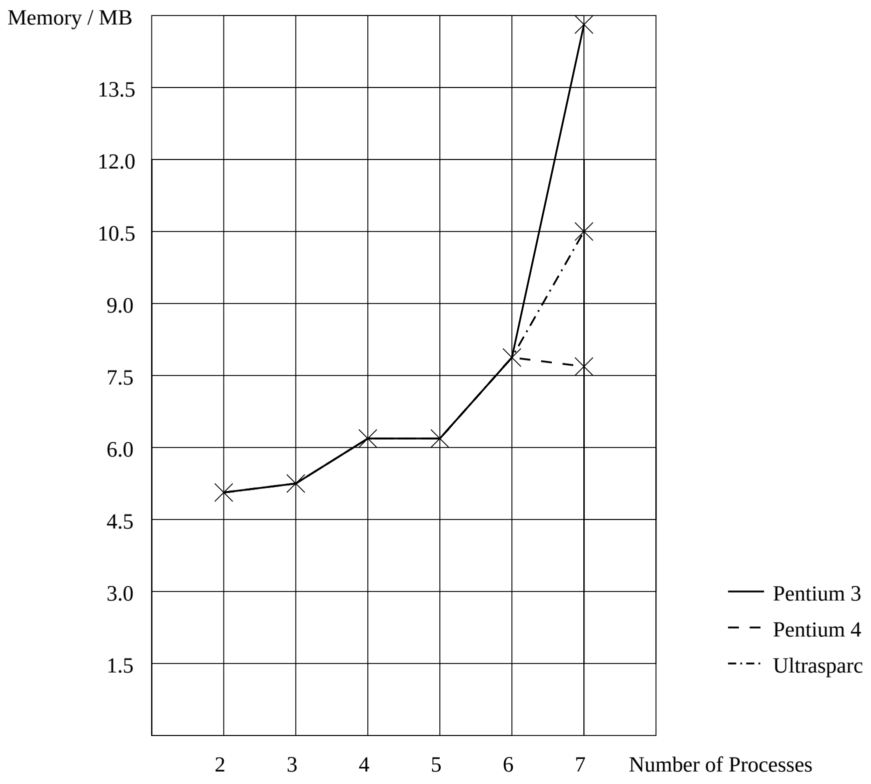


Figure 8.9: Fischer: Low-Level TKS Generation Memory

Considering physical execution times in order to generate the low-level TKS clearly increases the complexity of the system by a formidable time-factor. This time-factor is represented by the additional time-variables.

Nevertheless, it can be seen that JERRY can easily generate low-level formal models for 7 processes. In other words, the low-level performance of JERRY is even better than the high-level performance of the tool Kronos [75] (cf. Fig. 8.9 of section 8.1.7).

Generally, the Pentium 4 machine shows the best performance, while the Ultrasparc III shows the worst one. Of course, the number of required time-variables depends on the required accuracy that one wants to achieve for the examined system, i.e. the grade of the time's granulation. JERRY offers options for manually choosing the appropriate time accuracy, e.g. ms, μs, ns, etc. For this benchmark we have set the time granulation at $\mu s \times 10$, e.g. the longest transitions for 7 and 2 processes in Pentium 3 are 110 μs and 30 μs respectively.

The runtimes for the low-level TKS generation show a fast growth. This is due to the fact, that the execution times of the code are increasing for greater numbers of processes, which in turn results in additional time-variables. This leads to more time-consuming BDD-operations. Please note also that the runtimes depend on the grade of symbolic operations that can be achieved for the specifi c system (see section 6.2.3). This grade is for the Fischer benchmark very unfavorable, but this is not the case for other benchmarks, like the ones presented in sections A.1.2 and A.2.3.

On the other hand, it can be seen that the memory consumption remains very low, which clearly allows the generation of further systems with more than 7 processes, without the need of great amounts of memory.

8.1.4 WCET Analysis

For the WCET and BCET analysis we have obtained the results given in Table 8.6 for n processes. In particular, we have determined the smallest and largest number of macro steps it may take for a process to complete its task, i.e. to reach location *wait*$_3$ (cf. Fig. 8.3).

The symbolic breadth-fi rst search traversing of the algorithm EHLA allows JERRY to easily analyze large systems, like the one for 30 Fischer processes, which requires 186 boolean variables.

Fischer: WCET and BCET analysis - EHLA						
Number of Processes	Variables states	Time h:m:s	Memory kB	BDD nodes	BCET	WCET
5	34	0.53	4908	12264	25	25
10	65	2.19	5261	28616	45	45
15	95	9.53	5960	66430	65	65
20	126	22.44	7384	147168	85	85
25	156	49.18	12997	234038	105	105
30	186	1:06.26	14501	317842	125	125
Pentium 3, 1GHz, 1GB						

Table 8.6: WCET and BCET Analysis for Fischer's Mutex Protocol

According to the results, we can see that the WCET of a system with n processes is $4n + 5$, which would not be so simple to obtain analytically.

The runtimes and the memory consumption show a quadratic growth to the number of processes. In particular, for the runtimes we have approximately $0.02n^2s$ for 5 and 10 processes, $0.04n^2s$ for 15 processes, $0.05n^2s$ for 20 processes and $0.075n^2s$ for 25 and 30 processes.

For the memory consumption we have approximately $0.2n^2$ MB for 5 processes, $0.05n^2$ MB for 10 processes, $0.026n^2$ MB for 15 processes, $0.018n^2$ MB for 20 processes, $0.02n^2$ MB for 25 processes, $0.016n^2$ MB for 30 processes. Furthermore, we can see that the logical steps for BCET and WCET are identical. This is due to the fact, that all processes want to enter the critical region at the same time, while only one is allowed to do so. Hence, there are no points of time, where the critical region remains empty.

The results clearly depend on the variable ordering of the BDDs. For these experiments we used sifting as reordering method, which is a good average solution. Using other reordering methods it is generally possible to achieve better memory consumption, but this might also result in worse runtimes.

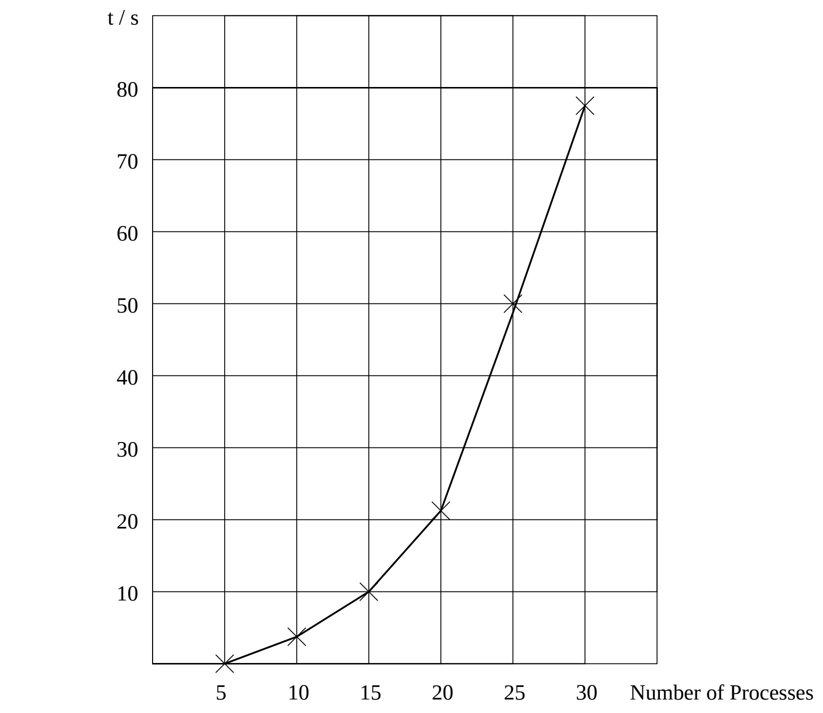


Figure 8.10: Fischer: EHLA Time

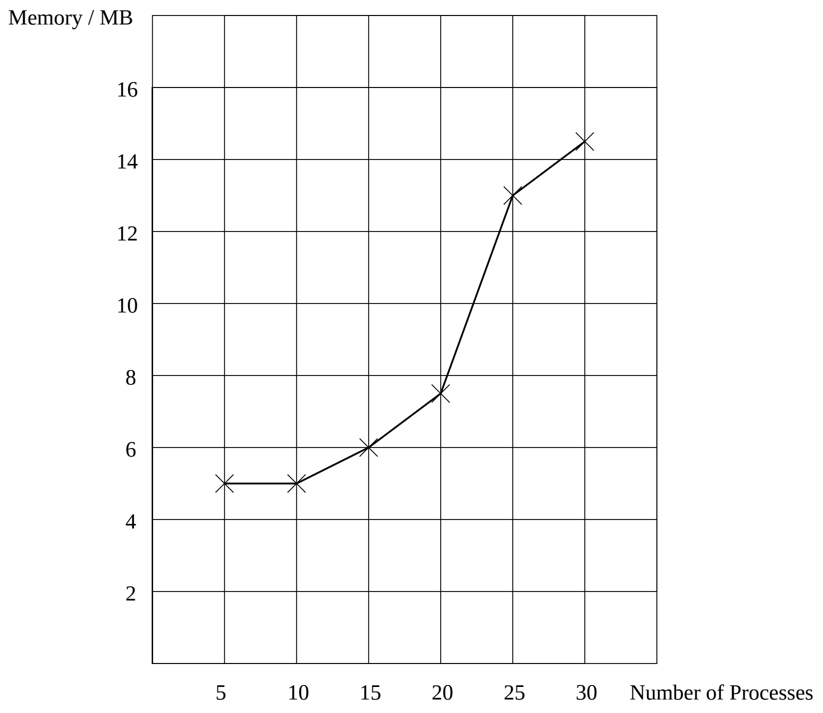


Figure 8.11: Fischer: EHLA Memory

8.1.5 UDS Verification

First we verifi ed qualitative-only properties at UDS-level. These properties guarantee that two different processes will not be present in the critical section at the same time. To prove this, we check all possible combinations of all possible processes. In other words, the number of specifi cations to be checked grows with the number of processes as follows:

- for 5 processes 20 different specifi cations
- for 10 processes 90 different specifi cations
- for 15 processes 210 different specifi cations
- for 20 processes 380 different specifi cations
- for 25 processes 600 different specifi cations
- for 30 processes 870 different specifi cations

Fischer: UDS verifi cation				
Number of Processes	Variables states	Time h:m:s	Memory kB	BDD nodes
5	34	0.42	4786	7154
10	65	1.90	5027	18369
15	95	6.05	5336	33726
20	126	20.58	6172	80738
25	156	33.04	6630	106288
30	186	51.60	7180	135926
Pentium 3, 1GHz, 1GB				

Table 8.7: Verifi cation of Qualitative Properties at UDS-Level for Fischer's Mutex Protocol

Due to the increasing number of specifi cations to be verifi ed, the runtimes and the memory consumption show a quadratic growth to the number of processes (cf. Table 8.7). In particular, for the runtimes we have approximately $0.02n^2s$ for 5, 10 and 15 processes, and $0.05n^2s$ for 20, 25 and 30 processes. For the memory consumption we have approximately $0.2n^2$ MB for 5 processes, $0.05n^2$ MB for 10 processes, $0.023n^2$ MB for 15 processes, $0.015n^2$ MB for 20 processes, $0.01n^2$ MB for 25 processes, $0.008n^2$ MB for 30 processes.

These values are satisfactory to the praxis: as can be seen in Table 8.7, JERRY needs only 51.6 seconds and 7180 kB of main memory to check 870 specifi cations in a system with 2^{186} states. Again, the results depend on the variable ordering of the BDDs. For these experiments we used sifting as reordering method, which is a good average solution. Using other reordering methods it is generally possible to obtain also smaller TKSs, but this might also result in worse runtimes.

8.1.6 High-Level TKS Verification

Fischer’s mutex protocol is well-suited for checking and comparing the performance of different JCTL-operators. For this purpose, we have tested the operators

- $\mathsf{EF}^{\kappa}\varphi \equiv \mathsf{E}[1\ \underline{\mathsf{U}}^{\kappa}\ \varphi]$
- $\mathsf{AF}^{\kappa}\varphi \equiv \neg\mathsf{EG}^{\kappa}\neg\varphi$
- $\mathsf{AG}^{\kappa}\varphi \equiv \neg\mathsf{E}[1\ \underline{\mathsf{U}}^{\kappa}\ \neg\varphi]$

according to the results of WCET and BCET analysis. In particular, we have proved the correctness of the WCET and BCET analysis by checking the following real-time specifi cations:

- Property 1: $\mathsf{EF}\ ^{=WCET}\Phi$
- Property 2: $\mathsf{AF}\ ^{=WCET}\Phi$
- Property 3: $\mathsf{AG}\ ^{=WCET}\Phi$

where Φ stands for: “all processes have visited the critical section and switched to their fi nal state”. Note that, properties 2 and 3 are true, due to the fact that in the Fischer benchmark we have BCET = WCET (cf. Table 8.6). The verifi cation results of Table 8.8 clearly show that also the quantitative JCTL-operators can easily handle very large systems.

The results of this section clearly demonstrate JERRY’s stable functioning: As can be seen in Table 8.8 and Figures 8.12 and 8.13, the operators EF^{κ} and AG^{κ} show almost the same behavior for their runtimes and memory consumption. This is an expected result, due to the duality of these operators (cf. defi nition 13 of section 3.2.2).

The runtimes and memory consumption of EF^{κ} and AG^{κ} show a quadratic growth to the number of processes. In particular, for the runtimes we have approximately $0.02n^2s$ for 5 and 10 processes, $0.05n^2s$ for 15 processes, $0.07n^2s$ for 20 and 25 processes and $0.09n^2s$ for 30 processes. For the memory consumption we have approximately $0.2n^2$ MB for 5 processes, $0.05n^2$ MB for 10 processes, $0.02n^2$ MB for 15 and 25 processes and $0.015n^2$ MB for 20 and 30 processes.

In contrast to EF^{κ} and AG^{κ}, the runtimes and memory consumption of the AF^{κ} operator show a cubic growth to the number of processes. This is also an expected result, due to its duality to the EG^{κ} operator (cf. defi nition 13 of section 3.2.2): EG^{κ} uses additional BDD-operations in order to compute the set of states required for each iteration (cf. Figure 5.5 of section 5.1).

In particular, for the runtimes we have approximately $0.003n^3s$ for 5 processes, $0.01n^3s$ for 10 processes and $0.015n^3s$ for 15, 20, 25 and 30 processes. For the memory consumption we have approximately $0.04n^3$ MB for 5 processes, $0.009n^3$ MB for 10 processes, $0.004n^3$ MB for 15 processes, $0.002n^3$ MB for 20 and 25 processes and $0.001n^3$ MB for 30 processes.

The results clearly depend on the variable ordering of the BDDs. For these experiments we used sifting as reordering method, which is a good average solution. Using other reordering methods it is generally possible to achieve better memory consumption, but this might also result in worse runtimes.

Fischer: High-Level Verifi cation - Prop. 1					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
5	34	2	0.39	4812	6132
10	65	2	2.68	5169	22484
15	95	2	11.74	5735	52122
20	126	2	28.67	6375	85848
25	156	2	48.11	11840	162498
30	186	2	1:18.09	13694	274918
Pentium 3, 1GHz, 1GB					

Fischer: High-Level Verifi cation - Prop. 2					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
5	34	2	0.40	4828	7154
10	65	2	9.18	8992	246302
15	95	2	56.67	13596	523264
20	126	2	1:34.66	15906	645904
25	156	2	4:35.25	31136	1278522
30	186	2	6:56.66	36615	1603518
Pentium 3, 1GHz, 1GB					

Fischer: High-Level Verifi cation - Prop. 3					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
5	34	2	0.40	4828	7154
10	65	2	2.60	5286	20939
15	95	2	11.54	6349	89936
20	126	2	28.69	8466	211554
25	156	2	47.45	11658	395514
30	186	2	1:17.78	12202	427196
Pentium 3, 1GHz, 1GB					

Table 8.8: Verifi cation of Quantitative Properties at High-Level for Fischer's Mutex Protocol

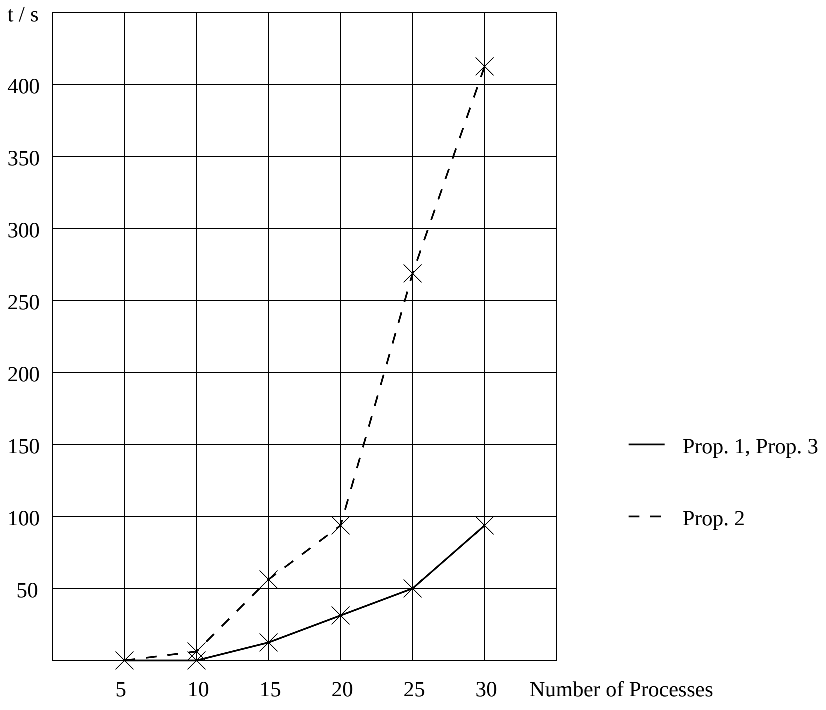


Figure 8.12: Fischer: High-Level TKS Verifi cation Time

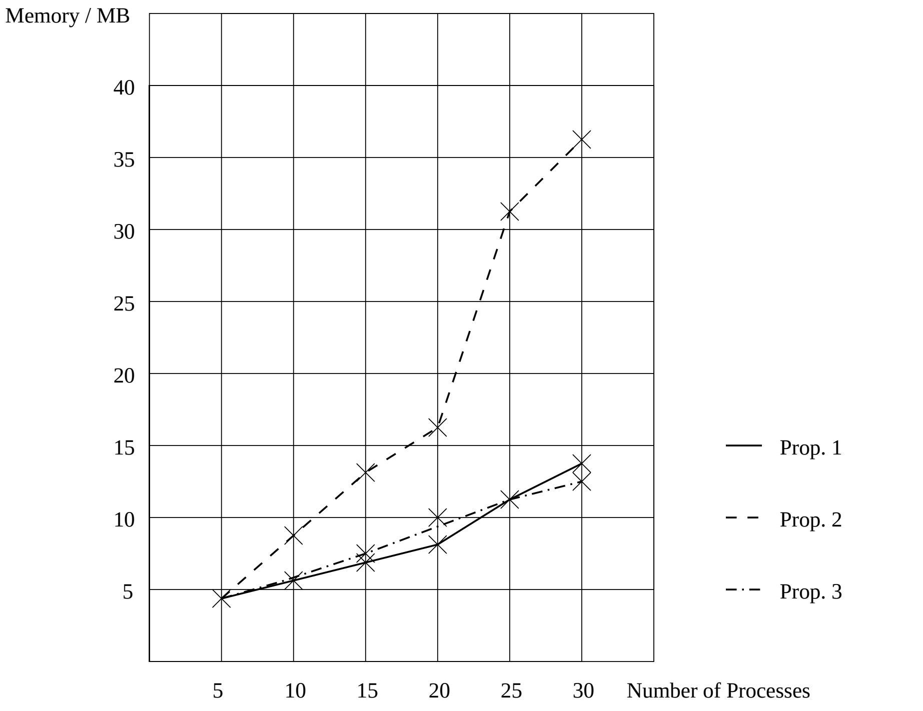


Figure 8.13: Fischer: High-Level TKS Verifi cation Memory

8.1.7 A Comparison to Other Tools

In order to compare JERRY′s performance, we have used some of the most popular tools available, to verify Fischer's mutex protocol. In particular, we have tested Kronos [75], UPPAAL [126] and Cadence-SMV [26]. The results are listed in the Tables 8.9, 8.10, 8.11 and 8.12 shown below.

Fischer: Verification with Kronos					
Number of Processes	Model Construction h:m:s	Memory kB	Verification h:m:s	Memory kB	Total Runtime h:m:s
2	0.10	1232	0.10	1600	0.20
3	0.10	1422	0.10	1876	0.20
4	0.90	1766	0.30	2476	1.20
5	1.36	11020	1.31	16200	2.67
6	1:50.10	73500	8.62	85132	1:58.72
Pentium 3, 1GHz, 1GB					

Table 8.9: Verification of Fischer's Mutex Protocol with Kronos

Fischer: Verification with UPPAAL		
Number of Processes	Time h:m:s	Memory kB
2	0.03	1780
3	0.04	1820
4	0.06	1910
5	0.30	2010
6	1.54	2820
7	8.33	6332
8	40.43	17548
9	2:21.09	60596
10	15:55.23	213000
11	>1.5h	
Pentium 3, 1GHz, 1GB		

Table 8.10: Verification of Fischer's Mutex Protocol with UPPAAL

Furthermore, an overall comparison of all tools is given in Table 8.13. The tool Kronos showed the worst performance of all tested tools. Kronos allowed the verification only up to 6 Fischer processes, due to a limitation on its control states: The default value is set at 2048, where the message "max control states exceeded" appears. We have changed this value manually to the maximum allowed of 32000 and verified up to 6 processes with this. However, for 7 processes was this value not enough.

With the tool UPPAAL we have used the options -A ("use convex hull approximation") and -T ("optimize time consumption when several properties are examined"). Without these options

Fischer: Verifi cation with Cadence-SMV				
Number of Processes	Variables states	Time h:m:s	Memory kB	BDD nodes
2	15	0.05	2365	6765
3	21	0.10	2678	11828
4	28	0.13	3401	22951
5	34	0.19	3420	24378
6	40	0.26	3540	27820
7	46	0.51	4312	50415
8	53	0.82	4630	62088
9	60	0.92	4802	81267
10	65	1.68	4640	76131
11	72	12.13	6072	26686
15	95	35.62	6208	31011
20	126	2:16.72	10966	67860
25	156	5:00.63	16360	81707
30	186	12:02.79	23988	129180
Pentium 3, 1GHz, 1GB				

Table 8.11: Verifi cation of Fischer's Mutex Protocol with Cadence-SMV

Fischer: Verifi cation with JERRY				
Number of Processes	Variables states	Time h:m:s	Memory kB	BDD nodes
2	15	0.10	4754	7154
3	21	0.24	4765	7154
4	28	0.41	4855	12264
5	34	0.55	4898	14308
6	40	0.96	5111	26572
7	46	1.78	5449	47012
8	53	3.71	5797	67452
9	60	3.30	5857	70518
10	66	6.51	6807	126728
11	72	11.21	7629	174762
15	95	16.95	7250	151256
20	126	35.20	14585	340326
25	156	1:13.41	16648	460922
30	186	3:49.52	19877	652036
Pentium 3, 1GHz, 1GB				

Table 8.12: Verifi cation of Fischer's Mutex Protocol with JERRY

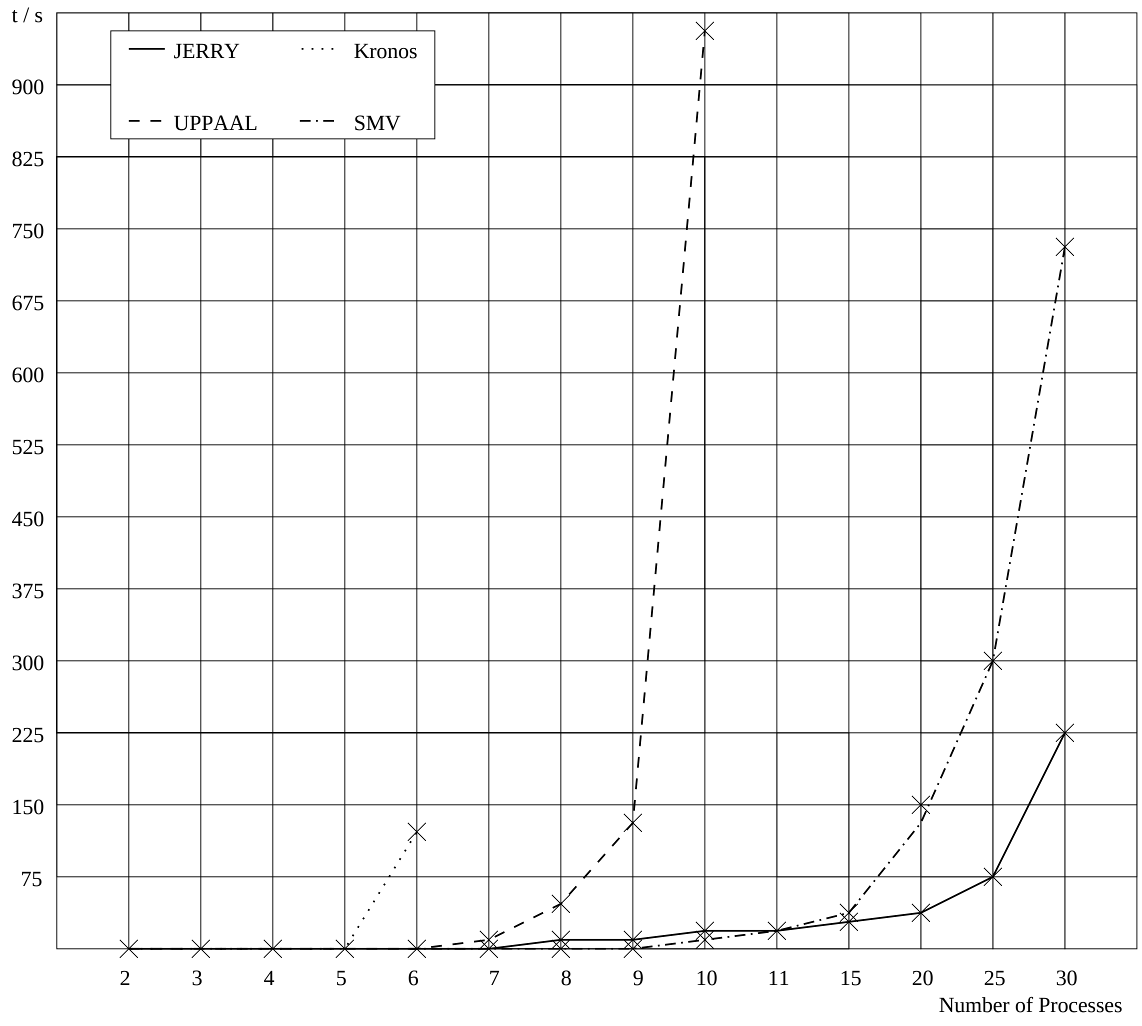


Figure 8.14: Fischer: Verifi cation Time

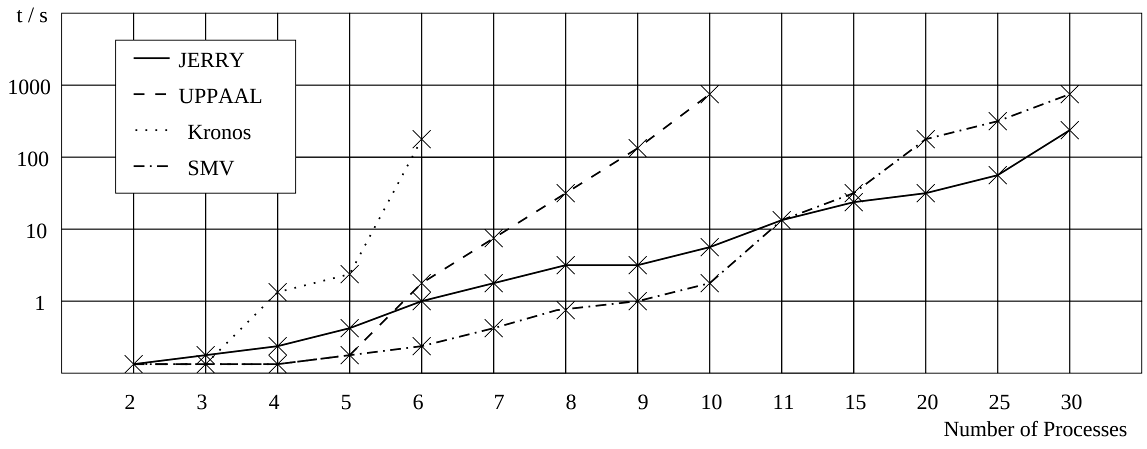


Figure 8.15: Fischer: Verifi cation Time - Log. Scale

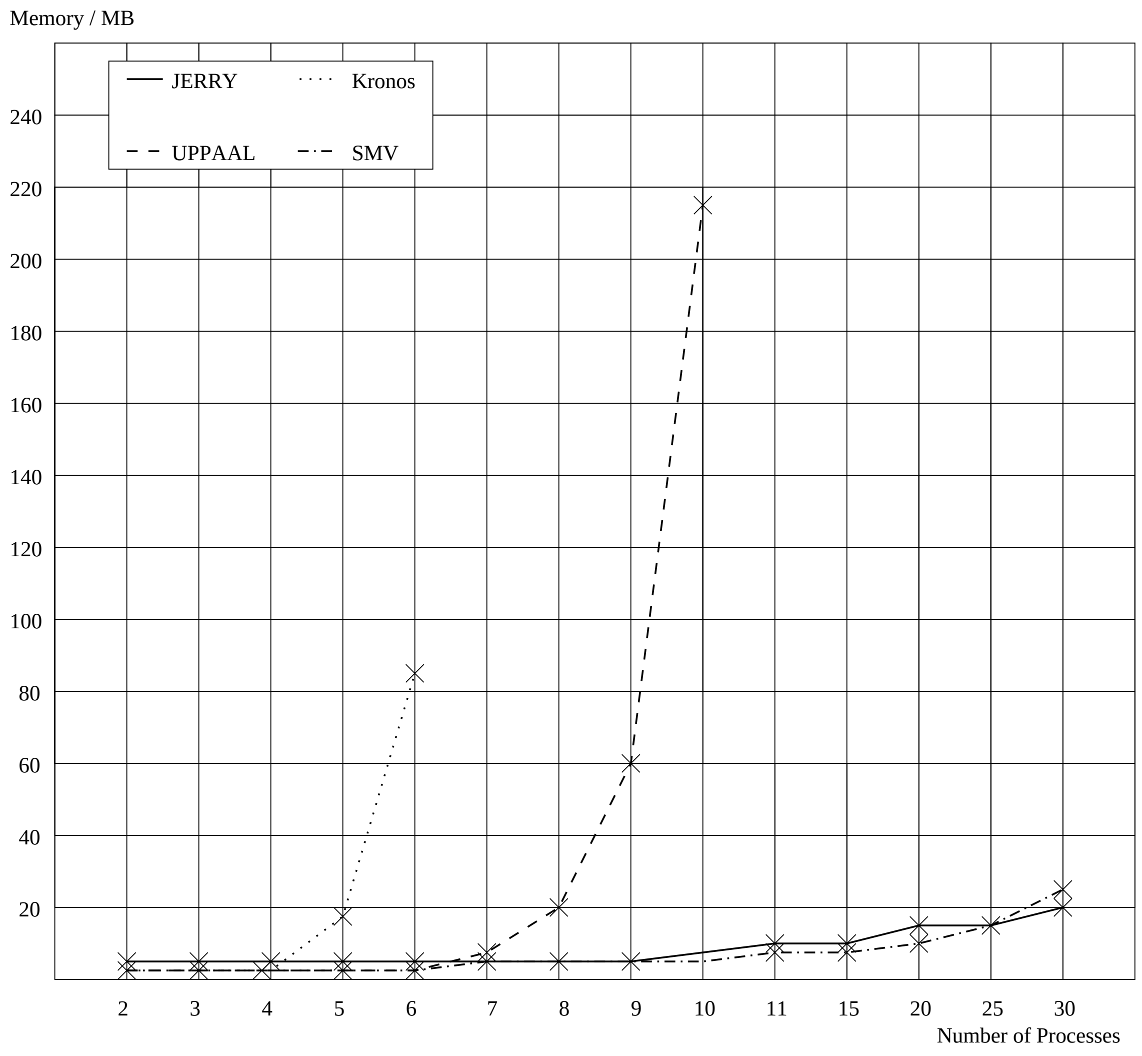


Figure 8.16: Fischer: Verification Memory

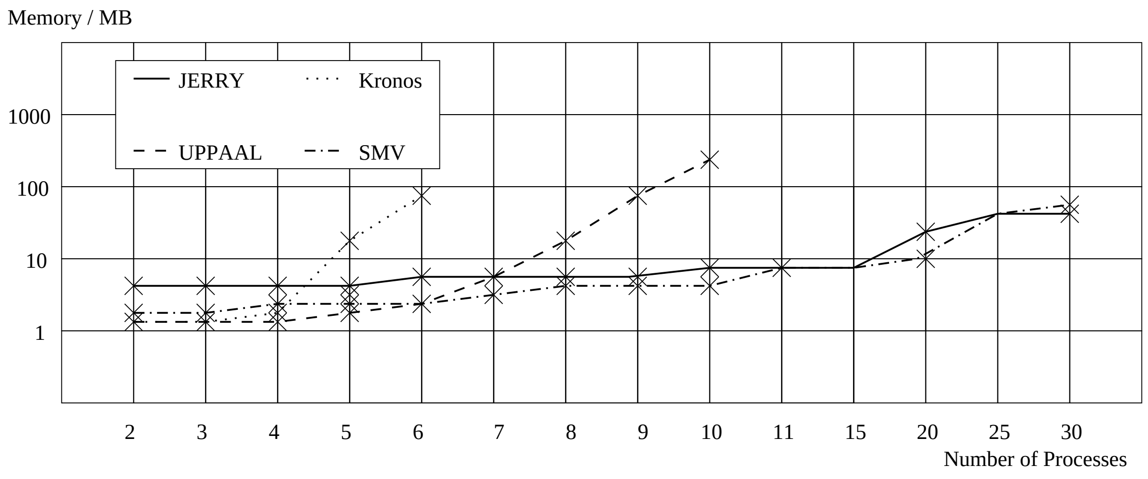


Figure 8.17: Fischer: Verification Memory - Log. Scale

the tool's performance decreases significantly and its results become worse than the Kronos' results. Using the options, we were able to verify up to 10 processes. We also started a run with 11 processes, which seemed not to come to the end. We decided to interrupt it after 1.5 hour.

Fischer: Overview								
Number of Processes	Time h:m:s				Memory kB			
	Kronos	UPPAAL	SMV	JERRY	Kronos	UPPAAL	SMV	JERRY
2	0.20	0.03	0.05	0.10	1600	1780	2365	4754
3	0.20	0.04	0.10	0.24	1876	1820	2678	4765
4	1.20	0.06	0.13	0.41	2476	1910	3401	4855
5	2.67	0.30	0.19	0.55	16200	2010	3420	4898
6	1:58.72	1.54	0.26	0.96	85132	2820	3540	5111
7	-	8.33	0.51	1.78	-	6332	4312	5449
8	-	40.43	0.82	3.71	-	17548	4630	5797
9	-	2:21.09	0.92	3.30	-	60596	4802	5857
10	-	15:55.23	1.68	6.51	-	213000	4640	6807
11	-	>1.5h	12.13	11.21	-	-	6072	7629
15	-	-	35.62	16.95	-	-	6208	7205
20	-	-	2:16.72	35.20	-	-	10966	14585
25	-	-	5:00.63	1:13.41	-	-	16360	16648
30	-	-	12:02.79	3:49.52	-	-	23988	19877
Pentium 3, 1GHz, 1GB								

Table 8.13: Verification of Fischer's Mutex Protocol - Comparison

Having experimented with several SMV-versions, like CMU SMV 2.4.3, 2.4.4. and 2.4.5. [93], but also NuSMV [31], which is based on the CUDD BDD-package, we found the Cadence-SMV [26] to have the best performance of all. In our experiments, none of the other SMV-versions could handle more than 15 Fischer processes. To our knowledge, this could be the result of a better BDD-variable ordering, used by Cadence-SMV. For the Cadence-SMV experiments we have used the BDD reordering option with sifting.

As can be seen, JERRY performs much better than all other tools, when large systems are considered. For small systems up to 10 processes, the Cadence-SMV shows a slightly better performance. This is due to the fact that JERRY uses many subroutines in order to classify, store and prepare data for later use, necessary for its modular functioning. These data can then be reused in order to perform a system's analysis at different stages, without having to start everything from the beginning.

Furthermore, as the curves of Figures 8.14, 8.15, 8.16 and 8.17 show, JERRY's behavior remain more flat and stable as the system's size grows.

8.1.8 Low-Level TKS Verification

JERRY's capabilities are best to see in a low-level verifi cation. Here, the original system is endowed by additional physical times, required for the code execution of the synchronous program on specifi c architectures (cf. section 6.2). This results in an enormous increase of the system's complexity and gives us the opportunity to demonstrate JERRY's sophisticated behavior.

Please note that we can not compare the low-level results to any other tool, since there exist no other tools capable of low-level verifi cation. Nevertheless, it is a challenge for us to check JERRY with such high complex systems.

Similar to the high-level verifi cation, we tested different JCTL-operators according to the results of WCET and BCET analysis. In particular, we have proved the correctness of the WCET and BCET analysis by checking the following real-time specifi cations:

- Property 4: $\mathsf{EF}^{\leq\ (WCET\ \times\ max\_trans)}\Phi$
- Property 5: $\mathsf{AF}^{\leq\ (WCET\ \times\ max\_trans)}\Phi$
- Property 6: $\mathsf{AG}^{\geq\ (WCET\ \times\ max\_trans)}\Phi$

where Φ stays for: "all processes have visited the critical section and switched to their fi nal state". max_trans is the maximum transition included in the system, as can be obtained from the Tables 8.3, 8.5 and 8.4. Note that, properties 2 and 3 are true, due to the fact that in the Fischer benchmark we have BCET = WCET (cf. Table 8.6).

The verifi cation results are shown in Tables 8.14, 8.15 and 8.16. In order to give an overview and compare different architectures, we consider also comparisons of the used target machines, shown in Figures 8.18, 8.19, 8.20, 8.21, 8.22, and 8.23. Generally, the model obtained for the Pentium 4 architecture benefi ts from having less time variables, due to faster code execution, which leads to better results. However, due to variable-reordering procedures, the Pentium 4 model has in many cases more BDD-nodes than the other two models (cf. Tables 8.3, 8.4 and 8.5 in section 8.1.3). This reduces sometimes signifi cantly the performance for this model. For example, for the verifi cation of properties 5 and 6 for 7 processes, the Pentium 4 model shows higher memory consumption (kBytes and BDD-nodes) than the Pentium 3 model. The Ultrasparc III model has the longest timed transitions, which increases its complexity and leads to worse model-checking results by the BDD-operators.

It's clearly to see that all operators have a good, stable performance and can easily handle the system. Moreover these results demonstrate the importance of JERRY's modular design: As can be seen by comparing with Tables 8.3, 8.5 and 8.4, the main time consuming factor for the low-level verifi cation is not the verifi cation itself, but the generation of the low-level model, performed in section 8.1.3. JERRY has the capability of storing the model together with all other important information and hence, can start the analysis directly at the low-level verifi cation stage, overcoming repeated (and needless) model generations.

The verification runtimes are too low (max. 3.42 seconds for Ultrasparc III) to allow us to make solid statements about their behavior. Moreover, the memory consumption is here mainly caused by the initialization and loading of the formal model and not by the model-checking procedures, which also makes difficult the judgment of their behavior. Nevertheless, we can recognize a quadratic growth to the number of processes for all target machines and all checked properties.

This is in accordance with the results obtained for the operators EF^{κ} and AG^{κ} in section 8.1.6. Property 5 is checking the operator AF^{κ}, which showed a cubic growth to the number of processes in section 8.1.6, due to its duality to the EG^{κ} operator (cf. definition 13 of section 3.2.2). Here, only a quadratic growth can be experienced, but this might be the result of the small number of iterations required to check the entire system. Generally, the EG^{κ} operator uses additional BDD-operations in order to compute the set of states required for each iteration (cf. Figure 5.5 of section 5.1).

The results clearly depend on the variable ordering of the BDDs. For these experiments we used sifting as reordering method, which is a good average solution. Using other reordering methods it is generally possible to achieve better memory consumption, but this might also result in worse runtimes.

Fischer: Low-Level Verifi cation - Pentium 3 - Prop. 4					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
2	15	2	0.06	4701	3066
3	21	3	0.25	4770	6132
4	28	4	0.37	4871	11242
5	34	3	0.53	4908	12264
6	40	4	1.05	5251	31682
7	46	4	1.37	5561	50078
Pentium 3, 1GHz, 1GB					

Fischer: Low-Level TKS Verifi cation - Pentium 3 - Prop. 5					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
2	15	2	0.04	4701	3066
3	21	3	0.33	4754	5110
4	28	4	0.56	4807	7154
5	34	3	0.73	4892	11242
6	40	4	1.56	4979	15330
7	46	4	2.08	5067	19418
Pentium 3, 1GHz, 1GB					

Fischer: Low-Level TKS verifi cation - Pentium 3 - Prop. 6					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
2	15	2	0.05	4701	3066
3	21	3	0.34	4754	5110
4	28	4	0.51	4807	7154
5	34	3	0.68	4876	10220
6	40	4	1.61	4962	14308
7	46	4	2.13	5099	21462
Pentium 3, 1GHz, 1GB					

Table 8.14: Verifi cation of Quantitative Properties at Pentium 3 for Fischer's Mutex Protocol

Fischer: Low-Level TKS verifi cation - Pentium 4 - Prop. 4					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
2	15	1	0.03	4685	2044
3	21	2	0.20	4754	5110
4	28	3	0.30	4791	6132
5	34	3	0.63	4940	14308
6	40	3	0.54	4898	10220
7	46	3	0.99	5079	20440
Pentium 3, 1GHz, 1GB					

Fischer: Low-Level TKS verifi cation - Pentium 4 - Prop. 5					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
2	15	1	0.03	4685	2044
3	21	2	0.21	4754	5110
4	28	3	0.49	4839	9198
5	34	3	0.78	4844	8176
6	40	3	0.81	4993	16352
7	46	3	1.59	5147	24528
Pentium 3, 1GHz, 1GB					

Fischer: Low-Level TKS verifi cation - Pentium 4 - Prop. 6					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
2	15	1	0.03	4685	2044
3	21	2	0.22	4754	5110
4	28	3	0.47	4839	9198
5	34	3	0.78	4844	8176
6	40	3	0.79	4977	15330
7	46	3	1.58	5161	25550
Pentium 3, 1GHz, 1GB					

Table 8.15: Verifi cation of Quantitative Properties at Pentium 4 for Fischer's Mutex Protocol

Fischer: Low-Level TKS verifi cation - Ultrasparc - Prop. 4					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
2	15	4	0.04	4717	4088
3	21	4	0.21	4802	8176
4	28	5	0.41	4956	16352
5	34	5	0.45	4863	9198
6	40	6	0.97	5473	44968
7	46	6	1.64	6367	97090
Pentium 3, 1GHz, 1GB					

Fischer: Low-Level TKS verifi cation - Ultrasparc - Prop. 5					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
2	15	4	0.07	4765	7154
3	21	4	0.37	4754	5110
4	28	5	0.89	4856	10220
5	34	5	0.79	4910	12264
6	40	6	1.71	5032	18396
7	46	6	3.41	5221	28616
Pentium 3, 1GHz, 1GB					

Fischer: Low-Level TKS verifi cation - Ultrasparc - Prop. 6					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
2	15	4	0.06	4765	7154
3	21	4	0.37	4754	5110
4	28	5	0.84	4840	9198
5	34	5	0.76	4911	12264
6	40	6	1.62	5016	17374
7	46	6	3.31	5253	30660
Pentium 3, 1GHz, 1GB					

Table 8.16: Verifi cation of Quantitative Properties at Ultrasparc III for Fischer's Mutex Protocol

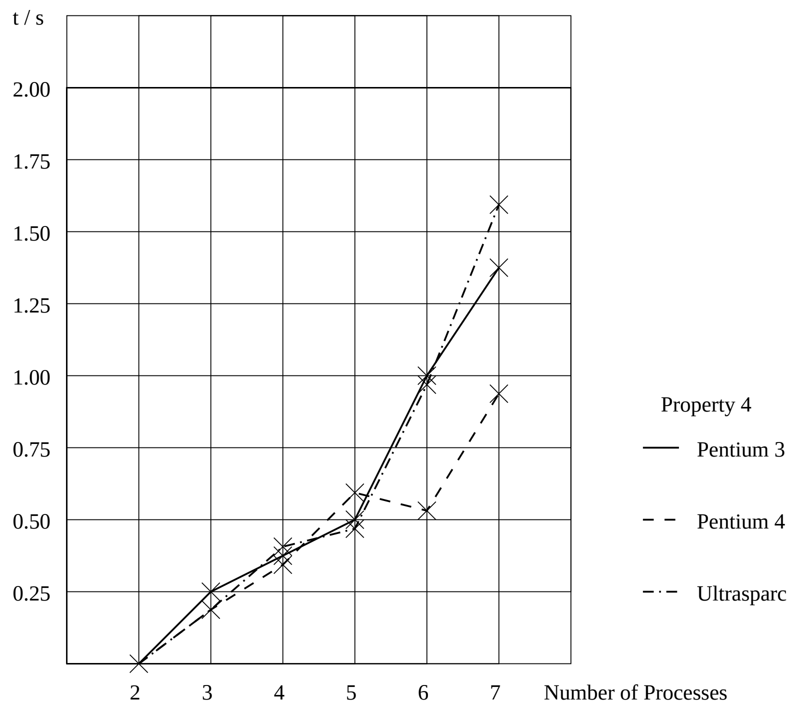


Figure 8.18: Fischer: Low-Level TKS Verifi cation Time - Property 4

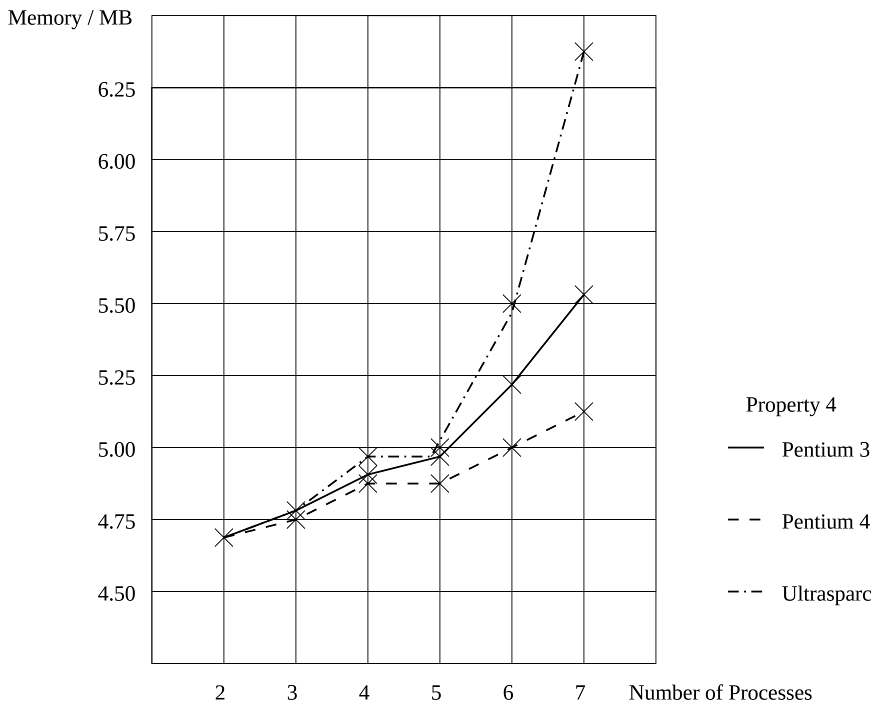


Figure 8.19: Fischer: Low-Level TKS Verifi cation Memory - Property 4

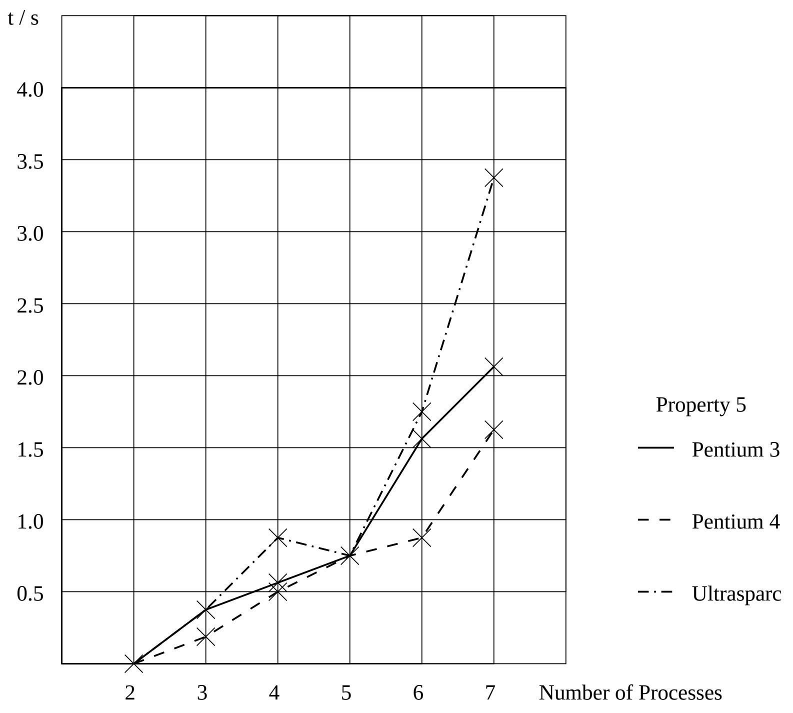


Figure 8.20: Fischer: Low-Level TKS Verifi cation Time - Property 5

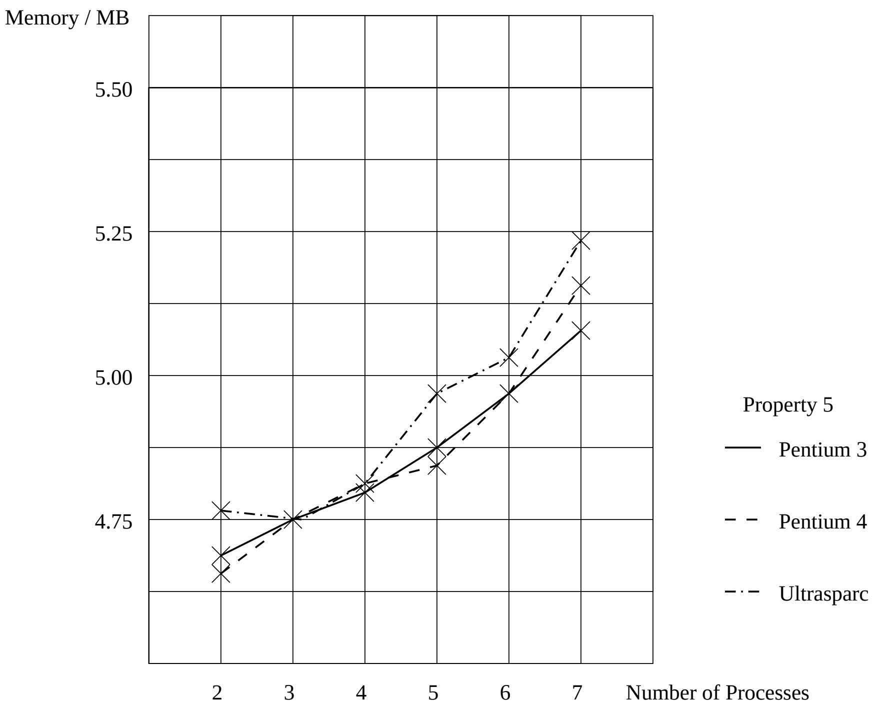


Figure 8.21: Fischer: Low-Level TKS Verifi cation Memory - Property 5

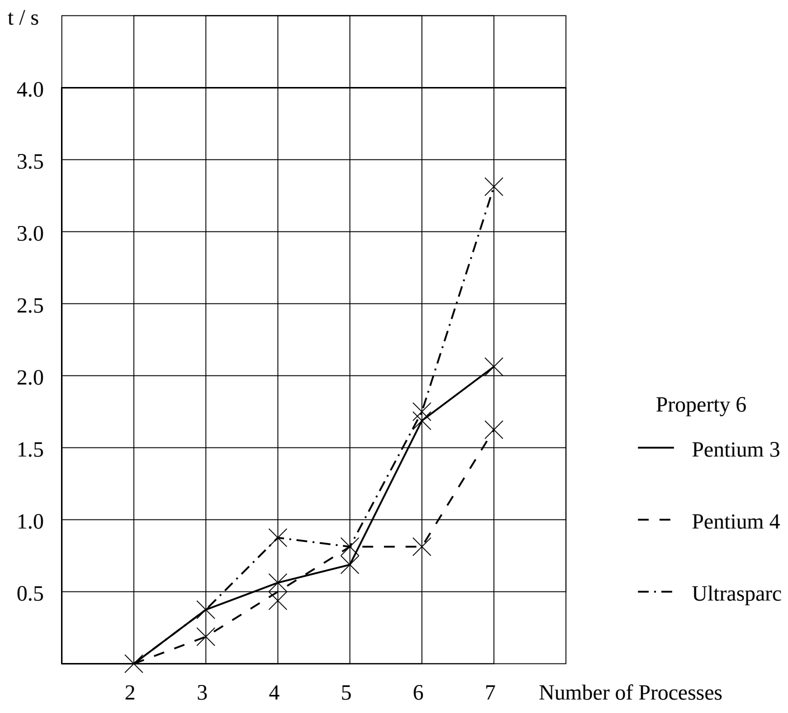


Figure 8.22: Fischer: Low-Level TKS Verifi cation Time - Property 6

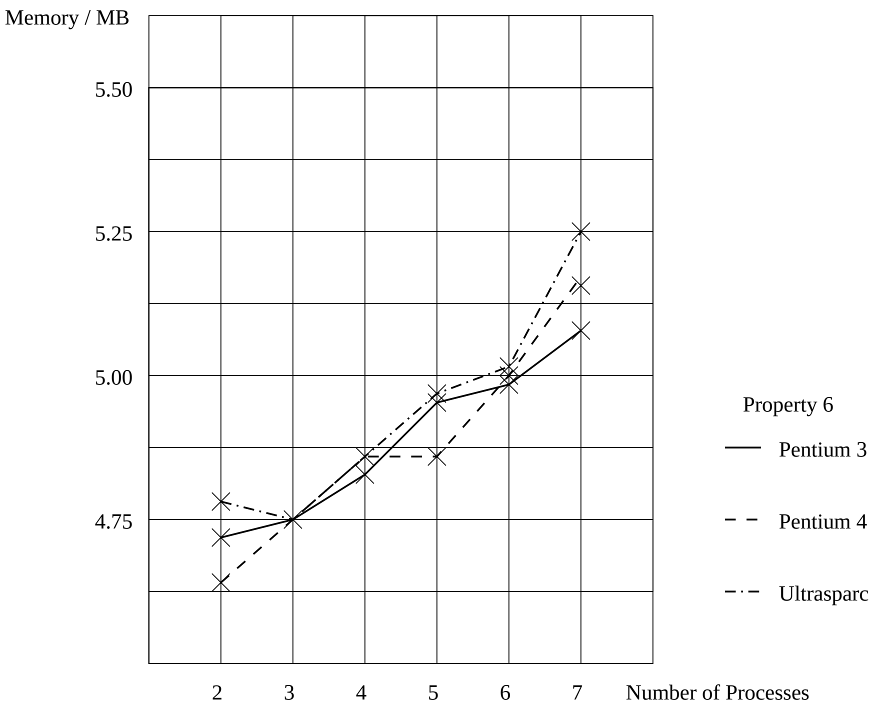


Figure 8.23: Fischer: Low-Level TKS Verifi cation Memory - Property 6

Appendix A

Further Experimental Results

Ὀρθόν ἀλήθει' ἀεί. [1]

ΣΟΦΟΚΛΗΣ (495 - 406 π.Χ.)

(Αντιγόνη, 1195)

A.1 Bus Arbiter

Bus-based systems consist of a central bus with several users, who use this as a message-channel for the exchange of information. A central problem in this context is the allocation of the bus to different users, so that no writing conflicts occur. This method of allocation is called arbitration. The idea for the arbitration process used here originates from the DMA-controller of Martin [94].

If some processes signal the necessity for a bus access, then the bus arbiter has to choose one of them and inform it that the access on the bus is permitted. After that, in order to ensure access exclusivity on the central bus, the arbiter has to wait for the end of the access to take further decisions. This will be given by a respective signal by the active process and informs that the bus is free and hence, access rights can be given again by the arbiter.

The implementation in Quartz represents several concurrent processes depending on the number of possible bus users. The processes can be divided into three categories: a process for the rotation of the tokens, a process for each bus user to set the persistence as well as a process for the output of the bus-release signals. The implementation of the arbiter is given in Fig. A.1. The signals $req_1, ..., req_n$ represent the bus-request signals of a process i, rdy is the signal after the end of an access and $ack_1, ..., ack_n$ represent the signaling of the bus allocation to a process i.

[1] Truth is eternally sure.
SOFOCLES (495 - 406 BC)

```
**module** $arbitrate(req_1, \cdots, req_n, rdy, ack_1, \cdots, ack_n)$
  $loop$
    $tok_1 : await\ arb;$
    $\vdots$
    $tok_n : await\ arb;$
  $end$
  $||$
  $loop$
    $pr_{01} : await\ (req_1 \wedge tok_1);$
    $pr_{11} : await\ !req_1$
  $end$
  $||$
  $\vdots$
  $||$
  $loop$
    $pr_{0n} : await\ (req_n \wedge tok_n);$
    $pr_{1n} : await\ !req_n$
  $end$
  $||$
  $loop$
    $arb : await(\bigvee_{i=1}^{n}(req_i);$
    **if** $\bigvee_{i=1}^{n}(tok_i \wedge pr_{1i})$ **then**
      **case** $req_1 \wedge tok_1 \wedge pr_{11} : emit\ ack_1;$
      $\vdots$
      **case** $req_n \wedge tok_n \wedge pr_{1n} : emit\ ack_n;$
    **else**
      **case** $req_1 : emit\ ack_1;$
      $\vdots$
      **case** $req_n : emit\ ack_n;$
    **end**;
    $wfa : await\ rdy;$
  $end;$
```

Figure A.1: Arbiter for n Processes in Quartz

A.1.1 UDS Generation

Starting the automatic formal analysis, we first consider again the generation of the unit-delay structure. Table A.1 shows the results for arbiter's UDS generation for n processes. All experiments were run on a Pentium 3 with 1GHz and 1GByte of main memory.

The columns of the table are as follows: The first column denotes the instantiation of the benchmark's parameter (number of processes). The second column shows how many Boolean variables were necessary to encode the state transition diagram, which means that the system has 2^{var} reachable states. Column three shows the determined runtimes for the UDS generation. Columns four and five show memory consumption, expressed in required BDD nodes and kBytes of memory respectively.

Arbiter: UDS generation				
Number of Processes	Variables states	Time h:m:s	BDD nodes	Memory kB
5	29	0.56	2046	5047
10	54	2.80	9104	7397
15	79	13.54	12304	20556
20	104	24.61	30527	20100
25	129	53.96	48461	30077
30	154	2:34.03	45312	50073
Pentium 3, 1GHz, 1GB				

Table A.1: UDS Generation for Arbiter

As can be seen, JERRY has no difficulties to generate a large system for 30 arbiter processes, requiring 154 boolean variables. This means that the system includes 2^{154} reachable states.

Please note that the runtimes include also the time needed for exporting and storing the formal model and all other needed data on hard disk.

Note also that due to the structure of the arbiter benchmark it is not possible to perform a WCET analysis, since it can not be guarantied that a process will finish its access on the bus and leave it free for other processes, i.e. there exist paths in the system that can circulate infinitely, so that a WCET can not be determined (cf. section 7.1.1). This structure, combined with high data dependency leaves also no space for a reasonable abstraction. Thus, we concentrate more on low-level real-time verification, as well as high-level verification of qualitative properties.

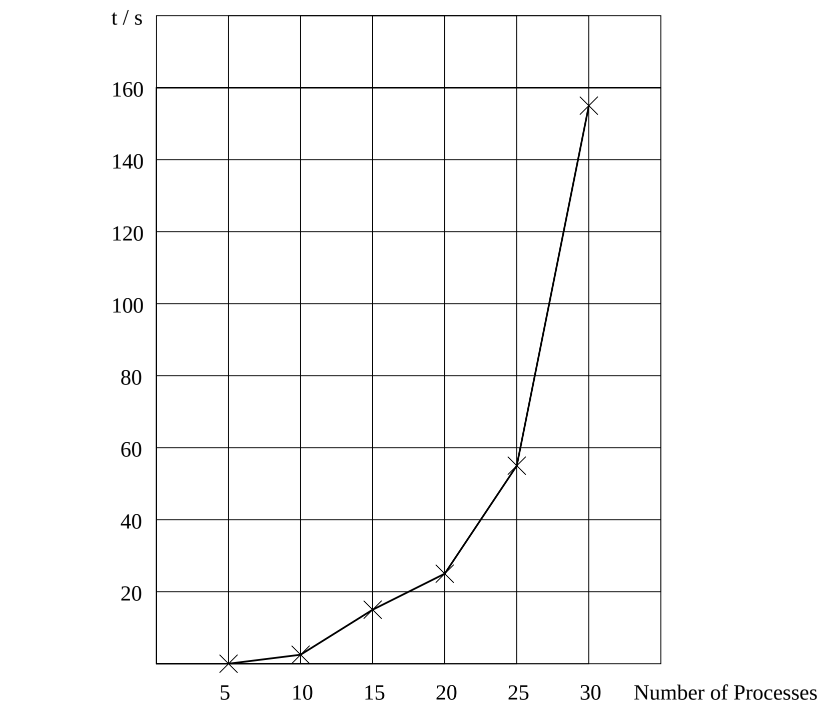


Figure A.2: Arbiter: UDS Generation Time

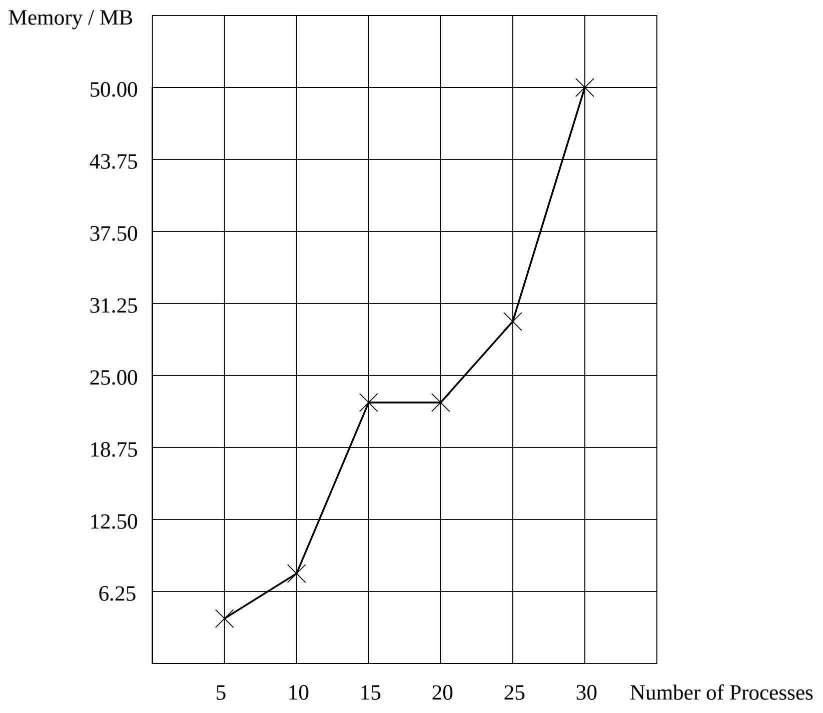


Figure A.3: Arbiter: UDS Generation Memory

A.1.2 Low-Level TKS Generation - Runtime Analysis

In order to use an arbiter within a bus system, certain specifications for transmission speed and latency periods have to exist. These can be translated into formal specifications. Thus, a specification of the form

$$req_i \rightarrow \mathsf{AF}^{\leq 100ns}\ ack_i$$

can express a maximum latency on a request signal of 100 ns. However, to express an indication of physical times in a specification, the formal model must contain the exact architecture-related times. For this purpose, we generated low-level TKSs by means of exact runtime analysis for the appropriate architecture (cf. section 6.2). We have performed this for three different architectures:

- Pentium 3, 1GHz, 1GB
- Ultrasparc III, 750MHz, 512MB
- Pentium 4, 2GHz, 512MB

The results for the obtained low-level TKSs and the runtime analysis are given in tables A.2, A.3 and A.4. The columns of the tables are as follows: The first column denotes the instantiation of the benchmark's parameter (number of processes). The second column shows how many Boolean variables were necessary to encode the state transition diagram and the physical times on the transitions. This means that the system has 2^{s_var} reachable states and the longest transition of the system is not exceeding $2^{t\_var} \times 10^{-6}$ seconds. Column three shows the determined runtimes for the runtime analysis and the low-level TKS generation, which are executed parallel. Columns four and five show memory consumption, expressed in required BDD nodes and kBytes of memory respectively. The required BDD nodes are given for both, the generated TKS and the obtained UDS (the TKS with untimed transitions), which is then used by the algorithms for all qualitative computations of the JCTL-operators. The last column finally, shows the determined runtimes for the minimal and maximal macro steps (in seconds $\times 10^{-6}$) on the target machines.

Each transition within the system holds exactly the time, which is needed for its own execution. As described in section 6.2.1, the size of generated code is linear to the size of the original synchronous program.

Note that the minimum and maximum execution times here are identical, since this benchmark contains no If-Then-Else-statements. This reduces significantly the runtimes and memory consumption and allows the generation of very large systems, like the Ultrasparc application for 30 processes, which contains 154 states– and 13 time variables and requires only approx. 1.5 min of runtime and 7.3 MBytes of main memory.

Arbiter: Low-Level TKS Generation - Runtime Analysis								
Number of	Variables		Time	BDD nodes		Memory	Transition [s]$\cdot 10^{-6}$	
Processes	states	time	h:m:s	UDS	TKS	kB	min	max
5	29	6	4.73	1482	1489	4786	50	50
10	54	8	7.29	5633	5642	5004	150	150
15	79	9	10.45	10959	10969	5265	367	367
20	104	10	29.88	23391	23402	6107	774	774
25	129	11	51.40	31474	31486	6250	1054	1054
30	154	11	1:19.32	44259	44271	7158	1197	1197
Pentium 3, 1GHz, 1GB								

Table A.2: Low-Level TKS Generation for Arbiter on Pentium 3 Architecture

Arbiter: Low-Level TKS Generation - Runtime Analysis								
Number of	Variables		Time	BDD nodes		Memory	Transition [s]$\cdot 10^{-6}$	
Processes	states	time	h:m:s	UDS	TKS	kB	min	max
5	29	5	3.13	1422	1428	4786	24	24
10	54	7	4.80	5702	5710	5004	67	67
15	79	7	6.98	11221	11229	5265	116	116
20	104	9	23.00	23391	23401	6107	370	370
25	129	10	39.08	31474	31485	6250	540	540
30	154	10	1:01.88	44259	44270	7158	613	613
Pentium 4, 2GHz, 512MB								

Table A.3: Low-Level TKS Generation for Arbiter on Pentium 4 Architecture

Arbiter: Low-Level TKS Generation - Runtime Analysis								
Number of	Variables		Time	BDD nodes		Memory	Transition [s]$\cdot 10^{-6}$	
Processes	states	time	h:m:s	UDS	TKS	kB	min	max
5	29	8	5.52	1409	1418	4786	143	143
10	54	9	8.38	5073	5083	5015	360	360
15	79	10	16.30	14041	14052	5625	596	596
20	104	11	31.20	23391	23403	6090	1831	1831
25	129	12	58.35	31208	31221	6200	3277	3277
30	154	13	1:23.43	44088	44102	7292	5820	5820
Ultrasparc III, 750MHz, 512MB								

Table A.4: Low-Level TKS Generation for Arbiter on Ultrasparc III Architecture

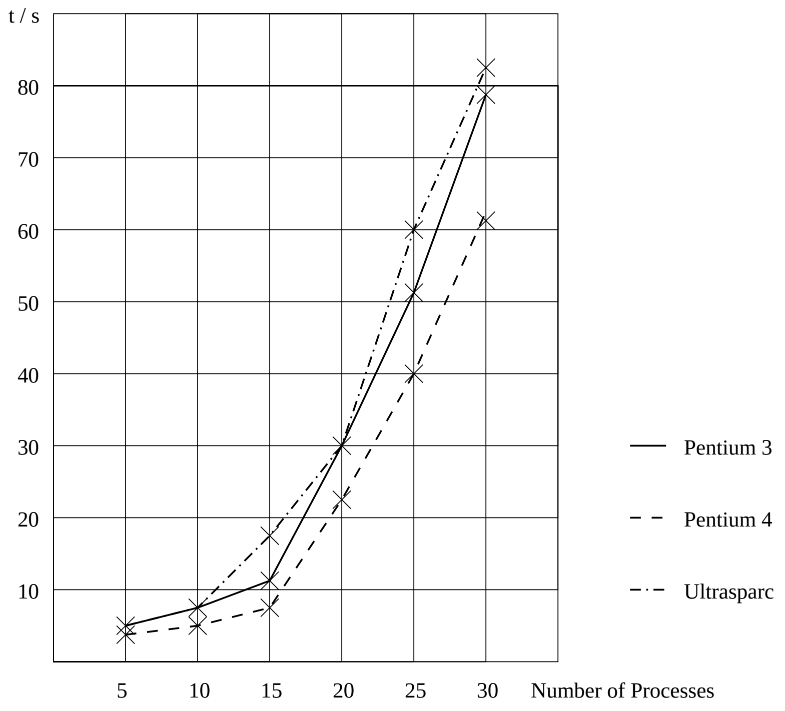


Figure A.4: Arbiter: Low-Level TKS Generation Time

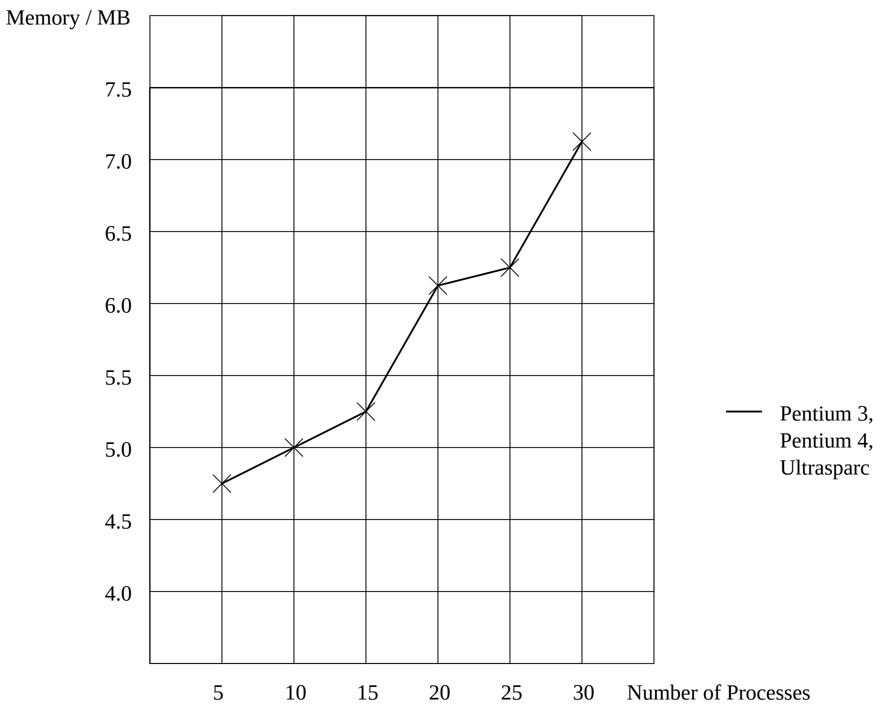


Figure A.5: Arbiter: Low-Level TKS Generation Memory

A.1.3 A Comparison to Other Tools

In order to compare JERRY′s performance for the qualitative specifi cations of the arbiter, we have tested the benchmark with SMV. As explained in section 8.1.7 we have chosen Cadence-SMV for our tests, as the best performing SMV version.

Arbiter: Verifi cation Comparison JERRY and SMV							
Number of Processes	Variables states	Time h:m:s		Memory kB		BDD nodes	
		JERRY	SMV	JERRY	SMV	JERRY	SMV
5	29	0.37	0.78	5001	3312	21462	13318
10	54	2.79	6.24	7150	3872	147168	27610
15	79	13.12	45.19	18087	8540	547792	239567
20	104	27.03	3:58.66	14427	45272	571298	2289214
25	129	58.24	3:26.93	28394	11132	1147706	224399
30	154	2:45.66	-	52722	>800000	2083858	-
Pentium 3, 1GHz, 1GB							

Table A.5: Verifi cation of Arbiter with JERRY and SMV

.

Arbiter: modular verifi cation with JERRY				
Number of Processes	Variables states	Time h:m:s	Memory kB	BDD nodes
5	29	0.23	4743	5110
10	54	1.60	4956	15330
15	79	4.50	5298	33726
20	104	6.08	5561	47012
25	129	15.05	6008	71540
30	154	29.56	6433	95046
Pentium 3, 1GHz, 1GB				

Table A.6: Arbiter Verifi cation with JERRY without Model Generation

The results of the direct comparison are listed in Table A.5. Furthermore, Table A.6 shows JERRY′s verifi cation results for the verifi cation stage only, i.e. when the formal model is already stored and must not be generated.

As can be seen, Cadence-SMV shows an instable behavior, specially for 20 processes (see also Fig. A.6 and A.7). As we started a run with 30 processes, the main memory consumption increased very fast up to the limit of our main memory, so we were forced to interrupt the run. Furthermore, as the curves of Fig. A.6 and A.7 show, JERRY′s behavior remain more flat and stable as the system's size grows and has much better runtimes than Cadence-SMV.

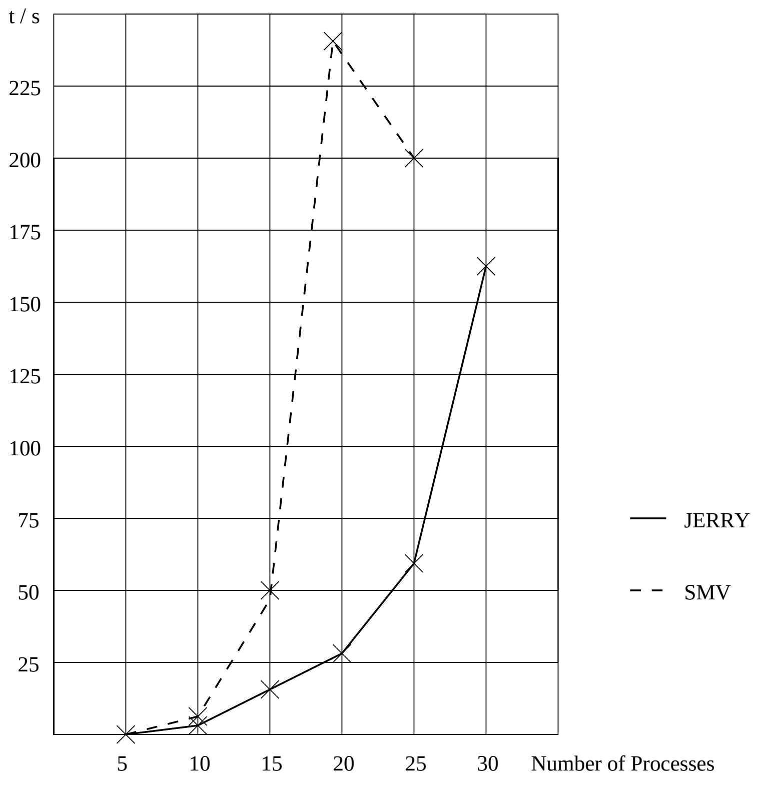


Figure A.6: Arbiter: Verifi cation Time

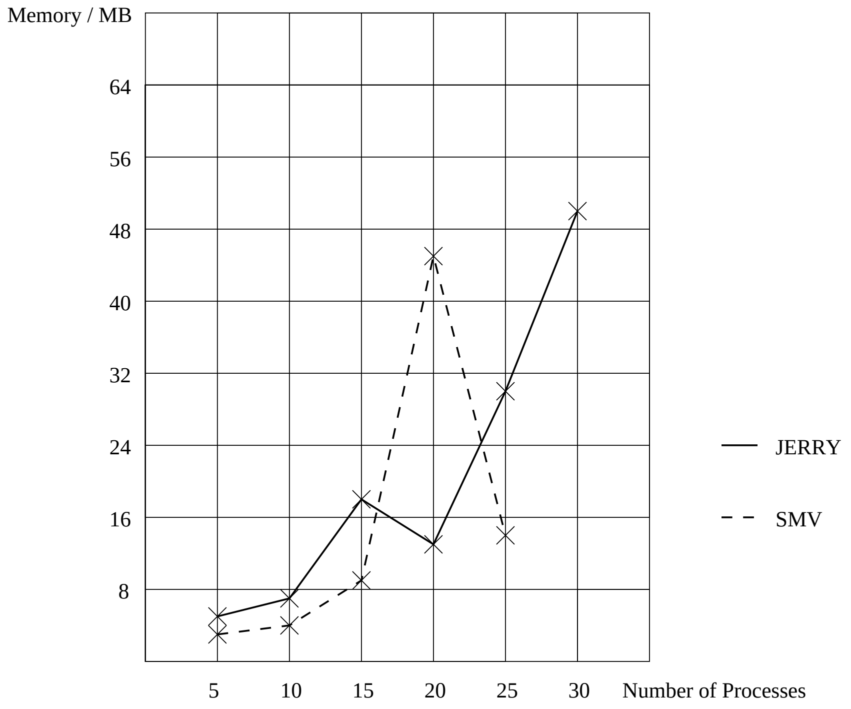


Figure A.7: Arbiter: Verifi cation Memory

Table A.6 finally, demonstrates again the benefits of JERRY's modular design: loading an already stored model lets us start directly the verification, overcoming the model generation. This improves the overall performance results significantly.

A.1.4 Low-Level TKS Verification

We have tested JERRY's performance for the low-level verification of the arbiter benchmark. Here, the original system is endowed by additional physical times, required for the code execution of the synchronous program on specific architectures (cf. section 6.2). This results an enormous increase of the system's complexity.

Again we want to mention, that we can not compare the low-level results to any other tool, since there exist no other tools capable of low-level verification. Nevertheless, it is a challenge for us to check JERRY with such high complex systems. According to the benchmark's function, we have proved the following specification:

$$\mathsf{EF}(arb \wedge tok_1 \wedge (\mathsf{AF}^{\leq\ max\_trans}\ tok_2))$$

This specification checks if there exists a path where the following holds: If the arbitration is permitted, then a token will be transmitted and this will happen within the maximum time required for a transition of the system (cf. Tables A.2, A.3 and A.4).

The verification results are shown in Tables A.7, A.8 and A.9. It's clearly to see that JERRY has a good, stable performance and can easily handle the system. Due to JERRY's modular design, we can again start the analysis directly at the low-level verification stage, overcoming repeated (and needless) model generations.

Arbiter: Low-Level TKS Verifi cation - Pentium 3					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
5	29	6	0.80	4834	8176
10	54	8	6.20	5713	58254
15	79	9	26.31	7344	150234
20	104	10	1:07.08	14655	337260
25	129	11	2:18.46	28580	671454
30	154	11	4:47.72	33543	958636
Pentium 3, 1GHz, 1GB					

Table A.7: Verifi cation of Quantitative Properties at Pentium 3 for Arbiter

Arbiter: Low-Level TKS Verifi cation - Pentium 4					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
5	29	5	0.78	4882	11242
10	54	7	4.50	5751	60298
15	79	7	20.71	7405	155344
20	104	9	58.33	14325	315798
25	129	10	2:09.01	27573	612178
30	154	10	3:30.27	20972	940240
Pentium 3, 1GHz, 1GB					

Table A.8: Verifi cation of Quantitative Properties at Pentium 4 for Arbiter

Arbiter: Low-Level TKS Verifi cation - Ultrasparc					
Number of	Variables		Time	Memory	BDD nodes
Processes	states	time	h:m:s	kB	
5	29	8	1.01	4914	13286
10	54	9	6.80	5580	49056
15	79	10	33.05	8090	196224
20	104	11	1:20.15	15263	376096
25	129	12	2:15.80	19876	643860
30	154	13	5:48:00	35474	1070034
Pentium 3, 1GHz, 1GB					

Table A.9: Verifi cation of Quantitative Properties at Ultrasparc III for Arbiter

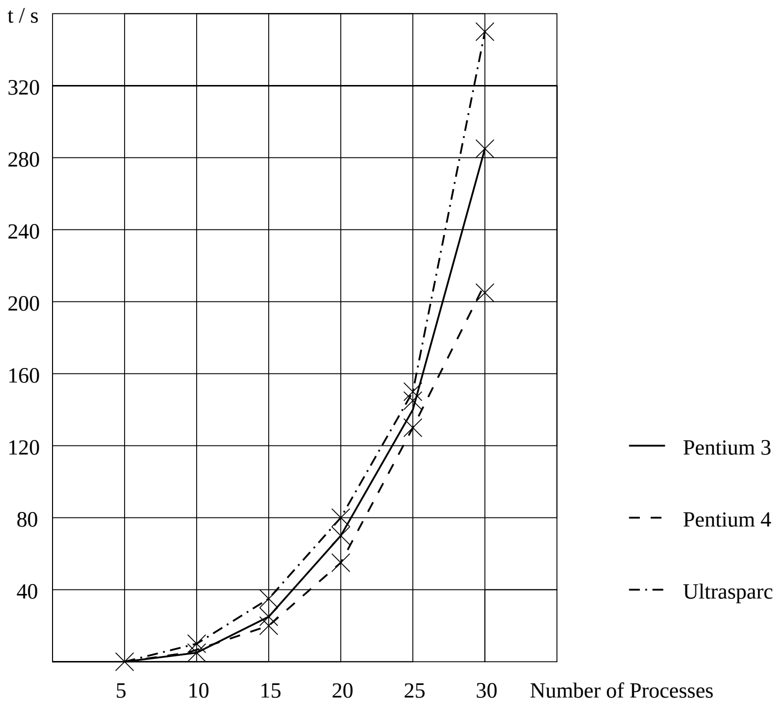


Figure A.8: Arbiter: Low-Level TKS Verifi cation Time

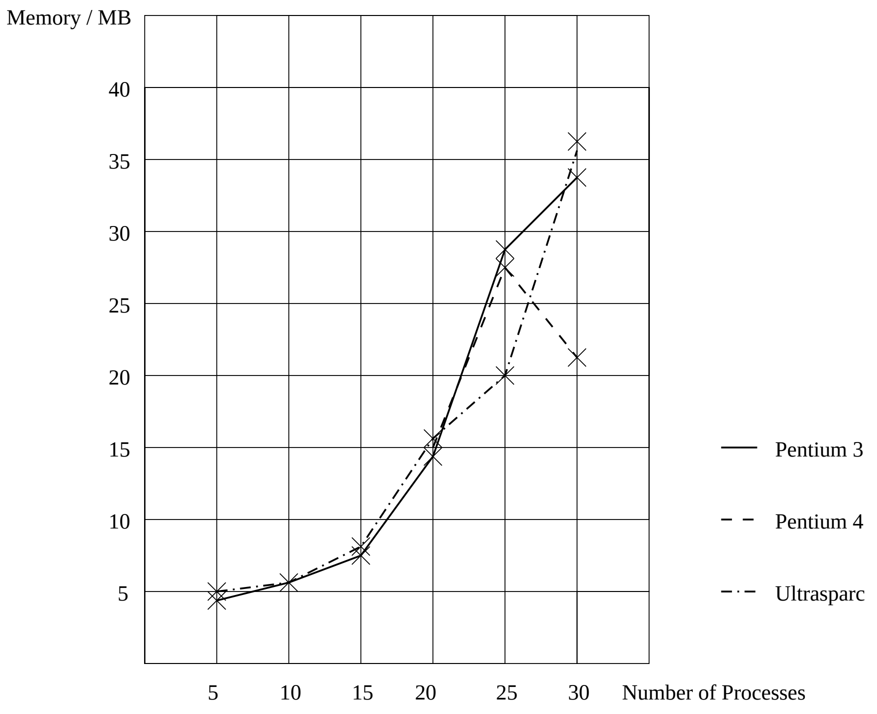


Figure A.9: Arbiter: Low-Level TKS Verifi cation Memory

A.2 Primality

Another benchmark that we have tested was a program for checking the primality of a given number. The idea is to first check if the number is even, and if not, to divide the number by all odd numbers starting from 3 up to the half of the number to be tested.

Fig. A.10 shows the algorithm in pseudo code. If n is even and $n \neq 2$ holds, then n is not a prime number ($n = 2$ is by definition a prime number). If n is odd, then n is divided by odd numbers x between 3 and $n/2$. Then the algorithm checks if the rest of the division is 0. If the algorithm detects a number x which results 0 as rest of the division, then the n is not prime - otherwise, it is.

```
Primality($n$)
if $even(x)$ then
  if $x = 2$ then
    return $prime$;
  end;
else
  forall $x >= 3 \wedge odd(x) \wedge x < n/2$ do
    if $n$ mod $x = 0$ then
      return $non\_prime$;
    end;
  end;
  return $prime$;
end;
```

Figure A.10: Algorithm for Checking the Primality of a Number n

In principle, this program is a transformational algorithm. Nevertheless, nowadays different encoding and encrypting algorithms exist, which are based on prime numbers. Thus, if we consider the real-time encryption of data in systems, this algorithm underlies certain real-time constraints. For the Primality benchmark, as well as for the following Euclid's GCD algorithm, is important to notice that, in order to focus on testing the tool's performance, we do not exploit the arithmetic instructions of the microprocessors. Instead, the arithmetic operations of a synchronous program are mapped to the Boolean level and are implemented as bit-operations on the microprocessors.

The program has been instantiated in Quartz for different bits and is given in Figure A.11. The algorithm's input is the number to be checked. As output, the program returns the signals $prime$ or non_prime. The divisions given by **div** 2 in the algorithm are performed by means of a binary shift operation. As division is hard to implement by a combinatorial circuit, we have chosen a sequential algorithm that requires b steps for a division of two b-bit numbers.

```
**module** Primality :
  **input** $p$ : **int**$[n]$;
  **output** $prime, non\_prime$;
  **local** *ls*$, x$ : **int**$[n-1]$ **in**
    $a := p$;
    **if** $a[0]$ **then**
      **weak abort**
        $x := 3$;
        **if** $x < (a$ **div** $2)$ **then emit** *ls* **end**;
        **while** *ls* **do**
          $d := a$;
          $D$ : **run** *Division*$()(d, x, m)$;
          **if** $m = 0$ **then emit** $non\_prime$ **end**;
          $\ell_1$ : **pause**;
          **next**$(x) := x + 2$;
          $\ell_2$ : **pause**;
          **if** $x < (d$ **div** $2)$ **then emit** *ls* **end**
        **end**
      **when** $non\_prime$;
      **if** $non\_prime$ **then emit** $prime$ **end**
    **else**
      **if** $a = 2$ **then emit** $prime$
      **else emit** $non\_prime$ **end**;
    **end**;
    $tc$ : **pause**
  **end**
**end**
```

Figure A.11: Algorithm to Test Primality

A.2.1 UDS Generation

We start again by first generating the unit-delay structure for the Primality algorithm. Table A.10 shows the results for the UDS generation for n bits. All experiments were run on a Pentium 3 with 1GHz and 1GByte of main memory.

The columns of the table are as follows: The first column denotes the instantiation of the benchmark's parameter (bits). The second column shows how many Boolean variables were necessary to encode the state transition diagram, which means that the system has 2^{var} reachable states. Column three shows the determined runtimes for the UDS generation. Columns four and five show memory consumption, expressed in required BDD nodes and kBytes of memory respectively.

Primality: UDS generation				
Bitwidth	Variables states	Time h:m:s	BDD nodes	Memory kB
4	44	0.85	1267	5036
5	55	2.25	6705	6059
6	66	11.13	9280	8992
7	77	47.11	53516	27845
8	88	3:43.25	86848	35150
Pentium 3, 1GHz, 1GB				

Table A.10: UDS Generation for Primality Algorithm

Note that the runtimes include also the time needed for exporting and storing the formal model and all other needed data on hard disk.

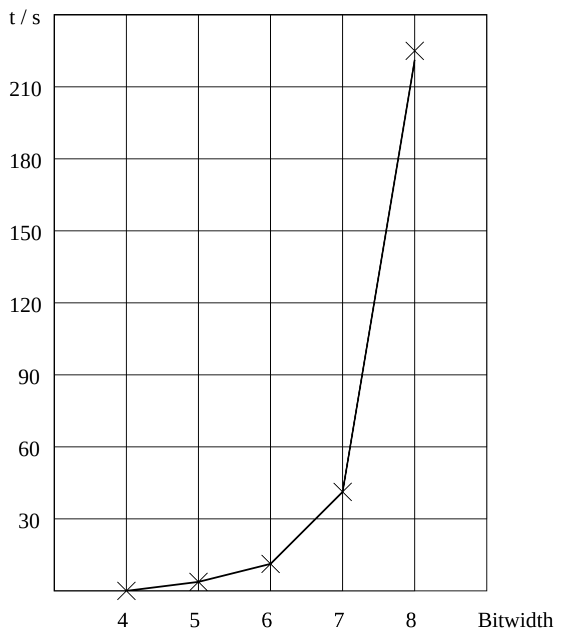


Figure A.12: Primality: UDS Generation Time

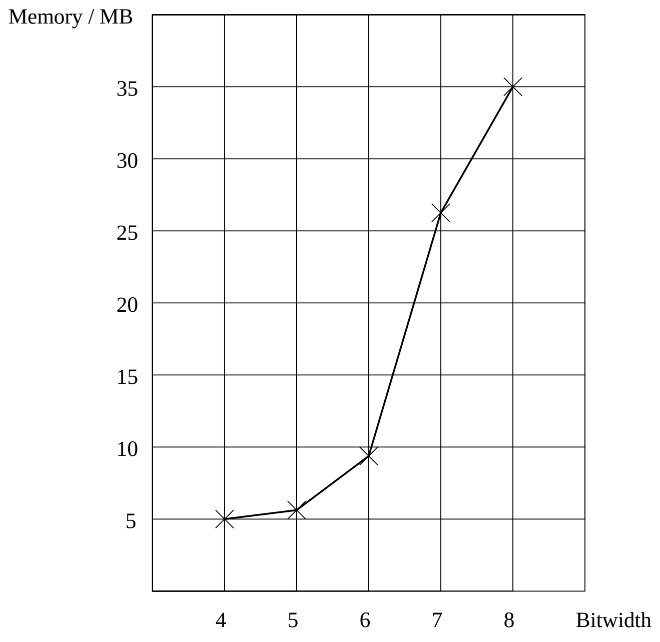


Figure A.13: Primality: UDS Generation Memory

A.2.2 High-Level TKS Generation - Chronos

The generation of a high-level TKS for the primality benchmark uses the algorithm Chronos, explained in section 6.1.2. The benchmark is well-suited to perform abstractions. We have abstract away from the states that regard the internal operations of the division. In particular, we have used the following abstraction (cf. Fig. A.11):

abstract
  D : **run** *Division*$()(d, x, m)$;
end;

Table A.11 shows the obtained results. All experiments were run on a Pentium 3 with 1GHz and 1GByte of main memory.

The columns of the table are as follows: The fi rst column denotes the instantiation of the benchmark's parameter (bits). The second column shows how many Boolean variables were necessary to encode the state transition diagram and the logical times on the transitions. This means that the system has 2^{s_var} reachable states and the longest transition of the system is not exceeding 2^{t_var} logical time units. Column three shows the determined runtimes for the high-level TKS generation. Columns four and fi ve show memory consumption, expressed in required BDD nodes and kBytes of memory respectively. The required BDD nodes are given for both, the generated TKS and the obtained UDS (the TKS with untimed transitions), which is then used by the algorithms for all qualitative computations of the JCTL-operators. Column six fi nally, shows the minimum and maximum duration (in logical time units) of the transitions contained in the high-level TKS.

Primality: High-Level TKS Generation - Chronos								
Bitwidth	Variables		Time	BDD nodes		Memory	Transition	
	states	time	h:m:s	UDS	TKS	kB	min	max
4	44	3	0.67	667	691	4956	1	4
5	55	3	3.68	1791	1796	6063	1	5
6	66	3	8.62	5292	5427	10145	1	6
7	77	3	45.61	13639	13655	12856	1	7
8	88	4	2:01.43	46490	46505	32839	1	8
Pentium 3, 1GHz, 1GB								

Table A.11: High-Level TKS Generation for Primality Algorithm

Again we see also for this benchmark, that the size in BDD nodes of the abstract structures is signifi cantly smaller than the size of the fi rst generated UDS (cf. Table A.10).

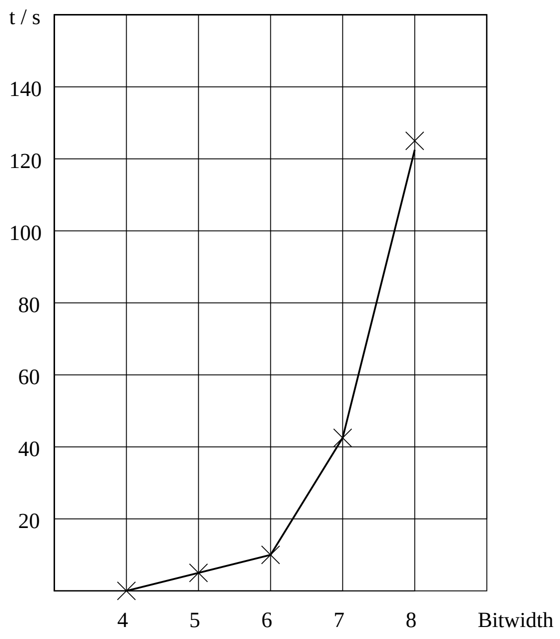


Figure A.14: Primality: High-Level TKS Generation Time

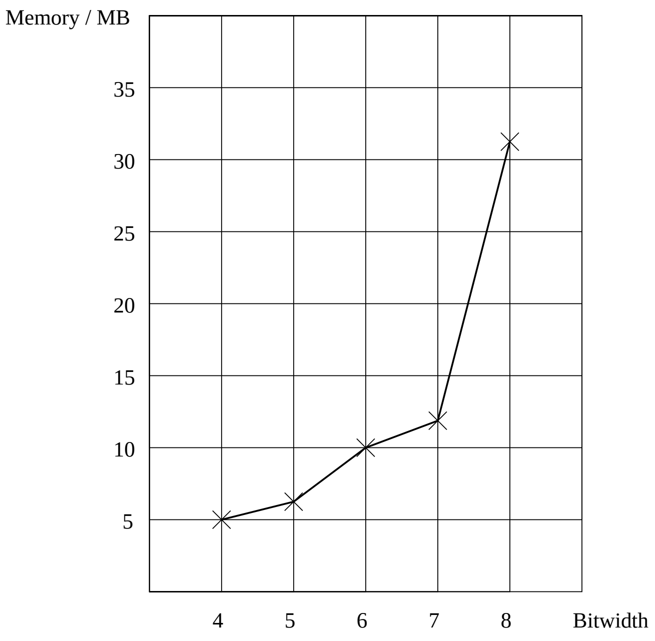


Figure A.15: Primality: High-Level TKS Generation Memory

A.2.3 Low-Level TKS Generation - Runtime Analysis

Similar to the previous benchmarks, we have performed an exact runtime analysis for the primality algorithm, in order to generate low-level TKSs for low-level verifi cation purposes (cf. section 6.2). We have performed this for three different architectures:

- Pentium 3, 1GHz, 1GB
- Ultrasparc III, 750MHz, 512MB
- Pentium 4, 2GHz, 512MB

The results for the obtained low-level TKSs and the runtime analysis are given in Tables A.12, A.13 and A.14.

The columns of the tables are as follows: The fi rst column denotes the instantiation of the benchmark's parameter (number of processes). The second column shows how many Boolean variables were necessary to encode the state transition diagram and the physical times on the transitions. This means that the system has 2^{s_var} reachable states and the longest transition of the system is not exceeding $2^{t\_var} \times 10^{-6}$ seconds. Column three shows the determined runtimes for the runtime analysis and the low-level TKS generation, which are executed parallel. Columns four and fi ve show memory consumption, expressed in required BDD nodes and kBytes of memory respectively. The required BDD nodes are given for both, the generated TKS and the obtained UDS (the TKS with untimed transitions), which is then used by the algorithms for all qualitative computations of the JCTL-operators. The last column fi nally, shows the determined runtimes for the minimal and maximal macro steps (in seconds $\times 10^{-6}$) on the target machines.

Note that each transition within the system holds exactly the time, which is needed for its own execution and this time lies always between the minimal and the maximal value. As can be seen, JERRY has again no diffi culties in generating a system that contains 88 state– and 11 time variables. Due to the different machine architectures (e.g. use of different pipelines, caches, etc.), the performance of the generated code may show some irregularities, like the 5-bit variant on Pentium 4 (cf. Table A.13). This demonstrates again clearly the necessity of an exact runtime analysis, since an estimation of runtime, based on a few example-data only, can be very misleading.

Primality: Low-Level TKS Generation - Runtime Analysis								
Bitwidth	Variables		Time	BDD nodes		Memory	Transition [s]$\cdot 10^{-6}$	
	states	time	h:m:s	UDS	TKS	kB	min	max
4	44	7	6.12	918	1153	4848	65	118
5	55	8	7.92	3045	3527	5063	92	129
6	66	8	15.51	9363	9928	5670	140	198
7	77	9	30.50	28407	27929	6680	305	425
8	88	10	1:33.36	86479	90163	8926	455	604
Pentium 3, 1GHz, 1GB								

Table A.12: Low-Level TKS Generation for Primality Algorithm on Pentium 3 Architecture

Primality: Low-Level TKS Generation - Runtime Analysis								
Bitwidth	Variables		Time	BDD nodes		Memory	Transition [s]$\cdot 10^{-6}$	
	states	time	h:m:s	UDS	TKS	kB	min	max
4	44	6	3.93	932	1151	4844	33	43
5	55	7	5.04	3046	3348	5063	46	121
6	66	7	11.25	9363	9945	5670	72	86
7	77	7	23.50	25526	25703	6461	93	112
8	88	8	1:34.39	86479	90084	8890	125	151
Pentium 4, 2GHz, 512MB								

Table A.13: Low-Level TKS Generation for Primality Algorithm on Pentium 4 Architecture

Primality: Low-Level TKS Generation - Runtime Analysis								
Bitwidth	Variables		Time	BDD nodes		Memory	Transition [s]$\cdot 10^{-6}$	
	states	time	h:m:s	UDS	TKS	kB	min	max
4	44	9	8.56	926	1282	4871	175	257
5	55	9	11.72	3047	3700	5063	259	374
6	66	10	21.73	9424	10868	5654	392	532
7	77	10	38.78	25982	26087	6616	565	725
8	88	11	1:30.70	86176	89985	8921	829	1068
Ultrasparc III, 750MHz, 512MB								

Table A.14: Low-Level TKS Generation for Primality Algorithm on Ultrasparc III Architecture

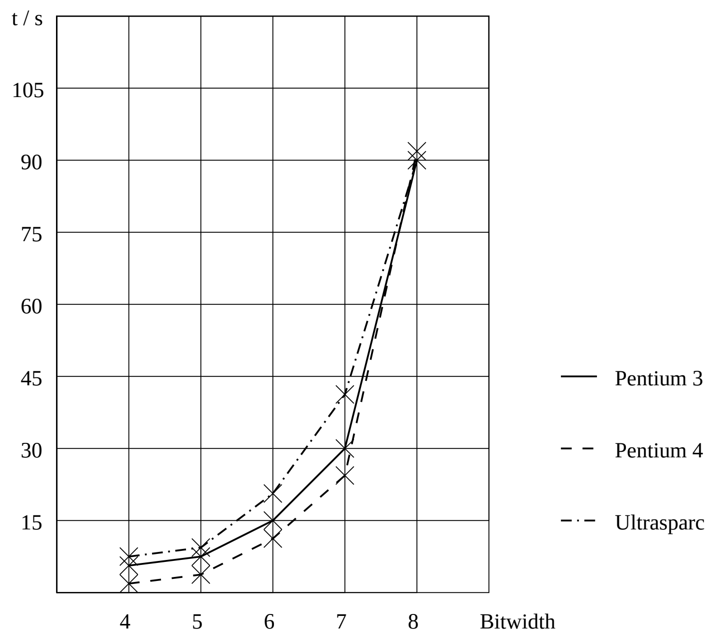


Figure A.16: Primality: Low-Level TKS Generation Time

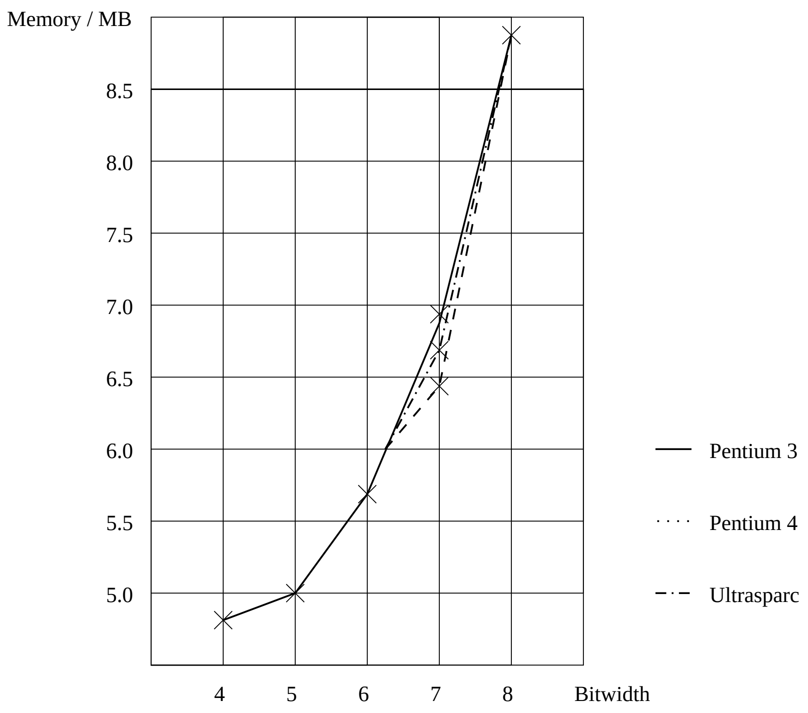


Figure A.17: Primality: Low-Level TKS Generation Memory

A.2.4 WCET Analysis

The WCET analysis of the Primality benchmark gives very interesting results. A rough estimation of the WCET would be as follows: as the largest input number for n bits is $2^n - 1$, we have to calculate no more than $2^{n-2} - 1$ divisions, since we divide only by odd numbers up to $2^{n-1} - 1$. The division with n bits will take n steps, and there are two further macro steps (the pause statements labeled with ℓ_1 and ℓ_2) in the loop body, and one after the loop (the pause statements labeled with *tc*). Hence, an analytical estimation for the WCET is $(n + 2)(2^{n-2} - 1) + 1$. Table A.15 shows some results that we have obtained for $n \in \{4, 5, 6, 7, 8\}$.

Primality: WCET and BCET analysis - EHLA							
Bitwidth	Variables states	Time h:m:s	Memory kB	BDD nodes	BCET	WCET	# Div.
4	44	0.94	5606	54166	1	13	2
5	55	5.18	6615	112420	1	43	6
6	66	12.48	14305	331128	1	113	14
7	77	1:00.64	31454	860524	1	271	30
8	88	3:47.62	56832	1467592	1	611	61
Pentium 3, 1GHz, 1GB							

Table A.15: WCET and BCET Analysis for Primality Algorithm

The columns of the table are as follows: The first column denotes the instantiation of the benchmark's parameter (bits). The second column shows how many Boolean variables were necessary to encode the state transition diagram, which means that the system has 2^{var} reachable states. Column three shows the determined runtimes for the WCET analysis. Columns four and five show memory consumption, expressed in kBytes of memory and required BDD nodes respectively. Columns six and seven show the best– and worst execution times respectively. Column eight finally, shows the maximum number of executed divisions, i.e. loop iterations. (cf. section 7.1.1).

As $2^{n-2} - 1$ is 15, 31, and 63 (for 6,7, and 8 bits, respectively), we see that our analytical estimation was too pessimistic. For the WCET, the analytical prediction would yield the numbers 121, 280, and 631 (for 6,7, and 8 bits, respectively) which is also too pessimistic. We see, that our exact bounds are better than the pessimistic analytical estimation.

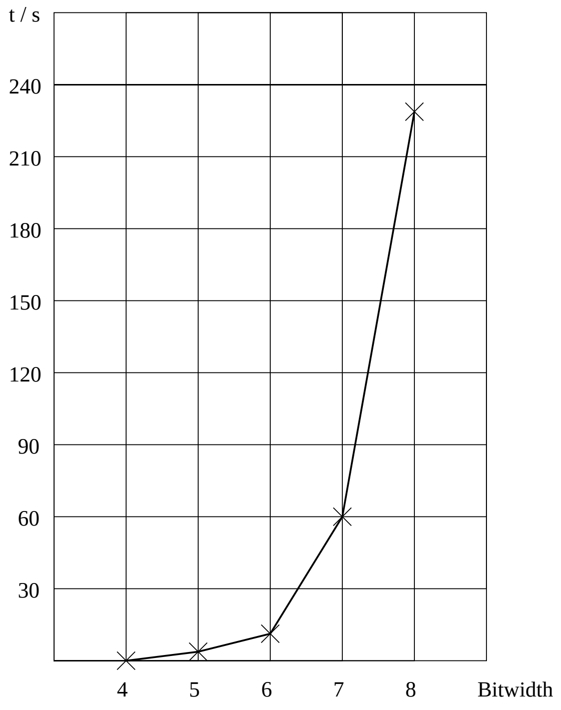


Figure A.18: Primality: EHLA Time

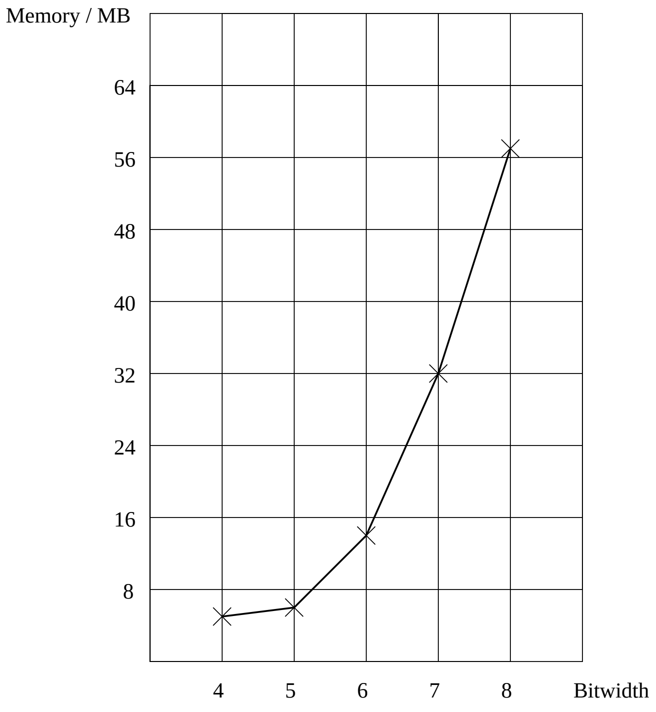


Figure A.19: Primality: EHLA Memory

A.2.5 High-Level TKS Verification

In order to perform high-level verifi cation for the primality benchmark, we have tested the operator $\mathsf{AF}^{\kappa}\varphi$, according to the results of WCET and BCET analysis. In particular, we have proved the correctness of the WCET and BCET analysis by checking the following real-time specifi cation:

$$\mathsf{AF}^{\leq\ WCET}(test\_complete)$$

where $test\_complete$ simply means that the primality test is completed. The verifi cation results are shown in Table A.16.

Primality: High-Level TKS Verifi cation					
Bitwidth	Variables		Time	Memory	BDD nodes
	states	time	h:m:s	kB	
4	44	3	0.77	4892	9198
5	55	3	2.61	5036	16352
6	66	3	9.33	5986	73584
7	77	3	26.91	13773	288204
8	88	4	3:16.33	30377	823732
Pentium 3, 1GHz, 1GB					

Table A.16: Verifi cation of Quantitative Properties at High-Level for Primality Algorithm

The columns of the table are as follows: The fi rst column denotes the instantiation of the benchmark's parameter (bits). The second column shows how many Boolean variables were necessary to encode the state transition diagram and the logical times on the transitions. This means that the system has 2^{s_var} reachable states and the longest transition of the system is not exceeding 2^{t_var} logical time units. Column three shows the determined runtimes for the high-level TKS verifi cation. Columns four and fi ve show memory consumption, expressed in kBytes of memory and required BDD nodes respectively. The verifi cation results of Table A.16 again show that JERRY has no diffi culties in verifying the primality benchmark.

A.2.6 Low-Level TKS Verification

We have tested JERRY's performance for the low-level verifi cation of the primality benchmark. Here, the original system is endowed by additional physical times, required for the code execution of the synchronous program on specifi c architectures (cf. section 6.2). This results an enormous increase of the system's complexity, specially for the Ultrasparc III machine, where the code was executed slower (cf. Table A.14) and hence, the time variables are more. Nevertheless, it is a challenge for us to check JERRY with such a high complex system.

Similar to the high-level verifi cation, we tested the system according to the results of WCET analysis. In particular, we have proved the correctness of the WCET analysis by checking the following real-time specifi cation:

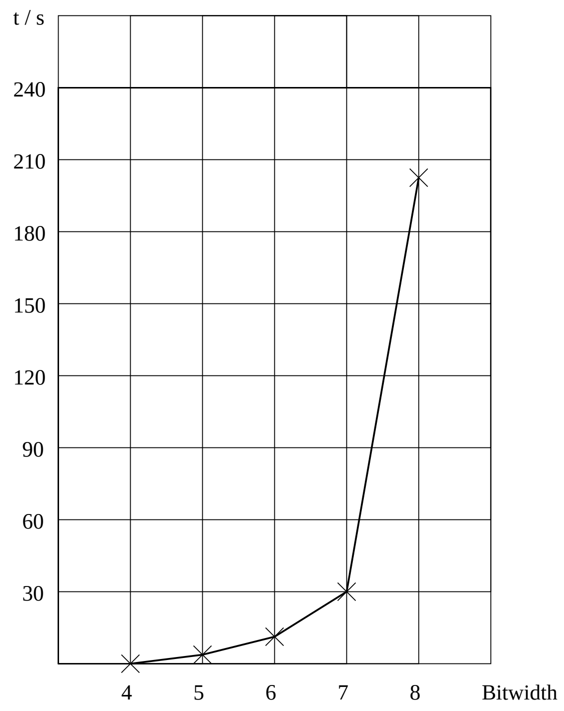


Figure A.20: Primality: High-Level TKS Verifi cation Time

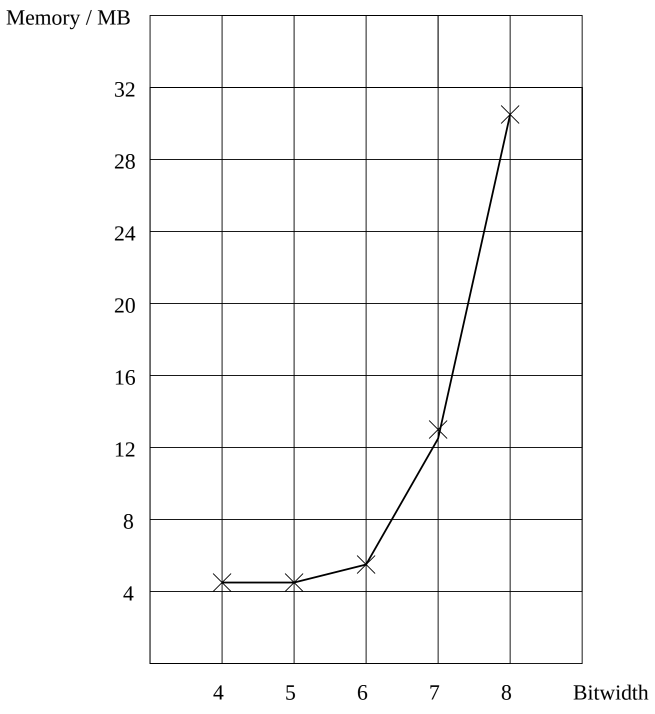


Figure A.21: Primality: High-Level TKS Verifi cation Memory

$$\mathsf{AF}^{\leq\ (WCET\times\ max\_trans)}(test\_complete)$$

where $test\_complete$ simply means that the primality test is completed and max_trans is the maximum transition included in the system, as can be obtained from the Tables A.12, A.13 and A.14. The verification results are shown in Tables A.17, A.18 and A.19.

Primality: Low-Level TKS Verification - Pentium 3					
Bitwidth	Variables		Time	Memory	BDD nodes
	states	time	h:m:s	kB	
4	44	7	2.56	6964	134904
5	55	8	11.62	7952	191114
6	66	8	1:42.93	12281	451724
7	77	9	1:41:53.78	58749	1513582
8	88	10	12:20:02.30	121160	5460546
Pentium 3, 1GHz, 1GB					

Table A.17: Verification of Quantitative Properties at Pentium 3 for Primality Algorithm

Primality: Low-Level TKS Verification - Pentium 4					
Bitwidth	Variables		Time	Memory	BDD nodes
	states	time	h:m:s	kB	
4	44	6	1.50	6459	105266
5	55	7	11.05	7994	191114
6	66	7	1:42.98	11027	370986
7	77	7	14:21.50	40093	1367436
8	88	8	4:12:56.52	87995	3195794
Pentium 3, 1GHz, 1GB					

Table A.18: Verification of Quantitative Properties at Pentium 4 for Primality Algorithm

Primality: Low-Level TKS Verification - Ultrasparc					
Bitwidth	Variables		Time	Memory	BDD nodes
	states	time	h:m:s	kB	
4	44	9	5.17	5204	26572
5	55	9	56.25	7357	154322
6	66	10	2:01.32	11453	394492
7	77	10	2:50:36.98	64399	1833468
8	88	11	>15h		
Pentium 3, 1GHz, 1GB					

Table A.19: Verification of Quantitative Properties at Ultrasparc III for Primality Algorithm

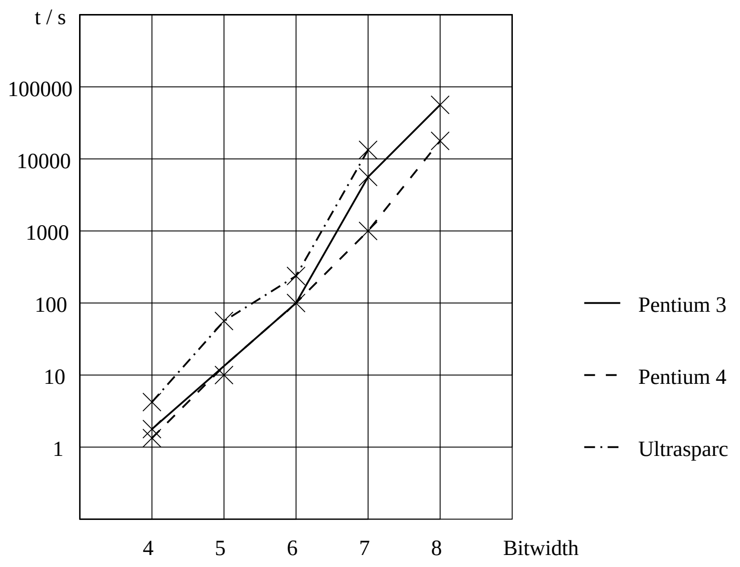


Figure A.22: Primality: Low-Level TKS Verification Time

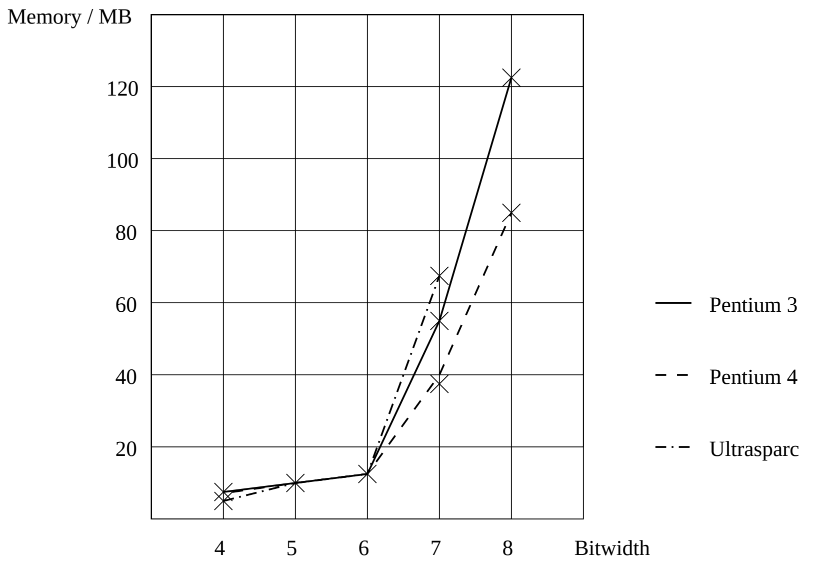


Figure A.23: Primality: Low-Level TKS Verification Memory

This example demonstrates the high complexity of a division operation, implemented by means of BDDs, but also JERRY's stable memory consumption. It's clearly to see that although the BBD nodes and the runtimes increase signifi cantly with the number of bits (approx. 5.5 million nodes for Pentium 3), JERRY shows a stable, low main memory consumption, which made it possible to handle the system, even after computing for a few hours. Due to JERRY's modular design, we can again start the analysis directly at the low-level verifi cation stage, overcoming repeated (and needless) model generations.

A.3 Euclid's Greatest Common Divisor Algorithm

As last benchmark we have analyzed Euclid's algorithm to compute the greatest common divisor of two given numbers. Figure A.24 shows a generic form of the algorithm for numbers represented by n bits. The subtractions were implemented by means of a combinational module.

```
module Euclid :
  input $a$ : int[$n$], $b$ : int[$n$];
  output $x$ : int[$n$], $y$ : int[$n$];
  $x := a$;
  $y := b$;
  do
    if $x \geq y$ then
      next($x$) := $x - y$
    else
      next($y$) := $y - x$
    end;
    $\ell$ : pause
  while $(x \neq 0) \wedge (y \neq 0)$;
  if $x = 0$ then next($x$) := $y$ end;
  $rdy$ : pause
  /* $x$ is the gcd of $a$ and $b$ */
end
```

Figure A.24: Euclid's GCD Algorithm

The two numbers are given as a and b. If a and b have a common divisor, then both numbers can be considered as $a := p \cdot q$ and $b := p \cdot q$. Upon the fact, that the greatest common divisor of two numbers is positive, a case distinction has to be made for the following subtractions:

If $a > b$, then the value b is subtracted from a. Thus, we get a positive number, which has the same greatest common divisor as a as well as b, since $a - b = (p - p') \cdot q$.

If $b < a$, then the value a is subtracted from b.

The algorithm terminates after a finite number of steps due to the fact, that at least the number 1 is the greatest common divisor of two numbers and consequently the termination-condition $a = 0$ or $b = 0$ are fulfilled. The greatest common divisor is returned after the loop termination from the remaining value, which is not 0.

A.3.1 UDS Generation

We have instantiated the Quartz program given in Figure A.24 with various numbers for the bitwidth of the numbers n, and generated first unit-delay structures for the systems. Table A.20 shows the obtained results. All experiments were run on a Pentium 3 with 1GHz and 1GByte of main memory.

The columns of the table are as follows: The first column denotes the instantiation of the benchmark's parameter (bits). The second column shows how many Boolean variables were necessary to encode the state transition diagram, which means that the system has 2^{var} reachable states. Column three shows the determined runtimes for the UDS generation. Columns four and five show memory consumption, expressed in required BDD nodes and kBytes of memory respectively.

Note that the runtimes include also the time needed for exporting and storing the formal model and all other needed data on hard disk.

Euclid: UDS generation				
Bitwidth	Variables states	Time h:m:s	BDD nodes	Memory kB
4	23	0.23	1631	4966
5	28	0.56	2004	5020
6	33	0.87	2164	5359
7	38	2.33	20405	5755
8	43	1.67	7884	5824
9	48	2.92	24988	6676
10	53	4.06	19067	6897
11	58	5.97	53557	8612
12	63	10.07	61189	8835
Pentium 3, 1GHz, 1GB				

Table A.20: UDS Generation for Euclid's GCD Algorithm

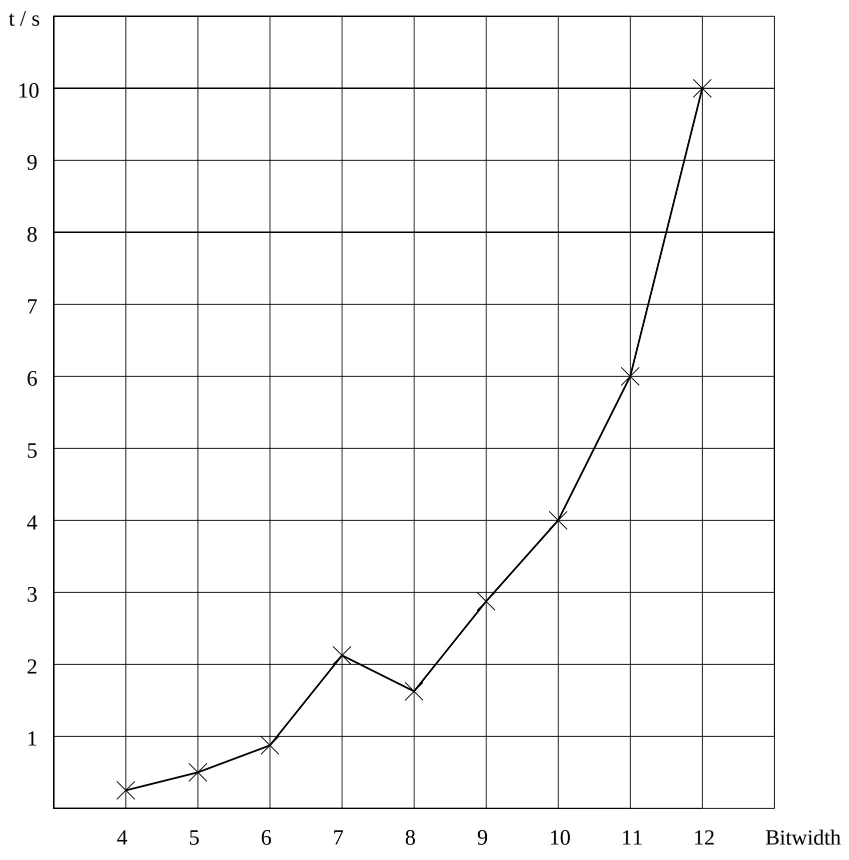


Figure A.25: Euclid: UDS Generation Time

A.3.2 High-Level TKS Generation - Chronos

The generation of a high-level TKS for Euclid's benchmark uses the algorithm Chronos, explained in section 6.1.2. The benchmark is well-suited to perform abstractions. We have abstract away from the states that regard the internal **do - while** - loop operations of the algorithm shown in Table A.24.

Table A.21 shows the obtained results. All experiments were run on a Pentium 3 with 1GHz and 1GByte of main memory.

The columns of the table are as follows: The first column denotes the instantiation of the benchmark's parameter (bits). The second column shows how many Boolean variables were necessary to encode the state transition diagram and the logical times on the transitions. This means that the system has 2^{s_var} reachable states and the longest transition of the system is not exceeding 2^{t_var} logical time units. Column three shows the determined runtimes for the high-level TKS generation. Columns four and five show memory consumption, expressed in required BDD nodes and kBytes of memory respectively. The required BDD nodes are given

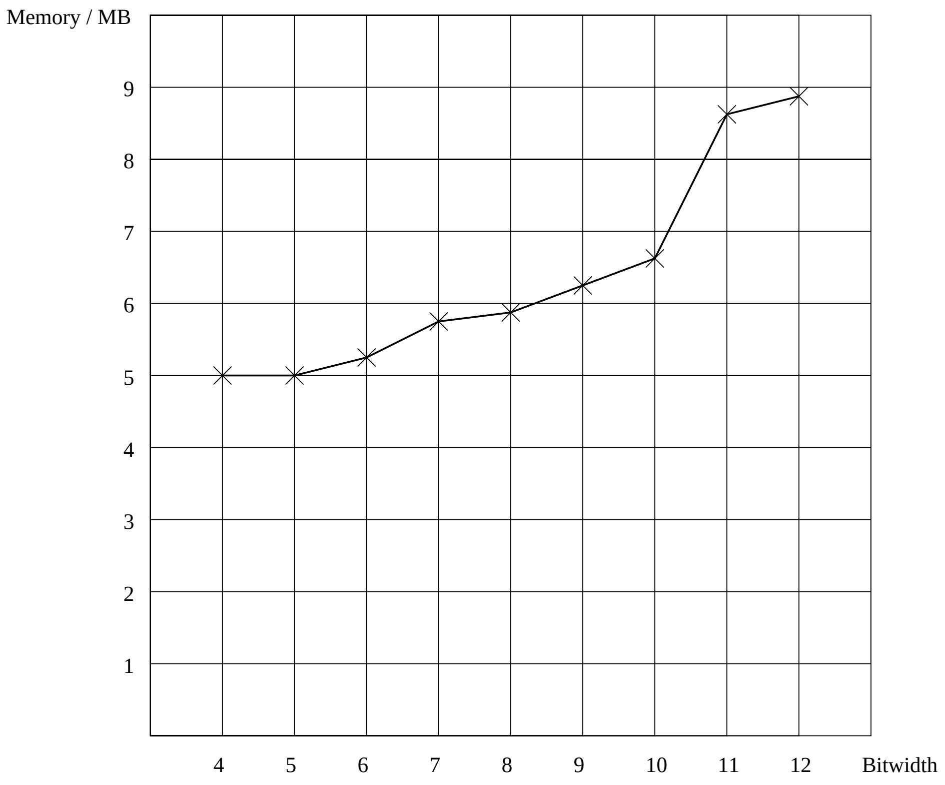


Figure A.26: Euclid: UDS Generation Memory

for both, the generated TKS and the obtained UDS (the TKS with untimed transitions), which is then used by the algorithms for all qualitative computations of the JCTL-operators. Column six fi nally, shows the minimum and maximum duration (in logical time units) of the transitions contained in the high-level TKS.

Euclid: High-Level TKS Generation - Chronos								
Bitwidth	Variables		Time	BDD nodes		Memory	Transition	
	states	time	h:m:s	UDS	TKS	kB	min	max
4	23	4	0.56	690	1041	5104	1	15
5	28	5	1.40	1736	4250	6450	1	31
6	33	6	5.46	7168	22385	8735	1	63
7	38	7	35.41	19399	46259	20429	1	127
8	43	8	59.39	34485	135341	28387	1	255
9	48	9	12:50.07	124482	349343	58014	1	511
10	53	10	1:06:24.57	285073	1136793	79346	1	1023
Pentium 3, 1GHz, 1GB								

Table A.21: High-Level TKS Generation for Euclid's GCD Algorithm

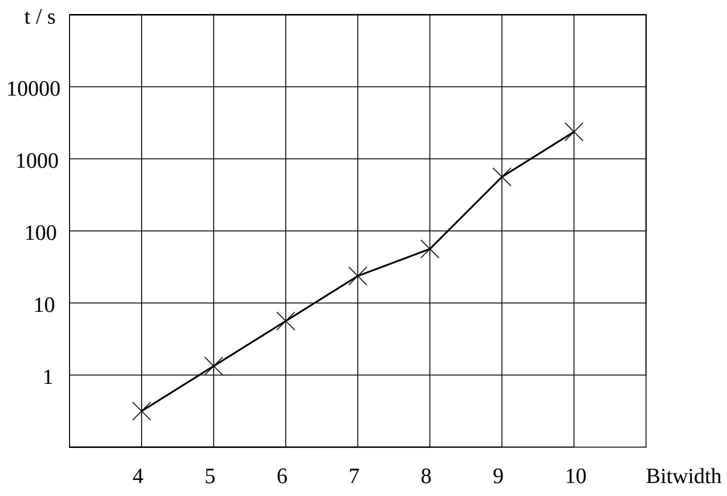


Figure A.27: Euclid: High-Level TKS Generation Time

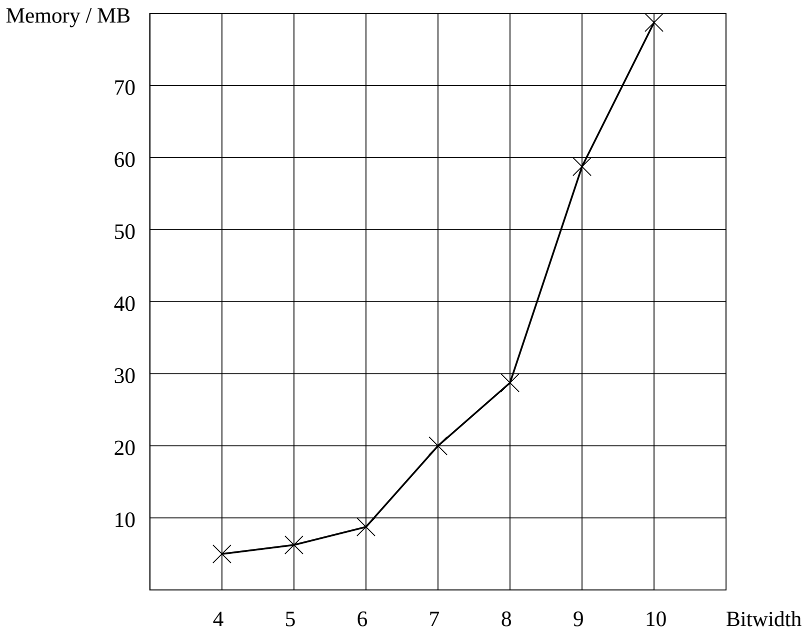


Figure A.28: Euclid: High-Level TKS Generation Memory

A.3.3 Low-Level TKS Generation - Runtime Analysis

As in the previous examples, we have performed an exact runtime analysis in order to generate low-level TKSs for the Euclid's algorithm, according to the techniques presented in section 6.2. In particular, we have considered again three different architectures:

- Pentium 3, 1GHz, 1GB
- Ultrasparc III, 750MHz, 512MB
- Pentium 4, 2GHz, 512MB

The results for the obtained TKSs and the runtime analysis are given in tables A.22, A.23 and A.24.

The columns of the tables are as follows: The first column denotes the instantiation of the benchmark's parameter (bits). The second column shows how many Boolean variables were necessary to encode the state transition diagram and the physical times on the transitions. This means that the system has 2^{s_var} reachable states and the longest transition of the system is not exceeding $2^{t\_var} \times 10^{-5}$ seconds. Column three shows the determined runtimes for the runtime analysis and the low-level TKS generation, which are executed parallel. Columns four and five show memory consumption, expressed in required BDD nodes and kBytes of memory respectively. The required BDD nodes are given for both, the generated TKS and the obtained UDS (the TKS with untimed transitions), which is then used by the algorithms for all qualitative computations of the JCTL-operators. The last column finally, shows the determined runtimes for the minimal and maximal macro steps (in seconds $\times 10^{-5}$) on the target machines. Note that each transition within the system holds exactly the time, which is needed for its own execution and this time lies always between the minimal and the maximal value.

It is somehow surprisingly, that the executable code shows the best performance on the Ultrasparc III machine (only 750 MHz). In particular, the longest transition for 8 bit on a Pentium 3 is approx. 7 times slower and on a Pentium 4 is approx. 5 times slower. This results to a much more compact model for the Ultrasparc, where only 8 time variables are used, compared to 11 time variables for the Pentiums 3 and 4.

Euclid: Low-Level TKS Generation - Runtime Analysis								
Bitwidth	Variables		Time	BDD nodes		Memory	Transition [s]$\cdot 10^{-5}$	
	states	time	h:m:s	UDS	TKS	kB	min	max
4	23	4	7.20	1257	1864	5179	4	11
5	28	6	20.24	1575	4049	6645	6	41
6	33	9	1:29.74	3974	13575	8067	8	375
7	38	11	9:13.02	7920	40680	20429	11	1089
8	43	11	53:28.06	4670	119739	38990	16	1623
Pentium 3, 1GHz, 1GB								

Table A.22: Low-Level TKS Generation for Euclid's GCD Algorithm on Pentium 3 Architecture

Euclid: Low-Level TKS Generation - Runtime Analysis								
Bitwidth	Variables		Time	BDD nodes		Memory	Transition [s]$\cdot 10^{-5}$	
	states	time	h:m:s	UDS	TKS	kB	min	max
4	23	5	4.33	1224	1801	5179	2	16
5	28	5	11.15	1572	3861	6512	3	31
6	33	8	48.48	4007	12544	8018	4	194
7	38	9	3:04.06	7970	29654	17670	5	369
8	43	11	21:12.11	5483	161476	38434	7	1154
Pentium 4, 2GHz, 512MB								

Table A.23: Low-Level TKS Generation for Euclid's GCD Algorithm on Pentium 4 Architecture

Euclid: Low-Level TKS Generation - Runtime Analysis								
Bitwidth	Variables		Time	BDD nodes		Memory	Transition [s]$\cdot 10^{-5}$	
	states	time	h:m:s	UDS	TKS	kB	min	max
4	23	5	14.91	1200	2170	5217	13	29
5	28	6	1:00.94	1584	4523	6467	20	45
6	33	7	5:10.89	4035	15495	8102	27	70
7	38	8	26:58.29	7224	40996	17029	36	183
8	43	8	2:20:49.90	4966	102972	39078	46	222
Ultrasparc III, 750MHz, 512MB								

Table A.24: Low-Level TKS Generation for Euclid's GCD Algorithm on Ultrasparc III Architecture

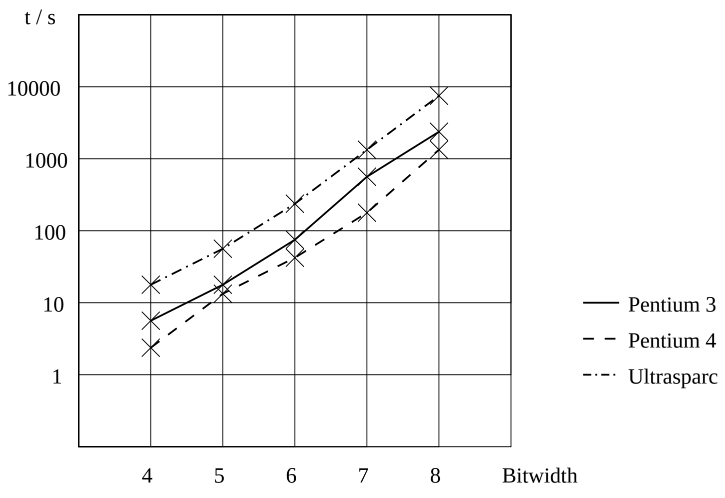


Figure A.29: Euclid: Low-Level TKS Generation Time - Log. Scale

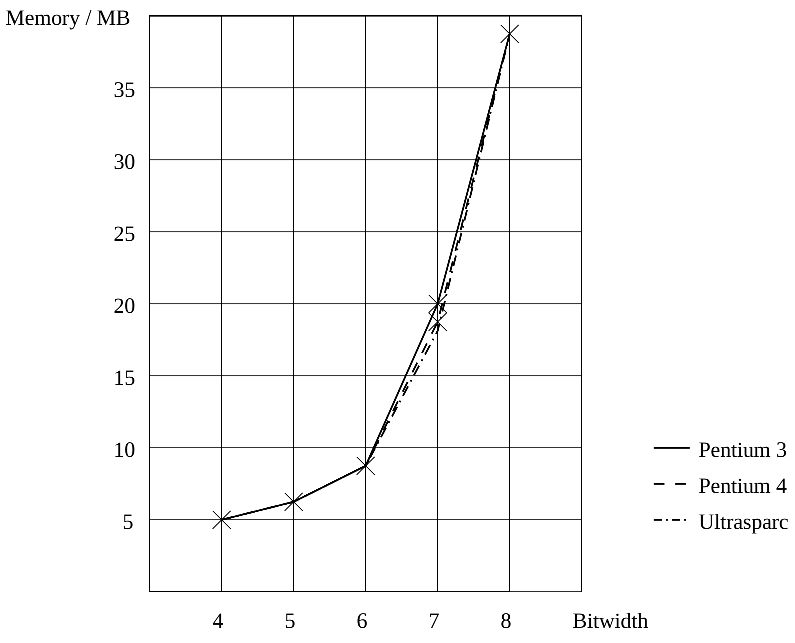


Figure A.30: Euclid: Low-Level TKS Generation Memory

A.3.4 WCET Analysis

For the WCET analysis of the Euclid's algorithm we have obtained the results given in Table A.25 for n processes. In particular, we have determined the smallest and largest number of macro steps needed in order to complete the test.

Euclid: WCET and BCET analysis - EHLA						
Bitwidth	Variables states	Time h:m:s	Memory kB	BDD nodes	BCET	WCET
4	23	0.29	4861	13286	2	16
5	28	0.54	5237	35770	2	32
6	33	1.78	5997	79716	2	64
7	38	7.87	8740	238126	2	128
8	43	13.84	15106	361788	2	256
9	48	1:14.11	52448	1072078	2	512
10	53	10:38.38	58620	1510516	2	1024
11	58	41:02.10	112830	2687860	2	2048
12	63	5:07:25.30	288358	9083536	2	4096
Pentium 3, 1GHz, 1GB						

Table A.25: WCET and BCET Analysis for Euclid's GCD Algorithm

The columns of the table are as follows: The first column denotes the instantiation of the benchmark's parameter (bits). The second column shows how many Boolean variables were necessary to encode the state transition diagram, which means that the system has 2^{var} reachable states. Column three shows the determined runtimes for the WCET analysis. Columns four and five show memory consumption, expressed in kBytes of memory and required BDD nodes respectively. Columns six and seven show the best– and worst execution times respectively.

It can be observed that Euclid's algorithm for n bits has a WCET of 2^n macro steps. The crucial macro step is the one that corresponds to the loop body that consists of a subtraction, a comparison and two tests on equality to 0. If a low-level analysis for a hardware implementation could tell us that the program for 8 bits can be implemented by a hardware circuit with a clock speed of 40 MHz, then we know that a gcd computation will be done in at least $256/40 \cdot 10^{-6}$ seconds, and for 12 bits, the circuit would require about $2^{12}/40 \cdot 10^{-6}$ seconds (≈ 0.1024 milliseconds).

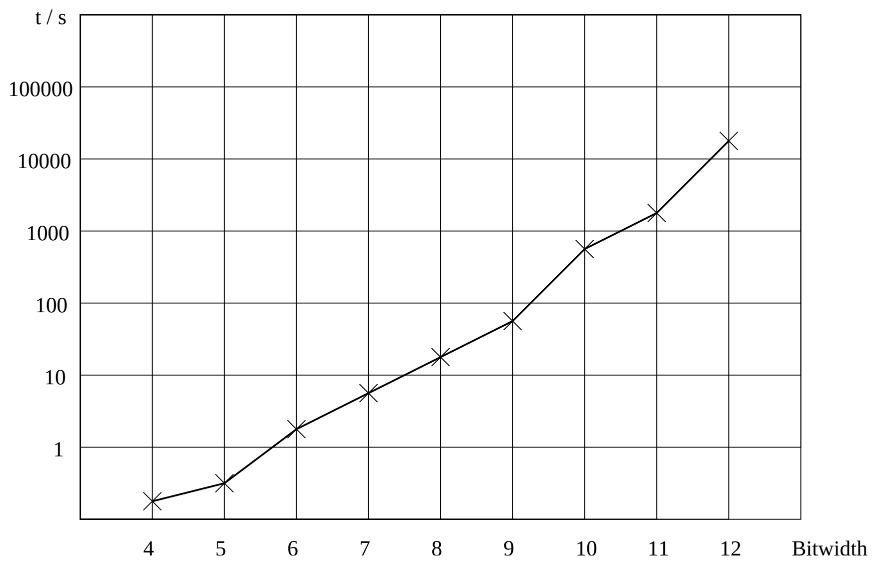


Figure A.31: Euclid: EHLA Time - Log. Scale

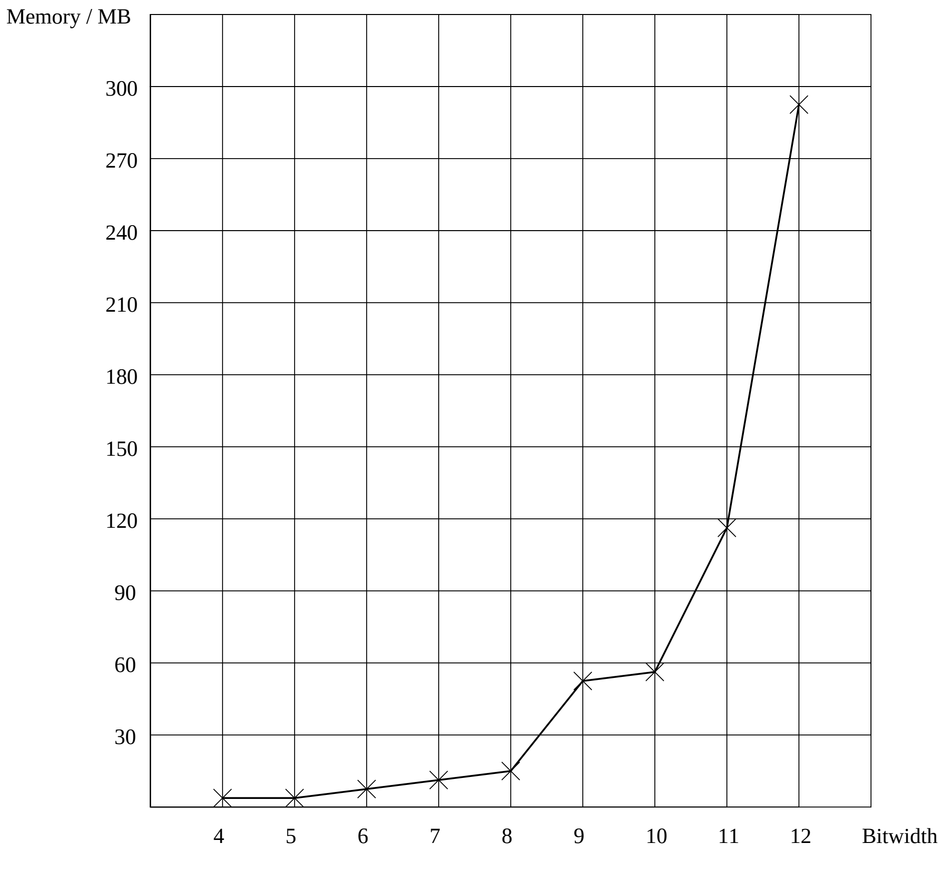


Figure A.32: Euclid: EHLA Memory

A.3.5 High-Level TKS Verification

In order to perform high-level verifi cation for the Euclid benchmark, we have tested the the operator $\mathsf{AF}^{\kappa}\varphi$, according to the results of WCET analysis. In particular, similar to the primality benchmark, we have proved the correctness of the WCET analysis by checking the following real-time specifi cation:

$$\mathsf{AF}^{\leq\ WCET}(test\_complete)$$

where $test\_complete$ simply means that the Euclid test is completed. Table A.26 contains the results for Euclid's algorithm to compute the greatest common divisor of two given n bit broad numbers.

Euclid: High-Level TKS Verifi cation					
Bitwidth	Variables		Time	Memory	BDD nodes
	states	time	h:m:s	kB	
4	23	4	0.38	4959	17374
5	28	5	1.01	5323	38836
6	33	6	5.79	7828	184982
7	38	7	48.30	20002	945350
8	43	8	2:40.66	59200	1508472
Pentium 3, 1GHz, 1GB					

Table A.26: Verifi cation of Quantitative Properties at High-Level for Euclid's GCD Algorithm

The columns of the table are as follows: The fi rst column denotes the instantiation of the benchmark's parameter (bits). The second column shows how many Boolean variables were necessary to encode the state transition diagram and the logical times on the transitions. This means that the system has 2^{s_var} reachable states and the longest transition of the system is not exceeding 2^{t_var} logical time units. Column three shows the determined runtimes for the high-level TKS verifi cation. Columns four and fi ve show memory consumption, expressed in kBytes of memory and required BDD nodes respectively.

The verifi cation results of Table A.26 show that JERRY behaves here similar to the Primality benchmark.

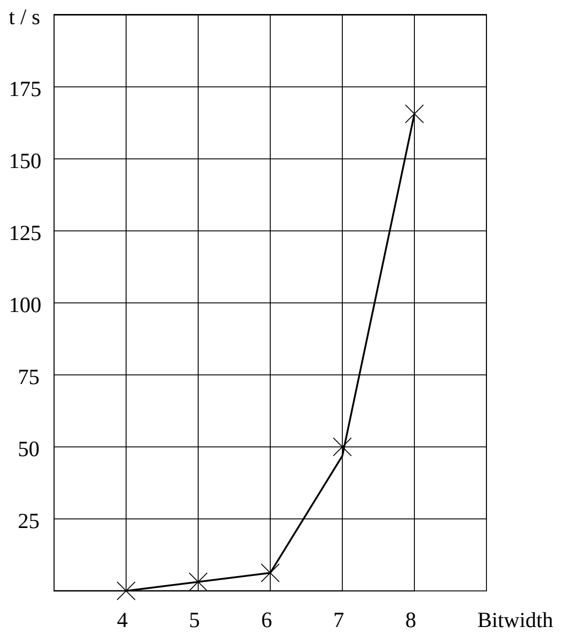


Figure A.33: Euclid: High-Level TKS Verification Time

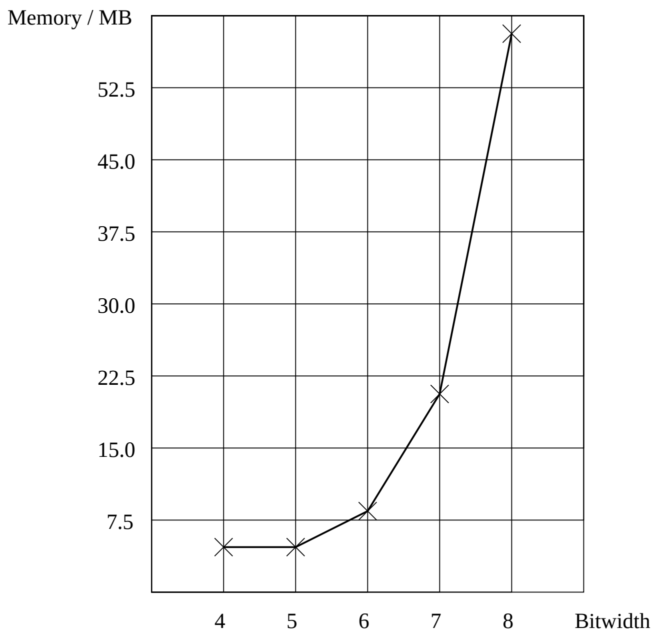


Figure A.34: Euclid: High-Level TKS Verification Memory

A.3.6 Low-Level TKS Verification

Considering the low-level verifi cation for Euclid's GCD algorithm, we have tested JERRY's performance on the low-level TKSs obtained by the runtime analysis (cf. Tables A.22, A.23 and A.24).

Here, the original systems are endowed by additional physical times, required for the code execution of the synchronous program on the appropriate architectures (cf. section 6.2). This results an enormous increase of the system's complexity.

Similar to the high-level verifi cation, we tested the system according to the results of WCET analysis. In particular, we have proved the correctness of the WCET analysis by checking the following real-time specifi cation:

$$\mathsf{AF}^{\leq\ (WCET\times\ max\_trans)}(test\_complete)$$

where $test\_complete$ simply means that the primality test is completed and max_trans is the maximum transition included in the system, as can be obtained from the Tables A.22, A.23 and A.24. The verifi cation results are shown in Tables A.27, A.28 and A.29.

In contrast to the Primality benchmark, the executable code for Euclid's GCD algorithm showed the best performance on the Ultrasparc III machine, where less time variables were needed. However, the Pentium 4 model is for 7 bits much more compact than the Ultrasparc model. This has an enormous "amplifying" effect on the verifi cation performance: the Pentium 4 model required approx. 1,9 million BDD nodes, compared to approx. 3,2 million BDD nodes for the Ultrasparc model. The worst model is the Pentium 3 one, requiring approx. 4,5 million BDD nodes. Naturally, these model sizes directly effect also the runtimes.

This example demonstrates again JERRY′s stable memory consumption. It's clearly to see that although the BBD nodes and the runtimes increase signifi cantly with the number of bits JERRY shows here again a stable, low main memory consumption, which made it possible to handle the systems. Due to JERRY′s modular design, we can again start the analysis directly at the low-level verifi cation stage, overcoming repeated (and needless) model generations.

Euclid: Low-Level TKS Verifi cation - Pentium 3					
Bitwidth	Variables		Time	Memory	BDD nodes
	states	time	h:m:s	kB	
4	23	4	1.34	5276	37814
5	28	6	10.34	7044	142058
6	33	9	1:49.48	19870	872788
7	38	11	1:44:03.43	143494	4533592
Pentium 3, 1GHz, 1GB					

Table A.27: Verifi cation of Quantitative Properties at Pentium 3 for Euclid's GCD Algorithm

Euclid: Low-Level TKS Verifi cation - Pentium 4					
Bitwidth	Variables		Time	Memory	BDD nodes
	states	time	h:m:s	kB	
4	23	5	1.20	5196	32704
5	28	5	3.87	7210	150234
6	33	8	35.11	19498	632618
7	38	9	12:46.51	65339	1868216
Pentium 3, 1GHz, 1GB					

Table A.28: Verifi cation of Quantitative Properties at Pentium 4 for Euclid's GCD Algorithm

Euclid: Low-Level TKS Verifi cation - Ultrasparc					
Bitwidth	Variables		Time	Memory	BDD nodes
	states	time	h:m:s	kB	
4	23	5	1.76	5557	53144
5	28	6	14.12	10682	351568
6	33	7	2:16.28	40473	1862084
7	38	8	58:11.51	72808	3234630
Pentium 3, 1GHz, 1GB					

Table A.29: Verifi cation of Quantitative Properties at Ultrasparc III for Euclid's GCD Algorithm

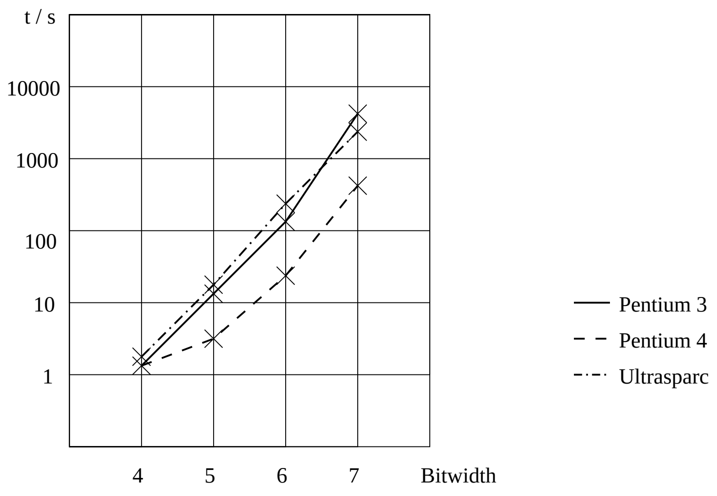


Figure A.35: Euclid: Low-Level TKS Verifi cation Time - Log. Scale

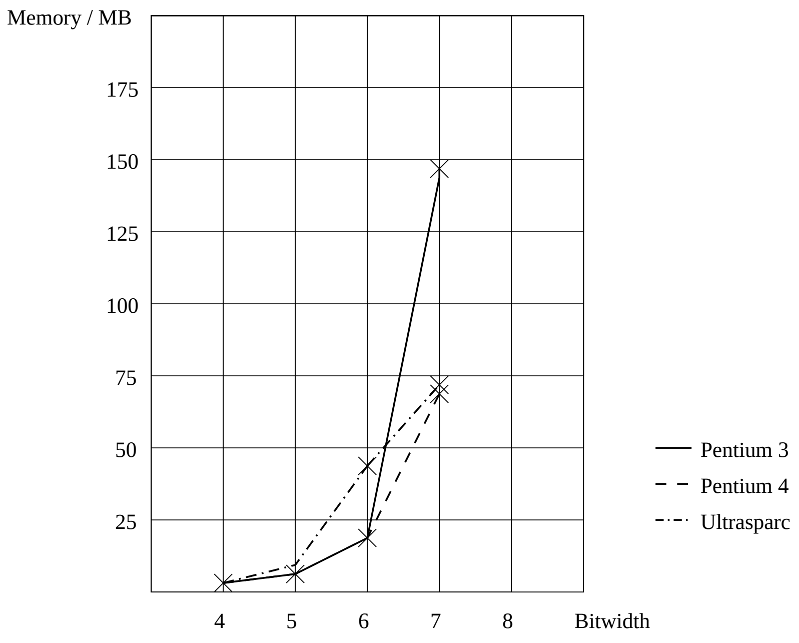


Figure A.36: Euclid: Low-Level TKS Verifi cation Memory

Appendix B

List of Logos and Abbreviations

A	The universal path quantifi er ($\forall$)
$\mathsf{A}[\cdot\ \mathsf{B}\ \cdot]$	The CTL-operator $\mathsf{A}[\cdot\ \mathsf{B}\ \cdot]$
$\mathsf{A}[\cdot\ \underline{\mathsf{B}}\ \cdot]$	The CTL-operator $\mathsf{A}[\cdot\ \underline{\mathsf{B}}\ \cdot]$
$\mathsf{A}[\cdot\ {}^{\kappa}\mathsf{B}\ \cdot]$	The left-bounded JCTL-operator $\mathsf{A}[\cdot\ {}^{\kappa}\mathsf{B}\ \cdot]$
$\mathsf{A}[\cdot\ {}^{\kappa}\underline{\mathsf{B}}\ \cdot]$	The left-bounded JCTL-operator $\mathsf{A}[\cdot\ {}^{\kappa}\underline{\mathsf{B}}\ \cdot]$
$\mathsf{A}[\cdot\ \mathsf{B}^{\kappa}\ \cdot]$	The right-bounded JCTL-operator $\mathsf{A}[\cdot\ \mathsf{B}^{\kappa}\ \cdot]$
$\mathsf{A}[\cdot\ \underline{\mathsf{B}}^{\kappa}\ \cdot]$	The right-bounded JCTL-operator $\mathsf{A}[\cdot\ \underline{\mathsf{B}}^{\kappa}\ \cdot]$
AF	The CTL-operator AF
AF^{κ}	The JCTL-operator AF^{κ}
AG	The CTL-operator AG
AG^{κ}	The JCTL-operator AG^{κ}
$\mathsf{A}[\cdot\ \mathsf{U}\ \cdot]$	The CTL-operator $\mathsf{A}[\cdot\ \mathsf{U}\ \cdot]$
$\mathsf{A}[\cdot\ \underline{\mathsf{U}}\ \cdot]$	The CTL-operator $\mathsf{A}[\cdot\ \underline{\mathsf{U}}\ \cdot]$
$\mathsf{A}[\cdot\ {}^{\kappa}\mathsf{U}\ \cdot]$	The left-bounded JCTL-operator $\mathsf{A}[\cdot\ {}^{\kappa}\mathsf{U}\ \cdot]$
$\mathsf{A}[\cdot\ {}^{\kappa}\underline{\mathsf{U}}\ \cdot]$	The left-bounded JCTL-operator $\mathsf{A}[\cdot\ {}^{\kappa}\underline{\mathsf{U}}\ \cdot]$
$\mathsf{A}[\cdot\ \mathsf{U}^{\kappa}\ \cdot]$	The right-bounded JCTL-operator $\mathsf{A}[\cdot\ \mathsf{U}^{\kappa}\ \cdot]$
$\mathsf{A}[\cdot\ \underline{\mathsf{U}}^{\kappa}\ \cdot]$	The right-bounded JCTL-operator $\mathsf{A}[\cdot\ \underline{\mathsf{U}}^{\kappa}\ \cdot]$
AX	The CTL-operator AX
$\mathsf{A}\underline{\mathsf{X}}$	The CTL-operator $\mathsf{A}\underline{\mathsf{X}}$
AX^{κ}	The JCTL-operator AX^{κ}
$\mathsf{A}\underline{\mathsf{X}}^{\kappa}$	The JCTL-operator $\mathsf{A}\underline{\mathsf{X}}^{\kappa}$
BCET	Best Case Execution Time
BDD	Binary Decision Diagram
B	The modal operator *weak before*
$\underline{\mathsf{B}}$	The modal operator *strong before*
Cadence-SMV	A variant of the SMV verifi cation tool by Cadence

CTL	The branching-time temporal logic CTL (Computation Tree Logic)
CUDD	The BDD-package CUDD (Colorado University Decision Diagrams)
E	The existence path quantifi er ($\exists$)
$\mathsf{E}[\cdot\ \mathsf{B}\ \cdot]$	The CTL-operator $\mathsf{E}[\cdot\ \mathsf{B}\ \cdot]$
$\mathsf{E}[\cdot\ \underline{\mathsf{B}}\ \cdot]$	The CTL-operator $\mathsf{E}[\cdot\ \underline{\mathsf{B}}\ \cdot]$
$\mathsf{E}[\cdot\ {}^{\kappa}\mathsf{B}\ \cdot]$	The left-bounded JCTL-operator $\mathsf{E}[\cdot\ {}^{\kappa}\mathsf{B}\ \cdot]$
$\mathsf{E}[\cdot\ {}^{\kappa}\underline{\mathsf{B}}\ \cdot]$	The left-bounded JCTL-operator $\mathsf{E}[\cdot\ {}^{\kappa}\underline{\mathsf{B}}\ \cdot]$
$\mathsf{E}[\cdot\ \mathsf{B}^{\kappa}\ \cdot]$	The right-bounded JCTL-operator $\mathsf{E}[\cdot\ \mathsf{B}^{\kappa}\ \cdot]$
$\mathsf{E}[\cdot\ \underline{\mathsf{B}}^{\kappa}\ \cdot]$	The right-bounded JCTL-operator $\mathsf{E}[\cdot\ \underline{\mathsf{B}}^{\kappa}\ \cdot]$
EF	The CTL-operator EF
EF^{κ}	The JCTL-operator EF^{κ}
EG	The CTL-operator EG
EG^{κ}	The JCTL-operator EG^{κ}
Equinox	Formal Framework for the Specifi cation, Modelling, Verifi cation and Runtime Anylysis of Real-Time Systems
Esterel	The synchronous language Esterel
$\mathsf{E}[\cdot\ \mathsf{U}\ \cdot]$	The CTL-operator $\mathsf{E}[\cdot\ \mathsf{U}\ \cdot]$
$\mathsf{E}[\cdot\ \underline{\mathsf{U}}\ \cdot]$	The CTL-operator $\mathsf{E}[\cdot\ \underline{\mathsf{U}}\ \cdot]$
$\mathsf{E}[\cdot\ {}^{\kappa}\mathsf{U}\ \cdot]$	The left-bounded JCTL-operator $\mathsf{E}[\cdot\ {}^{\kappa}\mathsf{U}\ \cdot]$
$\mathsf{E}[\cdot\ {}^{\kappa}\underline{\mathsf{U}}\ \cdot]$	The left-bounded JCTL-operator $\mathsf{E}[\cdot\ {}^{\kappa}\underline{\mathsf{U}}\ \cdot]$
$\mathsf{E}[\cdot\ \mathsf{U}^{\kappa}\ \cdot]$	The right-bounded JCTL-operator $\mathsf{E}[\cdot\ \mathsf{U}^{\kappa}\ \cdot]$
$\mathsf{E}[\cdot\ \underline{\mathsf{U}}^{\kappa}\ \cdot]$	The right-bounded JCTL-operator $\mathsf{E}[\cdot\ \underline{\mathsf{U}}^{\kappa}\ \cdot]$
EX	The CTL-operator EX
$\mathsf{E}\underline{\mathsf{X}}$	The CTL-operator $\mathsf{E}\underline{\mathsf{X}}$
EX^{κ}	The JCTL-operator EX^{κ}
$\mathsf{E}\underline{\mathsf{X}}^{\kappa}$	The JCTL-operator $\mathsf{E}\underline{\mathsf{X}}^{\kappa}$
F	The modal operator *future*
G	The modal operator *globally*
I_S	Stuttering Interpretation of a timed transition system
I_J	Jumping Interpretation of a timed transition system
JCTL	The real-time temporal logic JCTL (Time-Jumping Computation Tree Logic)
$\mathsf{JCTL}^{\leq}$	A fragment of the real-time temporal logic JCTL
JERRY	The real-time verifi cation tool JERRY (JCTL vERifi er for Real-time sYstems)
Kronos	The real-time verifi cation tool Kronos
LTL	The linear-time temporal logic LTL (Linear-Time Temporal Logic)
NuSMV	A variant of the SMV verifi cation tool based on the CUDD-package
Quartz	The synchronous language Quartz
SKS	A timed transition system based on stuttering interpretation (Stuttering Kripke Structure)
SMV	The verifi cation tool SMV (System Model Verifi er)

TCTL	The real-time temporal logic TCTL (Timed Computation Tree Logic)
TKS	The real-time formal model TKS (Timed Kripke Structure)
U	The modal operator *weak until*
$\underline{\mathsf{U}}$	The modal operator *strong until*
UDS	A qualitative Kripke structure (Unit Delay Structure)
UPPAAL	The real-time verifi cation tool UPPAAL
WCET	Worst Case Execution Time
X	The modal operator *weak next*
$\underline{\mathsf{X}}$	The modal operator *strong next*

Bibliography

[1] ALUR, R., COURCOUBETIS, C., AND DILL, D. Model Checking for Real-Time Systems. In *IEEE Symposium on Logic in Computer Science (LICS)* (Washington, D.C., June 1990), IEEE Computer Society Press, pp. 414–425.

[2] ALUR, R., COURCOUBETIS, C., AND DILL, D. Model Checking in Dense Real-time. Tech. rep., Stanford University, University of Crete, 1991.

[3] ALUR, R., COURCOUBETIS, C., AND DILL, D. Model-checking in dense real-time. *Information and Computation 104*, 1 (1993), 2–34.

[4] ALUR, R., AND DILL, D. Automata for Modeling Real-Time Systems. In *International Colloquium on Automata, Languages and Programming (ICALP)* (New York, 1990), vol. 433 of *LNCS*, Springer-Verlag.

[5] BALDAMUS, M., AND SCHNEIDER, K. Extending Esterel by asynchronous concurrency. In *GI/GMM/ITG Fachtagung zum Entwurf Integrierter Schaltungen* (Darmstadt, Germany, September 1999), VDE Verlag.

[6] BENGTSSON, J., LARSEN, K., LARSSON, F., PETTERSSON, P., AND YI, W. UPPAAL in 1995. In *Tools and Algorithms for the Construction and Analysis of Systems* (March 1996), no. 1055 in Lecture Notes In Computer Science, Springer-Verlag, pp. 431–434.

[7] BERNAT, G., AND BURNS, A. An approach to symbolic worst-case execution time analysis. In *IFAC Workshop on Real-Time Programming* (2000).

[8] BERRY, G. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 1998.

[9] BERRY, G. The constructive semantics of pure Esterel. http://www-sop.inria.fr/meije/esterel/, July 1999.

[10] BERRY, G. The Esterel v5_91 language primer. http://www.esterel.org, June 2000.

[11] BERTHET, C., COUDERT, O., AND MADRE, J. C. New ideas on symbolic manipulations of finite state machines. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)* (1990).

[12] BERTIN, V., POIZE, M., PULOU, J., AND SIFAKIS, J. Towards validated real-time software. In *Euromicro Conference on Real Time Systems* (Stockholm, 2000), pp. 157–164.

[13] BORNOT, A., SIFAKIS, J., AND TRIPAKIS, S. Modelling urgency in timed systems. In *International Symposium: Compositionality - The Significant Difference* (Holstein, Germany, 1997), vol. 1536 of *LNCS*.

[14] BORNOT, S., AND SIFAKIS, J. An algebraic framework for urgency. *Information and Computation 163, Elsevier publisher* (2000), 172–202.

[15] BOUSSINOT, F. SugarCubes implementation of causality. Research Report 3487, Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia Antipolis Cedex (France), September 1998.

[16] BROWNE, M., CLARKE, E., DILL, D., AND MISHRA, B. Automatic Verifi cation of Sequential Circuits Using Temporal Logic. *IEEE Transactions on Computers C-35*, 12 (December 1986), 1034–1044.

[17] BRYANT, R. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers C-35*, 8 (August 1986), 677–691.

[18] BURCH, J., CLARKE, E., AND LONG, D. Representing Circuits More Effi ciently in Symbolic Model Checking. In *ACM/IEEE Design Automation Conference (DAC)* (Los Alamitos, CA, June 1991), IEEE Computer Society Press, pp. 403–407.

[19] BURCH, J., CLARKE, E., AND LONG, D. Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration (VLSI)* (Edinburgh, Scotland, August 1991), A. Halaas and P. Denyer, Eds., IFIP Transactions, North-Holland, pp. 49–58.

[20] BURCH, J., CLARKE, E., MCMILLAN, K., AND DILL, D. Sequential Circuit Verifi - cation Using Symbolic Model Checking. In *ACM/IEEE Design Automation Conference (DAC)* (Los Alamitos, CA, June 1990), ACM/IEEE, IEEE Society Press, pp. 46–51.

[21] BURCH, J., CLARKE, E., MCMILLAN, K., DILL, D., AND HWANG, L. Symbolic Model Checking: 10^{20} States and Beyond. In *IEEE Symposium on Logic in Computer Science (LICS)* (Washington, D.C., June 1990), IEEE Computer Society Press, pp. 1–33.

[22] BURCH, J., CLARKE, E., MCMILLAN, K., DILL, D., AND HWANG, L. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computing 98*, 2 (June 1992), 142–170.

[23] BURNS, A., AND WELLINGS, A. *Real-Time Systems and their Programming Languages*. Addison-Wesley, 1990.

[24] BUTLER, K., ROSS, D., KAPUR, R., AND MERCER, M. Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. In *ACM/IEEE Design Automation Conference (DAC)* (1991), ACM/IEEE, IEEE, pp. 417–420.

[25] BUTTAZZO, G. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[26] CADENCE SMV. Website. http://www-cad.eecs.berkeley.edu/ kenmcmil/smv/.

[27] CAMPOS, S., AND CLARKE, E. Real-Time Symbolic Model Checking for Discrete Time Models. In *Theories and Experiences for Real-Time System Development* (May 1994), T. Rus and C. Rattray, Eds., AMAST Series in Computing, World Scientific Press, AMAST Series in Computing.

[28] CAMPOS, S., CLARKE, E., MARRERO, W., AND MINEA, M. Verus: a tool for quantitative analysis of finite-state real-time systems. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems* (1995).

[29] CAMPOS, S., CLARKE, E., AND MINEA, M. The Verus tool: A quantitative approach to the formal verification of real-time systems. In *Conference on Computer Aided Verification (CAV)* (June 1997), O. Grumberg, Ed., vol. 1254 of *LNCS*, Springer Verlag, pp. 452–455.

[30] CERANS, K. Decidability of bisimulation equivalences for parallel timer procasses. In *Conference on Computer Aided Verification (CAV'92)* (Berlin, 1992), vol. 663 of *LNCS*, Springer Verlag.

[31] CIMATTI, A., CLARKE, E., GIUNCHIGLIA, F., AND ROVERI, M. NUSMV: A new symbolic model verifier. In *Conference on Computer Aided Verification (CAV)* (Trento, Italy, 1999), N. Halbwachs and D. Peled, Eds., vol. 1633 of *LNCS*, Springer-Verlag, pp. 495–499.

[32] CLARKE, E., AND EMERSON, E. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Workshop on Logics of Programs* (Yorktown Heights, New York, May 1981), D. Kozen, Ed., vol. 131 of *LNCS*, Springer-Verlag, pp. 52–71.

[33] CLARKE, E., EMERSON, E., AND SISTLA, A. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems 8*, 2 (April 1986), 244–263.

[34] CLARKE, E., GRUMBERG, O., AND LONG, D. Verification Tools for Finite State Concurrent Systems. In *A Decade of Concurrency-Reflections and Perspectives* (Noordwijkerhout, Netherlands, June 1993), J. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., vol. 803 of *LNCS*, REX School/Symposium, Springer-Verlag, pp. 124–175.

[35] CLARKE, E., GRUMBERG, O., AND LONG, D. Model checking and abstraction. *ACM Transactions on Programming Languages and systems 16*, 5 (September 1994), 1512–1542.

[36] CLARKE, E., GRUMBERG, O., AND LONG, D. Model checking. In *Deductive Program Design* (Marktoberdorf, Germany, 1996), vol. 152 of *Nato ASI Series F*, Springer-Verlag.

[37] COUDERT, O., BERTHET, C., AND MADRE, J. Verifi cation of synchronous sequential machines using symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems* (Grenoble, France, June 1989), vol. 407 of *LNCS*, Springer-Verlag, pp. 365–373.

[38] COUSOT, P. Abstract interpretation. *ACM Computing Surveys 28*, 2 (June 1996), 324–328.

[39] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unifi ed lattice model for static analysis of programs by construction or approximation of fi xpoints. In *ACM Symposium on Principles of Programming Languages (POPL)* (1977), ACM, pp. 238–252.

[40] DAWS, C., OLIVERO, A., TRIPAKIS, S., AND YOVINE, S. The tool KRONOS. In *Hybrid Systems III* (1996), vol. 1066 of *LNCS*, Springer.

[41] DAWS, C., OLIVERO, A., AND YOVINE, S. Verifying ET-LOTOS programs with KRONOS. In *International Conference on Formal Description Techniques* (1994).

[42] DAWS, C., AND TRIPAKIS, S. Model checking of real-time reachability properties using abstractions. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (1998).

[43] DIJKSTRA, E. Guarded commands, nondeterminiacy and formal derivation of programs. *Communications of the ACM 18*, 8 (August 1975), 453–457.

[44] ECL HOMEPAGE. Website. http://www-cad.eecs.berkeley.edu/

[45] EDWARDS, S. Compiling Esterel into sequential code. In *Design Automation Conference (DAC)* (Los Angeles, California, June 2000), pp. 322–327.

[46] EDWARDS, S., MA, T., AND DAMIANO, R. Using a hardware model checker to verify software. In *International Conference on ASIC (ASICON)* (Shanghai, China, 2001).

[47] EMERSON, A., MOK, A., SISTLA, A., AND SRINIVASAN, J. Quantitative temporal reasoning. In *C3 Workshop on Automatic Verification of Finite State Systems* (1989), W.-P. de Roever, Ed., pp. S7–1–S7–15.

[48] EMERSON, E. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science* (Amsterdam, 1990), J. van Leeuwen, Ed., vol. B, Elsevier Science Publishers, pp. 996–1072.

[49] EMERSON, E. Real time and the μ-calculus. In *REX'91 Workshop on Real Time: Theory in Practice* (1991), W.-P. de Roever, Ed., Springer Verlag, LNCS 600, pp. 176–194.

[50] EMERSON, E., AND CLARKE, E. Characterizing correctness properties of parallel programs as fi xpoints. In *International Colloquium on Automata, Languages and Programming (ICALP)* (Berlin, 1981), vol. 85 of *LNCS*, Springer-Verlag, pp. 169–181.

[51] EMERSON, E., MOK, A., SISTLA, A., AND SRINIVASAN, J. Quantitative Temporal Reasoning. *Journal of Real-Time Systems 4* (1992), 331–352.

[52] EMERSON, E., AND SISTLA, A. P. Symmetry and model checking. In *Workshop on Computer Aided Verification (CAV)* (June/July 1993), C. Courcoubetis, Ed.

[53] ERMEDAHL, A., AND GUSTAFSSON, J. Deriving annotations for tight calculation of execution time. In *International European Conference on Parallel Processing (EuroPar)* (Passau, Germany, 1997), vol. 1300 of *LNCS*, Springer Verlag, pp. 1298–1307.

[54] ESTEREL. Website. http://www.esterel.org.

[55] ESTEREL-TECHNOLOGY. Website. http://www.esterel-technologies.com.

[56] FORTH, R., AND MOLITOR, P. An effi cient heuristic for state encoding minimizing the BDD representations of the transition relations of fi nite state machines. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)* (Tokyo, Japan, 2000), pp. 61–66.

[57] FRÖSSL, J., GERLACH, J., AND KROPF, T. An Effi cient Algorithm for Real-Time Model Checking. In *European Design and Test Conference (EDTC)* (Paris, France, March 1996), IEEE Computer Society Press (Los Alamitos, California), pp. 15–21.

[58] GHOSH, S., MARTONOSI, M., AND MALIK, S. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems 21*, 4 (1999), 703–746.

[59] GNU. gcc website. http://www.gcc.gnu.org.

[60] HALBWACHS, N. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, 1993.

[61] HALBWACHS, N., CASPI, P., RAYMOND, P., AND PILAUD, D. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE 79*, 9 (sep 1991), 1305–1320.

[62] HALBWACHS, N., FERNANDEZ, J.-C., AND BOUAJJANNI, A. An executable temporal logic to express safety properties and its connection with the language Lustre. In *Symposium on Lucid and Intensional Programming (ISLIP)* (Quebec, Canada, April 1993).

[63] HALFHILL, T. Embedded market breaks new ground, 2000. Microprocessor Report.

[64] HAREL, D., AND NAAMAD, A. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engenieering Methods 5*, 4 (1996).

[65] HARMON, M. G., BAKER, T. P., AND WHALLEY, D. B. A retargetable technique for predicting execution time. In *IEEE Real-Time Systems Symposium* (1992), pp. 68–77.

[66] HEALY, C., VAN ENGELEN, R., AND WHALLEY, D. A general approach for the tight timing predictions of non-rectangular loops. In *IEEE Real-Time Technology and Applications Symposium* (1999).

[67] HENZINGER, T., NICOLLIN, X., SIFAKIS, J., AND YOVINE, S. Symbolic Model Checking for Real-Time Systems. In *IEEE Symposium on Logic in Computer Science (LICS)* (Santa-Cruz, California, June 1992), IEEE Computer Society Press, pp. 394–406.

[68] HO, P. H., AND WONG-TOI, H. Automated analysis of an audio control protocol. In *Conference on Computer Aided Verification (CAV)* (Liege, Belgium, July 1995), P. Wolper, Ed., vol. 939 of *LNCS*, Springer Verlag, pp. 381–394.

[69] HOARE, C. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (October 1969), 576–583.

[70] HOARE, C. Communicating sequential processes. *Communications of the ACM 21*, 8 (August 1978), 666–677.

[71] HOLZMANN, G. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[72] INTEL. Intel vtune website. http://www.intel.com/software/products/vtune/vpa.

[73] JAGADEESAN, L. J., PUCHOL, C., AND OLNHAUSEN, J. E. V. Safety property verifi - cation of Esterel programs and applications to telecommunications software. In *Conference on Computer Aided Verification (CAV)* (Liege, Belgium, July 1995), P. Wolper, Ed., vol. 939 of *LNCS*, Springer Verlag, pp. 127–140.

[74] JESTER HOME PAGE. Website. http://www.parades.rm.cnr.it/projects/jester/jester.html.

[75] KRONOS. Website. http://www-verimag.imag.fr/TEMPORISE/kronos/.

[76] LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering 3*, 2 (March 1977), 125–143.

[77] LAMPORT, L. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems* (1987).

[78] LARSEN, K., AND YI, W. Time-abstracted bisimulation: Implicit specifi cation and decidability. In *MFPS 93* (1993), vol. 802 of *Lecture Notes in Computer Scienece*, Springer Verlag.

[79] LASSEZ, J.-L., NGUYEN, V., AND SONENBERG, E. Fixed point theorems and semantics. a folk tale. *Information Processing Letters 14*, 3 (1982), 112–116.

[80] LIU, Y., AND GOMEZ, G. Automatic accurate time-bound analysis for high-level languages. In *Languages, Compilers and Tools for Embedded Systems (LCTES)* (1998).

[81] LOETZBEYER, A. *Temporale Realzeitverifikation*. PhD thesis, Universitaet Karlsruhe, March 1999.

[82] LOGOTHETIS, G., AND SCHNEIDER, K. Abstraction from counters: An application on real-time systems. In *Design, Automation and Test in Europe (DATE)* (Paris,France, March 2000), IEEE Computer Society Press, pp. 486–493.

[83] LOGOTHETIS, G., AND SCHNEIDER, K. A new approach to the specifi cation and verifi cation of real-time systems. In *Euromicro Conference on Real-Time Systems* (Delft, The Netherlands, June 2001), IEEE Computer Society, pp. 171–180. http://goethe.ira.uka.de/fmg/ps/LoSc01.ps.gz.

[84] LOGOTHETIS, G., AND SCHNEIDER, K. Symbolic model checking of real-time systems. In *International Symposium on Temporal Representation and Reasoning* (Cividale del Friuli, Italy, June 2001), IEEE/ACM, pp. 214–223. http://goethe.ira.uka.de/fmg/ps/LoSc01a.ps.gz.

[85] LOGOTHETIS, G., AND SCHNEIDER, K. Extending synchronous languages for generating abstract real-time models. In *European Conference on Design, Automation and Test in Europe (DATE)* (Paris, France, March 2002), IEEE Computer Society, pp. 795–802. http://goethe.ira.uka.de/fmg/ps/LoSc02.ps.gz.

[86] LOGOTHETIS, G., AND SCHNEIDER, K. Exact high level wcet analysis of synchronous programs by symbolic state space exploration. In *European Conference on Design, Automation and Test in Europe (DATE)* (Munich, Germany, March 2003), IEEE Computer Society, pp. 196–203. http://goethe.ira.uka.de/fmg/ps/LoSc03.ps.gz.

[87] LOGOTHETIS, G., SCHNEIDER, K., AND METZLER, C. Exact low-level runtime analysis of synchronous programs for formal verifi cation of real-time systems. In *Forum on Specification and Design Languages (FDL)* (Frankfurt, Germany, September 2003), Kluwer Academic Publishers, pp. 385–404. http://goethe.ira.uka.de/fmg/ps/LoSM03b.ps.gz.

[88] LOGOTHETIS, G., SCHNEIDER, K., AND METZLER, C. Generating formal models for real-time verifi cation by exact low-level analysis of synchronous programs. In *24th IEEE International Real-Time Systems Symposium (RTSS)* (Cancun, Mexico, December 2003), IEEE Computer Society Press. http://goethe.ira.uka.de/fmg/ps/LoSM03c.ps.gz.

[89] LOGOTHETIS, G., SCHNEIDER, K., AND METZLER, C. Runtime analysis of synchronous programs for low-level real-time verification. In *16th Symposium on Intergrated Circuits and System Design (SBCCI)* (Sao Paulo, Brazil, September 2003), IEEE Computer Society Press, pp. 211–216. http://goethe.ira.uka.de/fmg/ps/LoSM03a.ps.gz.

[90] LOISEAUX, C., GRAF, S., SIFAKIS, J., BOUAJJANI, A., AND BENSALEM, S. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design 6* (February 1995), 1–35.

[91] LUNDQVIST, T., AND STENSTRÖM, P. Integrating path and timing analysis using instruction-level simulation techniques. In *Languages, Compilers and Tools for Embedded Systems (LCTES)* (1998).

[92] LYNCH, N., AND VAADRAGER, F. Forward and backward simulation for timing-based systems. In *Workshop on Stepwise Refinement of Distributed Systems* (1992), vol. 600 of *LNCS*, Springer-Verlag, pp. 397–446.

[93] MCMILLAN, K. The SMV system, symbolic model checking - an approach. Tech. Rep. CMU-CS-92-131, Carnegie Mellon University, 1992.

[94] MCMILLAN, K. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[95] MCMILLAN, K. Cadence SMV. Available on: http://www-cad.eecs.berkeley.edu/~kenmcmil, 2000.

[96] MILNER, R. *Communication and Concurrency*. Prentice-Hall International, London, 1989.

[97] MUELLER, F. Timing analysis for instruction caches, 2000.

[98] MUELLER, F., WHALLEY, D., AND HARMON, M. Predicting instruction cache behavior, 1993.

[99] NICOLLIN, X., AND SIFAKIS, J. An overview and synthesis on timed process algebras. In *Computer Aided Verification (CAV'91)* (Aalborg, Denmark, 1991), LNCS 575, Springer Verlag.

[100] PELED, D. Combining partial order reductions with on-the-fly model-checking. In *Conference on Computer Aided Verification (CAV)* (Stanford, California, USA, June 1994), D. L. Dill, Ed., vol. 818 of *LNCS*, Springer-Verlag, pp. 377–390.

[101] PNUELI, A. The temporal logic of programs. In *Symposium on Foundations of Computer Science* (New York, 1977), vol. 18, IEEE, pp. 46–57.

[102] PNUELI, A. Applications of temporal logic to the specifi cation and verifi cation of reactive systems: A survey of current trends. In *Current Trends in Concurrency* (New-York, 1986), J. Baker, W.-P. de Roever, and G. Rozenberg, Eds., vol. 224 of *LNCS*, Springer-Verlag, pp. 510–584.

[103] PUSCHNER, P. Worst-case execution time analysis at low cost. In *Distributed Computer Control Systems* (Seoul, Korea, 1997), pp. 16–21.

[104] PUSCHNER, P., AND KOZA, C. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems 1*, 1 (1989), 159–176.

[105] QUEILLE, J., AND SIFAKIS, J. Specifi cation and verifi cation of concurrent systems in CESAR. In *International Symposium in Programming* (1981).

[106] RAMCHANDANI, C. Analysis of asynchronous concurrent systems by timed petri nets. In *Project MAC Technical Report MAC-TR-120* (1974), Massachusetts Institute for Technology, Cambridge MA.

[107] RESSOURCES, M. D. Embedded processor forum. http://www.mdronline.com.

[108] RUF, J., AND KROPF, T. Using MTBDDs for composition and model checking of real-time systems. In *International Conference on Formal Methods in Computer Aided Design (FMCAD)* (November 1998), vol. 1166 of *LNCS*, Springer.

[109] RUF, J., AND KROPF, T. Using MTBDDs for discrete timed symbolic model checking. *Multiple-Valued Logic - An International Journal* (1998). Special Issue on Decision Diagrams, Gordon and Breach Publishers.

[110] SCHNEIDER, K. CTL and equivalent sublanguages of CTL*. In *IFIP Conference on Computer Hardware Description Languages and their Applications (CHDL)* (Toledo,Spain, April 1997), C. Delgado Kloos, Ed., IFIP, Chapman and Hall, pp. 40–59.

[111] SCHNEIDER, K. A verifi ed hardware synthesis for Esterel. In *International IFIP Workshop on Distributed and Parallel Embedded Systems* (Schloß Ehringerfeld, Germany, 2000), F. Rammig, Ed., Kluwer Academic Publishers, pp. 205–214.

[112] SCHNEIDER, K. Embedding imperative synchronous languages in interactive theorem provers. In *International Conference on Application of Concurrency to System Design (ICACSD 2001)* (Newcastle upon Tyne, UK, June 2001), IEEE Computer Society Press, pp. 143–156. http://goethe.ira.uka.de/fmg/ps/Schn01a.ps.gz.

[113] SCHNEIDER, K. *Exploiting Hierarchies in Temporal Logics, Finite Automata, Arithmetics, and μ-Calculus for Efficiently Verifying Reactive Systems*. Habilitation Thesis. University of Karlsruhe, Faculty of Informatics (Fakultät für Informatik), 2001.

[114] SCHNEIDER, K. Formal reasoning about synchronous programming languages. Internal report 2001-15, University of Karlsruhe, December 2001. http://goethe.ira.uka.de/fmg/ps/Schn01c.ps.gz.

[115] SCHNEIDER, K. Proving the equivalence of microstep and macrostep semantics. In *International Conference on Theorem Proving in Higher Order Logic* (Hampton, Virginia, USA, 2002), vol. 2410 of *LNCS*, Springer Verlag, pp. 314–331. http://goethe.ira.uka.de/fmg/ps/Schn02.ps.gz.

[116] SCHNEIDER, K., AND WENZ, M. A new method for compiling schizophrenic synchronous programs. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)* (Atlanta, USA, November 2001), ACM, pp. 49–58. http://goethe.ira.uka.de/fmg/ps/ScWe01.ps.gz.

[117] SHYAMASUNDAR, R., AND AGHAV, J. Realizing real-time systems from synchronous language specifi cations. In *Real Time Systems Symposium, Work in Progress Session* (Orlando, Florida, USA, November 2000), IEEE.

[118] SIFAKIS, J. A unifi ed approach for studying the properties of transition systems. *Theoretical Computer Science*, 18 (1982), Elsevier Publishers pp. 227–258.

[119] SIFAKIS, J., AND YOVINE, S. Compositional specifi cation of timed systems. In *13th Annual Symposium on Theoretical Aspects of Computer Science (STACS96)* (1996), no. 1046 in LNCS, Springer-Verlag, pp. 347–359.

[120] SOMENZI, F. CUDD: CU decision diagram package, release 2.3.0, 1998. ftp://vlsi.colorado.edu/pub/ and http://vlsi.Colorado.EDU/.

[121] STOREY, N. *Safety-Critical Computer Systems*. Addison-Wesley Publishing Company, 1996.

[122] TARSKI, A. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific J. Math 5* (1955), 285–309.

[123] TELELOGIC. Website. http://www.telelogic.com.

[124] TRIPAKIS, S., AND YOVINE, S. Analysis of timed systems based on time-abstracting bisimulations. In *Conference on Computer Aided Verification (CAV)* (New Brunswick, NJ, USA, July/August 1996), R. Alur and T. A. Henzinger, Eds., vol. 1102 of *LNCS*, Springer Verlag, pp. 232–243.

[125] TRIPAKIS, S., AND YOVINE, S. Analysis of timed systems based on time-abstracting bisimulations. *Formal Methods in System Design 18*, 1 (January 2001), 25–68.

[126] UPPAAL. Website. http://www.uppaal.com.