Forward and Time-Jumping Symbolic Model Checking for Real Time Systems

Georgios Logothetis

University of Karlsruhe
Department of Computer Science
Institute for Computer Design and Fault Tolerance
P.O. Box 6980, 76128 Karlsruhe, Germany
George.Logothetis@informatik.uni-karlsruhe.de

Abstract. Synchronous languages are widely used in industrial applications for the design and implementation of real-time embedded and reactive systems and are also well-suited for real-time verification purposes, since they have clean formal semantics. In this paper we focuse on the real-time temporal logic JCTL, which can directly support the real-time formal verification of synchronous programs for the design of systems in earlier (high-level) as well as in later (low-level) design stages, creating a bridging between industrial real-time descriptions and formal real-time verification. We extend the model-checking capabilities of JCTL, by introducing new forward symbolic model-checking techniques, allowing JCTL to benefit from both, forward–, as well as traditional backward state traversal methods and of course, their combination. In addition to this, we also introduce special methods that allow the algorithms for model-checking JCTL formulae to perform time-jumps during the state space traversal. In other words, the algorithms are able to perform less iterations than the length of the traversed paths (measured in time units).

1 Introduction

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. Decreasing time-to-market and the overall design costs requires guaranties for the correctness of a real-time systems' design, i.e. a proof that certain actions are executed within some strict runtime limits or that they will start only after some point of time. For this purpose, several approaches to the verification of real-time systems have been developed [1,13,5,2,6,21], that are based on different formalisms for describing finite state transition systems endowed with some notion of time. The idea is hereby to design, develop and realize a real-time system within a formal framework. This requires a description format, which supports formal semantics, in order to construct a real-time formal model, that can be used for formal verification purposes.

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, most of the existing real-time verification techniques, like [1,13,5,2,6], require the use of special input formats for the description of real-time systems in order to obtain formal models that can be used for verifi cation purposes. Taking advantage of these techniques in industrial applications, which are usually already described in other formats, requires the 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, since it can not be guaranteed that the re-described system will have the same behavior as the original one.

In contrast to this, *synchronous languages* [10,3,4,9,11,25,26,23,8,18], offer a very attractive alternative for real-time verifi cation purposes of industrial applications. In particular, the usage of synchronous languages has important advantages for the analysis and verifi cation of the real-time behavior, since they have clean formal semantics. Furthermore they distinguish between micro and macro steps [12]: micro steps are statements that are executed in zero time (in the programmer's view), while a macro step consists of a fi nite number of micro steps and consumes a logical unit of time.

In [23], it has been shown how timed transition systems can be generated from synchronous programs for high-level real-time verifi cation, incorporating abstraction techniques that retain the quantitative temporal information. Nevertheless, 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. 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[1]. 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.

For this purpose, effi cient methods were introduced in [24]. There, an exact and detailed low-level (architecture-dependent) runtime analysis of synchronous programs was introduced, which computes the exact execution times of *all possible single transitions* of a system and simultaneously generates a real-time formal model, that can directly be used for architecture-dependent real-time verifi cation purposes.
The approaches presented in [23] and [24] allow the generation of real-time formal models directly out of synchronous languages. This enabled a bridging between industrial real-time descriptions and formal real-time verifi cation, i.e. a complete, formally verifi able design, development and realization of a real-time system out of an industrial real-time description, supporting formal verifi cation of a real-time system in earlier (high-level) as well as in later (low-level) design stages.

[1] 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.

The approaches presented in [23] and [24] consider *timed Kripke structures* as formal models, which were introduced in [21] for the modelling of atomic, non-interruptible timed actions, required by the macro-steps of an implementation of a synchronous program. These models allow also the use of abstractions without loss of quantitative properties.

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 in [21] as the first real-time extension of the temporal logic CTL that can handle atomic, non-interruptible timed transitions. JCTL allows the direct use of established symbolic verification techniques.

The algorithms for model-checking JCTL formulae presented in [21] are based on fixpoint iterations. The state space is traversed backwards 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. These traditional symbolic model checking techniques based on backward state traversal, consider a set of states where a property holds and collect further states by using a predecessor relation.

In contrast to this, there also exist forward state traversal techniques, which utilize a successor relation. Dependent on the application, either forward or backward state traversal yields in the best possible verification results.

In particular, in [16,17,14] was shown, that symbolic methods which utilize forward state traversal in the formal model yield faster processing time for some qualitative model-checking applications. Forward state traversal techniques are also used very successful in [15,7]. However, none of the above mentioned approaches can benefit from a directly usage of industrial description languages and must therefore take special input formats into account.

Nevertheless, such approaches have clearly shown that it is advantageous to support also forward state traversal techniques for model-checking purposes.

In this paper we therefore present efficient forward symbolic model-checking techniques for the verification of real-time systems. In particularly, we introduce forward traversing methods for the real-time temporal logic JCTL, in order to benefit from both, backward and forward state traversal methods and of course, their combination. In addition to this, we also introduce special methods that allow the algorithms for model-checking JCTL formulae to perform time-jumps during the state space traversal. In other words, the algorithms are able to perform less iterations than the length of the traversed paths (measured in time units). The algorithms traverse the state space by breadth-first search and are implemented in our model checking tool Equinox, using the CUDD BDD library [27]. Moreover, we give the complexity of the JCTL model checking algorithms. It turns out that the complexity to compute the set of states, where a given JCTL formula holds, is the same as for the backward state traversal approach ([22].

The outline of the paper is as follows: In chapter 2 we give an overview of the formalisms on which this paper is based. In section 3, we then proceed with the description of the forward, time-jumping symbolic model checking procedure for the real-time temporal logic JCTL. In section 4, we then conclude with preliminary experimental results obtained by our verifi cation tool Equinox.

2 Background

2.1 Timed Kripke Structures

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

Definition 1 (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.*

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 $t_0 < t < t_0 + k$.*

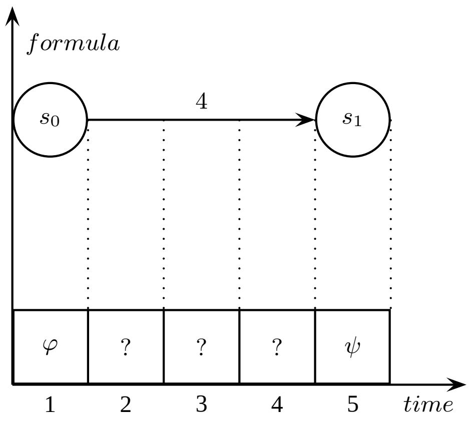


Fig. 1. A Timed Transition in a TKS

As example, consider Figure 1, 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, qualitative 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 (UDSs)*.

2.2 The Real-Time Temporal Logic JCTL

In this section we explain in detail the temporal logic JCTL, which is the first real-time extension of CTL, that is based on interpreting timed transition systems with interpretation I_J. JCTL is directly defined on timed Kripke structures and thus, is also able to handle transitions that correspond to the non-interruptible atomic timed actions of an implementation of a synchronous program.

For example, JCTL can handle 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 constrasct to other real-time 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 timed actions.

For the JCTL definition below, only a small subset of basic logical operators is required, which can be extended further by abbreviations.

Definition 2 (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*
- $\neg\varphi$, $\varphi \land \psi \in$ 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*.

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

Definition 3 (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 defined for a fixed 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 4 (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*
$$\left(a+1 \leq \tau_\pi^{(0)} \leq b\right) \wedge \left(\mathcal{K}, \pi^{(1)} \models \varphi\right)$$
- $\mathcal{K}, s \models \mathsf{E}\underline{\mathsf{X}}^{\geq a+1}\varphi$ *iff there is a path* $\pi \in \mathsf{Paths}_\mathcal{K}(s)$ *with associated duration function* τ_π *with*
$$\left(a+1 \leq \tau_\pi^{(0)}\right) \wedge \left(\mathcal{K}, \pi^{(1)} \models \varphi\right)$$
- $\mathcal{K}, s \models \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[a,b]}\ \psi]$ *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*
$$\left(a \leq \sum_{j=0}^{i-1} \tau_\pi^{(j)} \leq b\right) \wedge$$
$$\left(\mathcal{K}, \pi^{(i)} \models \psi\right) \wedge \left(\forall j < i.\ \mathcal{K}, \pi^{(j)} \models \varphi\right)$$
- $\mathcal{K}, s \models \mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\geq a}\ \psi]$ *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*
$$\left(a \leq \sum_{j=0}^{i-1} \tau_\pi^{(j)}\right) \wedge$$
$$\left(\mathcal{K}, \pi^{(i)} \models \psi\right) \wedge \left(\forall j < i.\ \mathcal{K}, \pi^{(j)} \models \varphi\right)$$
- $\mathcal{K}, s \models \mathsf{EG}^{[a,b]}\varphi$ *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*
$$\left(a \leq \sum_{j=0}^{i-1} \tau_\pi^{(j)} \leq b\right) \rightarrow \left(\mathcal{K}, \pi^{(i)} \models \varphi\right)$$
- $\mathcal{K}, s \models \mathsf{EG}^{\geq a}\varphi$ *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 satisfies 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 satisfies 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 first i states $\pi^{(0)}$, $\pi^{(1)}$, ..., $\pi^{(i-1)}$ 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 $\pi^{(i)}$ satisfies 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 first i states $\pi^{(0)}$, $\pi^{(1)}$, ..., $\pi^{(i-1)}$ 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 $\pi^{(i)}$ satisfies 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.

The set of the basic JCTL operators is complete for a CTL-like logic, i.e. that *further* JCTL *operators, like e.g. the formulae $\mathsf{A}\underline{\mathsf{X}}^{\kappa}\varphi$, $\mathsf{A}[\psi\ \underline{\mathsf{U}}^{\kappa}\ \varphi]$, and $\mathsf{AG}^{\kappa}\varphi$ can be defined in terms of $\mathsf{E}\underline{\mathsf{X}}^{\kappa}\varphi$, $\mathsf{E}[\psi\ \underline{\mathsf{U}}^{\kappa}\ \varphi]$, $\mathsf{EG}^{\kappa}\varphi$,* as well as qualitative-only CTL operators. Some examples for abbreviations are:

– $\mathsf{A}[\psi\ \underline{\mathsf{U}}^{[a,b]}\ \varphi] \equiv \mathsf{A}[\psi\ \underline{\mathsf{U}}^{\geq a}\ \varphi] \wedge \neg\mathsf{EG}^{[a,b]}\neg\psi$
– $\mathsf{A}[\psi\ \underline{\mathsf{U}}^{\geq a+1}\ \varphi] \equiv \mathsf{AG}^{\leq a}(\varphi \wedge \mathsf{AXA}[\psi\ \underline{\mathsf{U}}\ \varphi])$
– $\mathsf{AG}^{\kappa}\varphi \equiv \neg\mathsf{E}[\neg\varphi\ \underline{\mathsf{U}}^{\kappa}\ 1]$

As example for the JCTL semantics, consider the timed Kripke structure shown in Figure 2. There, we have:

– $\left[\!\!\left[\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\leq 9}\ \psi]\right]\!\!\right] = \{s_0, s_1, s_2, s_4, s_5\}$
– $\left[\!\!\left[\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\leq 6}\ \psi]\right]\!\!\right] = \{s_1, s_2, s_4, s_5\}$
– $\left[\!\!\left[\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[3,5]}\ \psi]\right]\!\!\right] = \{s_4\}$
– $\left[\!\!\left[\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\geq 9}\ \psi]\right]\!\!\right] = \{s_0\}$

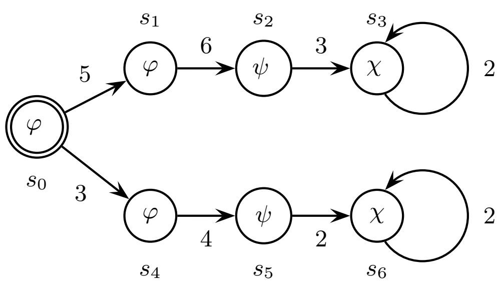


Fig. 2. An Example for the JCTL Semantics

- $\left[\!\left[\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{\geq 12}\ \psi]\right]\!\right] = \{\}$
- $\left[\!\left[\mathsf{EG}^{\leq 4}\varphi\right]\!\right] = \{s_0, s_1\}$
- $\left[\!\left[\mathsf{EG}^{[2,4]}\varphi\right]\!\right] = \{s_0, s_1\}$
- $\left[\!\left[\mathsf{EG}^{\geq 4}\varphi\right]\!\right] = \{\}$
- $\left[\!\left[\mathsf{E}\underline{\mathsf{X}}^{\leq 7}\varphi\right]\!\right] = \{s_0\}$
- $\left[\!\left[\mathsf{E}\underline{\mathsf{X}}^{\leq 4}\varphi\right]\!\right] = \{s_0\}$
- $\left[\!\left[\mathsf{E}\underline{\mathsf{X}}^{[4,8]}\varphi\right]\!\right] = \{s_0\}$
- $\left[\!\left[\mathsf{E}\underline{\mathsf{X}}^{\geq 2}\varphi\right]\!\right] = \{s_0\}$
- $\left[\!\left[\mathsf{E}\underline{\mathsf{X}}^{\geq 6}\varphi\right]\!\right] = \{\}$

Note that in the results of $\left[\!\left[\mathsf{EG}^{[2,4]}\varphi\right]\!\right]$ 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 definition of $\left[\!\left[\mathsf{EG}^{[a,b]}\varphi\right]\!\right]$, we have

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

which is always true.

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

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

In order to explain how time-jumps are performed by the algorithms presented in next section, we also explain here the notion of the *Tracks of a state*:

Definition 5 (Tracks of a State). *Given a TKS $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$, a state $s_\tau \in \mathcal{S}$, the set $\mathcal{R}_\tau = \{(s, t, s') \in \mathcal{R} \mid s = s_\tau\}$, representing the transitions leading from s_τ to all its successor states and $t_{max} = max\{t \in \mathbb{N} \mid \exists s_\tau \in \mathcal{S}.s' \in \mathcal{S}.(s_\tau, t, s') \in \mathcal{R}_\tau\}$, defined as the maximum duration of all transitions in $\mathcal{R}_\tau$. Then, we define $\mathsf{Track}_\mathcal{K}(s_\tau) = \big\{(s_\tau, t) \mid t \in \{1, \ldots, t_{max}\}\big\}$, as the set of all tracks of the state s_τ. Hence, a single track is a tuple (s, k), where $s \in \mathcal{S}, k \in \mathbb{N}$ and $k \leq t$, such that for $s' \in \mathcal{S}$, we have $(s, t, s') \in \mathcal{R}$.*

3 Forward Symbolic Model Checking

In this section, we present forward 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 directly applied to timed Kripke structures and traverse the state space by breadth-fi rst search. They are implemented in our model-checking tool Equinox, using the CUDD BDD library [27]. 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. In Equinox, the qualitative-only transition system (a UDS) is always given in advance – before the TKS construction, so there are no additional operations required in order to obtain it. Figure 3 shows an overview of the overall modular construction of Equinox and its design flow. Equinox consists of the BDD-based tool JERRY, which is used for model-checking and runtime analysis purposes and a compiler for the synchronous language Quartz and its extensions [25,23,24]. Having experimented with many BDD-tools available, we consider the CUDD-package [27] as a reliable BDD-package, offering a great number of useful features. Consequently all our tools use the CUDD-package for BDD-manipulation.

The algorithms for the basic JCTL operators $\mathsf{EG}^{[a,b]}\varphi$ and $\mathsf{E}[\varphi \ \underline{\mathsf{U}}^{[a,b]} \ \psi]$ are shown in Figures 5 and 6 respectively. The operators $\mathsf{E}\underline{\mathsf{X}}^{[a+1,b]}\varphi$ and $\mathsf{E}\underline{\mathsf{X}}^{\geq a+1}\varphi$ need no iterating algorithms, since they can be checked trivially by checking at the outgoing transitions if there is a path satisfying the time constraint and if the transitions are leading to a state where φ holds. The algorithms for the operators $\mathsf{EG}^{\geq a}\varphi$ and $\mathsf{E}[\varphi \ \underline{\mathsf{U}}^{\geq a} \ \psi]$ are only used in order to avoid a notion of intervals, which would contain infi nity. They can both be expressed as abbreviations of the basic JCTL operators $\mathsf{EG}^{[a,b]}\varphi$ and $\mathsf{E}[\varphi \ \underline{\mathsf{U}}^{[a,b]} \ \psi]$ and of known qualitative-only CTL operators:

- $\mathsf{EG}^{\geq a+1}\varphi \equiv \mathsf{EF}^{\leq a}\mathsf{EXEG}\varphi$
- $\mathsf{E}[\psi \ \underline{\mathsf{U}}^{\geq a+1} \ \varphi] \equiv \mathsf{EG}^{\leq a}(\varphi \wedge \mathsf{EXE}[\psi \ \underline{\mathsf{U}} \ \varphi])$

Of course, we also have

- $\mathsf{EG}^{\geq 0}\varphi \equiv \mathsf{EG}\varphi$
- $\mathsf{E}[\psi \ \underline{\mathsf{U}}^{\geq 0} \ \varphi] \equiv \mathsf{E}[\psi \ \underline{\mathsf{U}} \ \varphi]$

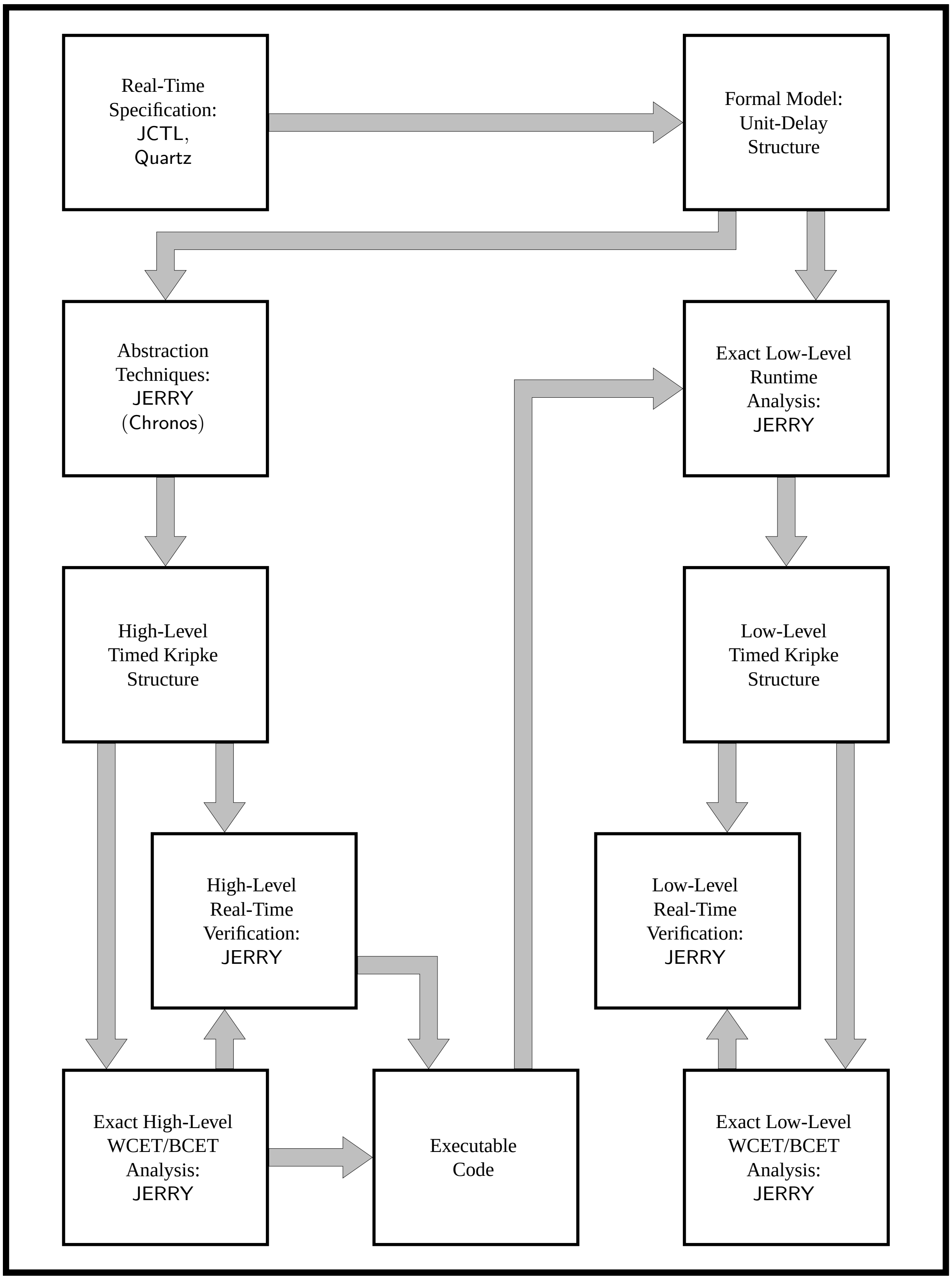
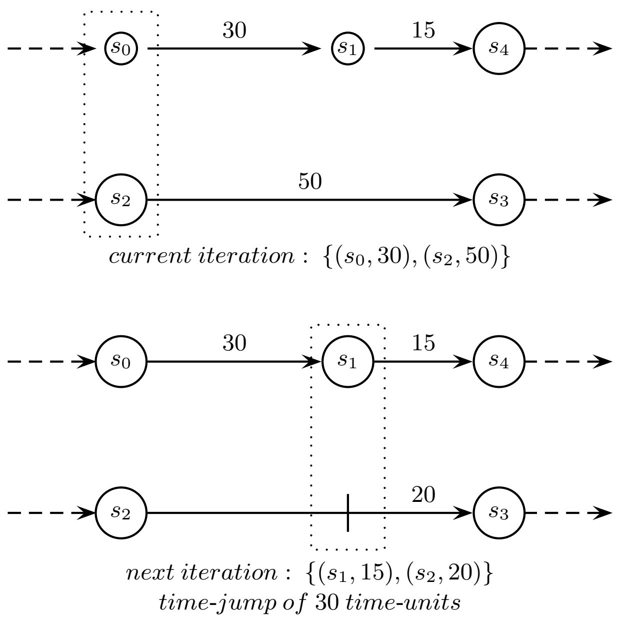

Real-Time Specification: JCTL, Quartz
Formal Model: Unit-Delay Structure
Abstraction Techniques: JERRY (Chronos)
Exact Low-Level Runtime Analysis: JERRY
High-Level Timed Kripke Structure
Low-Level Timed Kripke Structure
High-Level Real-Time Verification: JERRY
Low-Level Real-Time Verification: JERRY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

Fig. 3. Equinox: A Formal Framework for the Specification, Modelling, Verification and Runtime Analysis of Real-Time Systems



s_0
30
s_1
15
s_4
s_2
50
s_3
$current\ iteration:\ \{(s_0,30),(s_2,50)\}$
s_0
30
s_1
15
s_4
s_2
20
s_3
$next\ iteration:\ \{(s_1,15),(s_2,20)\}$
$time\text{-}jump\ of\ 30\ time\text{-}units$

Fig. 4. Time-Jumps in JCTL Forward Model-Checking

Before we consider the JCTL algorithms, we first explain the principle of time-jumps. According to the definition of a TKS, no qualitative information about the validity of a formula is given at the intermediate points of a transition. This allows the algorithms to jump through time, into the next reachable valid formula, i.e. into the closest (measured in time-units) state, or set of states. Figure 4 demonstrates this principle, performed by the presented JCTL algorithms. At a current iteration $\Im_k$ shown in the upper part of Fig. 4, the algorithm considers the set of states $\{s_0, s_2\}$. At this point, it determines the minimum time needed to reach the next valid formula, by computing the minimum time contained in the current tracks of the states $\{(s_0, 30), (s_2, 50)\}$. Hence, the next iteration $\Im_{k+1}$ shown in the lower part of Fig. 4 allows a time-jump of 30 time-units, which brings the computation into the next valid formula at state s_1. Now, the next minimum of the current set of tracks $\{(s_1, 15), (s_2, 20)\}$ can be considered for the next time jump.

The example shows clearly the strength of the JCTL model-checking procedure: The presented algorithms allow the logic to easily handle examples with huge timed transitions. The larger the duration of the minimum current track, the greater the time-jump will be.

The correctness of the function $\mathsf{EG}^{[a,b]}\varphi$ (Figure 5) can be seen as follows: The algorithm starts considering all transitions in $\mathcal{R}$ of the system and traverses forward through the state space, determining the successor transitions $\mathcal{R}_{succ}$ of $\mathcal{R}$ and collecting them in $\mathcal{R}_{run}$. This is done by extracting all transitions from the working set $\mathcal{R}_{run}$ which satisfy the condition that their transition time t_{min} is minimal.

The duration of all transitions from $\mathcal{R}_{run}$, which consume more than t_{min}, is decreased by t_{min} and stored in $\mathcal{R}_{gt}$, since no state will be reached by the time-jump from these transitions, but they will also move t_{min} time-units forward into time. The variable i equals the time steps we traversed forward in the transition relation. The set $\mathcal{R}_{new}$ discards the set of current states in a sequence of three sets of states: predecessors, current and successors. This done in order to be able to remove states which violate the property $\mathsf{EG}^{[a,b]}\varphi$ directly, without going back through the transition relation, i.e. without loosing their origin.

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 4). The algorithm terminates either when the time has reached the value b, or when the set of valid states has been emptied.

```
**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**
```

Fig. 5. Forward-Traversing Symbolic Algorithm for the $\mathsf{EG}^{[a,b]}\varphi$ Operator

The correctness of the function $\mathsf{E}[\varphi \ \underline{\mathsf{U}}^{[a,b]} \ \psi]$ (Figure 6) can be seen as follows:

The algorithm starts considering all transitions $\mathcal{R}$ of the system where φ holds and traverses forward through the state space determining the successor transitions $\mathcal{R}_{succ}$ of $\mathcal{R}$ where also φ holds and collecting them in $\mathcal{R}_{run}$. This is done in the exact manner as in the previous algorithm. In each loop iteration and if the time is within the interval $[a, b]$, all states of transitions that lead to some states ψ are collected, while their successor transitions are not being collected further since they already fulfi lled the property $\mathsf{E}[\varphi \ \underline{\mathsf{U}}^{[a,b]} \ \psi]$. When the time is outside the interval $[a, b]$, all transitions which lead to φ are collected. The algorithm terminates either when the time has reached the value b, or when the set of valid states has been emptied.

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

Fig. 6. Forward-Traversing Symbolic Algorithm for the $\mathsf{E}[\varphi \ \underline{\mathsf{U}}^{[a,b]} \ \psi]$ Operator

Furthermore, it is easily seen that if we consider the worst-case and assume that no time-jumps are possible for the model-checking procedure, the number of steps is multiplied with the maximum delay time $\hat{\tau}_{\mathcal{K}}$ that appears in $\mathcal{K}$. Hence, for the complexity of the forward-traversing JCTL we have the following

Theorem 1 (Complexity of forward-traversing JCTL). *For any TKS $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ and any JCTL formula φ, the functions $\mathsf{StatesEG}^{[a,b]}(\mathcal{S}_\varphi)$ and $\mathsf{StatesEU}^{[a,b]}(\mathcal{S}_\varphi, \mathcal{S}_\psi)$ given in Figures 5 and 6 run 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:

- $\mathsf{Time}(\mathsf{StatesEU}^{[a,b]}) \in O\big(k\hat{\tau}_{\mathcal{K}}(|\mathcal{R}| + |\mathcal{S}|)\big)$, where $k \leq b$
- $\mathsf{Time}(\mathsf{StatesEG}^{[a,b]}) \in O(k\hat{\tau}_{\mathcal{K}}(|\mathcal{R}| + |\mathcal{S}|))$, 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.

Hence, the forward traversing functions $\mathsf{StatesEG}^{[a,b]}(\mathcal{S}_\varphi)$ and $\mathsf{StatesEU}^{[a,b]}(\mathcal{S}_\varphi, \mathcal{S}_\psi)$ have the same complexity like the backward-traversing ones presented in [22]. The algorithms terminate when either the upper time bound of the interval is reached or the set $\mathcal{R}_{run}$ is empty. The variable i, representing the length of the currently processed paths, is incremented in every iteration and will reach the fi nite upper time bound.

4 Experimental Results

We have implemented the algorithms in our tool framework Equinox and have tested several benchmarks. In this section, we present our experimental results that we have obtained with a benchmark, which is very popular in the world of real-time formal verifi cation, Fischer's mutual exclusion protocol [20].

Figure 7 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:

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*
$s_5$ :      $\lambda := 0$;
```

Fig. 7. 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. [20]). Nevertheless, Fischer's mutex protocol is an excellent example which has been widely considered by many researchers as a benchmark.

High-Level Verification						
			EU		EG	
n	vars	BDD nodes	time [sec]	gain [%]	time [sec]	gain [%]
5	36	666	0.63	0	0.66	0
10	67	2299	6.56	0	4.46	0
15	97	4097	40.24	0	30.51	0
20	128	7036	55.57	0	65.06	0
25	158	8256	117.49	0	97.45	0
30	188	17061	262.60	0	159.43	0

Table 1. Fischer's High-Level Verification

The results for the $\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[a,b]}\ \psi]$ algorithm were performed on a specifi cation which verifi es, that all processes have fi nished within some given time constraint. The results for the $\mathsf{EG}^{[a,b]}\varphi$ algorithm were performed on a specifi cation which verifi es, that all processes entered their critical sections in the time from start to their maximal execution time.

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, which means that the system has 2^{var} reachable states. Column three shows memory consumption, expressed in required BDD nodes. Columns four and six show the determined runtimes for model checking with the operators $\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[a,b]}\ \psi]$ and $\mathsf{EG}^{[a,b]}\varphi$

Low-Level Verification						
			EU		EG	
n	vars	BDD nodes	time [sec]	gain [%]	time [sec]	gain [%]
2	17	317	0.07	3.70	0.04	3.10
3	24	441	0.43	45.95	0.33	12.5
4	32	581	0.71	43.84	0.59	24.44
5	37	776	0.92	73.86	0.69	22.38
6	44	1167	3.44	39.62	1.54	7.41
7	50	1635	3.25	50.51	2.54	13.79

Table 2. Fischer's Low-Level Verification

respectively. Columns five and seven denote the gain on iterations when using time-jumps, expressed as the ratio of iterations of the time-jump procedure, divided by the iterations of a normal, single time step procedure.

Note that the modular design of Equinox allows us to isolate the execution times for performing the verification only. The construction of the model is performed in advance and not added to the runtimes.

Table 1 shows a worst-case example, where we consider only untimed transitions, in order to avoid time-jumps – so we have 0% gain for all tests. The results clearly demonstrate that our verification tool Equinox and the presented forward model-checking algorithms have the ability to handle very large systems. The symbolic breadth-first search traversing of the algorithms, makes it possible to easily handle a system with 30 Fischer processes, which requires 188 boolean variables, i.e. it includes 2^{188} reachable states.

To our knowledge, no other real-time verification tool has ever succeeded in verifying Fischer's mutex protocol for more than 11 processes, which is well known as a very difficult benchmark. However, here is one (special) exception: In [19], an approach was developed especially for the verification of Fischer's mutex protocol, that allowed the verification of 50 processes by the tool CMC. But this is an approach, which was developed especially for this protocol. Equinox is able to handle 30 processes without any of these special techniques.

Furthermore, it is of course possible to obtain much better results by transforming the benchmarks, using the methods presented in [23], avoiding the worst-case of no time-jumps.

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

For the low-level verifi cation results shown in Table 2, an exact runtime analysis of the benchmark for the appropriate architecture is necessary in advance [24]. We have performed this on an Pentium III 1GHz to get the appropiate timing information for the low-level model. Here, the original system is endowed by additional physical times, required for the code execution of the synchronous program. This results in an enormous increase of the system's complexity by a formidable time-factor.

Nevertheless, Table 2 clearly demonstrates that this increased complexity effects mainly only the model construction, which, due to the many BDD operations is a time-consuming task, while the model-checking algorithms show a similar behaviour, like the one of the high-level verifi cation.

Clearly, the duration of the timed transitions increases with an increasing number of processes, i.e. the target machine needs longer execution times. Usually one would expect that the time-jump gain should also increase with an increasing duration of timed transitions, as it is the case for 2,3,4 and 5 processes. Nevertheless, the examples of 6 and 7 processes show that the gain achieved by the time-jumps depends also on the resulted grade of the time's granulation at the timed transitions. As can be seen, this effects both algorithms, the $\mathsf{EG}^{[a,b]}\varphi$ and the $\mathsf{E}[\varphi\ \underline{\mathsf{U}}^{[a,b]}\ \psi]$.

For example, the set of tracks $\{(s_0, 300), (s_1, 40)\}$ will result in a time-jump of 40 time-units, while the set $\{(s_2, 70), (s_3, 80)\}$ will result in a greater time-jump of 70 time-units, altough the state s_0 has a longer transition of 300 time-units.

References

1. R. Alur, C. Courcoubetis, and D. Dill. Model Checking in Dense Real-time. Technical report, Stanford University, University of Crete, 1991.
2. J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL in 1995. In *Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes In Computer Science, pages 431–434. Springer-Verlag, March 1996.
3. G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.
4. G. Berry. The Esterel v5_91 language primer. http://www.esterel.org, June 2000.
5. S. Campos and E. Clarke. Real-Time Symbolic Model Checking for Discrete Time Models. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development*, AMAST Series in Computing. World Scientific Press, AMAST Series in Computing, May 1994.
6. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III*, volume 1066 of *LNCS*. Springer, 1996.
7. D. L. Dill. The murphi verification system. In R. Alur and T. A. Henzinger, editors, *Conference on Computer Aided Verifi cation (CAV)*, volume 1102 of *LNCS*, pages 390–393, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.
8. ECL Homepage. Website. http://www-cad.eecs.berkeley.edu/
9. Esterel. Website. http://www.esterel.org.
10. N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, 1993.
11. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, sep 1991.

12. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engenieering Methods*, 5(4), 1996.
13. T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 394–406, Santa-Cruz, California, June 1992. IEEE Computer Society Press.
14. T. A. Henzinger, O. Kupferman, and S. Qadeer. From pre-historic to post-modern symbolic model checking. *Form. Methods Syst. Des.*, 23(3):303–327, 2003.
15. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
16. H. Iwashita and T. Nakata. Forward model checking techniques oriented to buggy designs. In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 400–404. IEEE Computer Society, 1997.
17. H. Iwashita, T. Nakata, and F. Hirose. Ctl model checking based on forward state traversal. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 82–87. IEEE Computer Society, 1996.
18. Jester Home Page. Website. www.parades.rm.cnr.it/projects/projects.html.
19. K. Kristoffersen, F. Laroussinie, K. Larsen, P. Pettersson, and W. Yi. A compositional proof of a real-time mutual exclusion protocol. In *In Proc. of the 7th International Joint Conference on the Theory and Practice of Software Development*, 1997.
20. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 1987.
21. G. Logothetis and K. Schneider. A new approach to the specification and verification of real-time systems. In *Euromicro Conference on Real-Time Systems*, pages 171–180, Delft, The Netherlands, June 2001. IEEE Computer Society. http://goethe.ira.uka.de/fmg/ps/LoSc01.ps.gz.
22. G. Logothetis and K. Schneider. Symbolic model checking of real-time systems. In *International Symposium on Temporal Representation and Reasoning*, pages 214–223, Cividale del Friuli, Italy, June 2001. IEEE/ACM. http://goethe.ira.uka.de/fmg/ps/LoSc01a.ps.gz.
23. G. Logothetis and K. Schneider. Extending synchronous languages for generating abstract real-time models. In *European Conference on Design, Automation and Test in Europe (DATE)*, pages 795–802, Paris, France, March 2002. IEEE Computer Society. http://goethe.ira.uka.de/fmg/ps/LoSc02.ps.gz.
24. G. Logothetis, K. Schneider, and C. Metzler. Generating formal models for real-time verification by exact low-level analysis of synchronous programs. In *24th IEEE International Real-Time Systems Symposium (RTSS)*, pages 256–265, Cancun, Mexico, December 2003. IEEE Computer Society Press. http://goethe.ira.uka.de/fmg/ps/LoSM03c.ps.gz.
25. K. Schneider. A verified hardware synthesis for Esterel. In F. Rammig, editor, *International IFIP Workshop on Distributed and Parallel Embedded Systems*, pages 205–214, Schloß Ehringerfeld, Germany, 2000. Kluwer Academic Publishers.
26. K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *International Conference on Application of Concurrency to System Design (ICACSD 2001)*, pages 143–156, Newcastle upon Tyne, UK, June 2001. IEEE Computer Society Press. http://goethe.ira.uka.de/fmg/ps/Schn01a.ps.gz.
27. F. Somenzi. CUDD: CU decision diagram package, release 2.3.0, 1998. ftp://vlsi.colorado.edu/pub/ and http://vlsi.Colorado.EDU/.