Exact Low-Level Runtime Analysis of Synchronous Programs for Formal Verification of Real-Time Systems

G. Logothetis¹, K. Schneider² and C. Metzler¹

¹ 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

² University of Kaiserslautern
Department of Computer Science
Reactive Systems Group
P.O. Box 3049, 67653 Kaiserslautern, Germany
Klaus.Schneider@informatik.uni-kl.de

Abstract. Synchronous programming languages are well-suited for the implementation and verification of real-time systems. We describe a new approach to determine the execution times of synchronous programs wrt. given microprocessors and generate real-time formal models endowed with notions of physical time for formal verification purposes by means of model-checking. The execution times are exact, since we consider all possible input sequences. In order to benefit from the native mathematical operators of the target CPU, we avoid the conversion of these operators to their logical equivalents, usually performed by other model-checking algorithms. For this purpose, we introduce a new technique that constructs a transition system which preserves mathematical operations as atomic actions.

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. For real-time systems, the essential task is to guarantee that certain actions are executed within some strict runtime limits or that they will start only after some point of time. Decreasing time-to-market and the overall design costs requires to check as early as possible in the design flow that these runtime constraints are met. For this purpose, several approaches to the verification of real-time systems have been developed [1, 18, 9, 2, 10, 22] that are based on different formalisms for describing finite state transition systems endowed with some notion of time.

Real-time systems are usually described in languages like VHDL, Verilog, Esterel, C, C++, SystemC, etc. In particular, the usage of synchronous languages like Esterel [5, 12] has important advantages for the analysis and verification of the real-time behavior, for the following reasons:
• 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 pre-emption and suspension.
• Synchronous languages have clean formal semantics. In particular, the definition of control flow predicates in [30] supports static runtime analysis. Moreover, the compilation of synchronous languages leads to a transition system which can be used for formal verification purposes. There are already tools for verifying synchronous programs by symbolic model checking [30, 24].
• Synchronous languages distinguish between micro and macro steps [16]. Micro steps are statements that are executed within zero time (in the programmer’s model). A macro step consists of a finite number of micro steps and consumes a logical unit of time after the execution of its micro steps. Consequently, all threads run in lockstep and automatically synchronize at each macro step. The important fact for formal verification is that the languages are designed in such a way that macro steps can not contain loops; in other words: loop bodies must necessarily consist of macro steps.

In [24], it has already been shown how high-level real-time models can be generated out of synchronous programs for real-time verification purposes. 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. 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. Hence, for low-level real-time verification purposes, one must be able to consider the execution times of a program, in other words, the physical times required by the micro steps.

A popular method for determining execution times of a given program is worst case execution time (WCET) analysis [28]. Low-level analysis is done on the object code, and hence depends on the chosen hardware/software partitioning and the chosen architecture (micro controllers). 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 [17, 13, 26, 27].

Unfortunately, considering worst- or best-case execution times only, based on WCET analysis is not advisable for real-time verification purposes, since this would yield in wrong results. To be able to reason about real-time properties of a system (for instance, using real-time temporal logics like the one presented in [23]), 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 [17] 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 this paper, we therefore present a new technique that performs an exact and detailed low-level (architecture-dependent) runtime analysis of synchronous programs. Our approach 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 verification purposes. In particular, we present a new technique for code generation which enables an exact low-level runtime analysis of synchronous programs. After that, an efficient method is introduced to perform the analysis and determine the execution times required for the actions of the code. These execution times are in parallel used to construct the formal model, whose transitions are labelled with notions of physical time, denoting the exact execution times of the macro steps that correspond to the transitions.

Model-checking algorithms used for formal verification 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 transition system which preserves mathematical operations as atomic actions. After that, the transitions can be easily labelled by their physical execution times required by the target CPU.

Our approach takes advantage of established symbolic techniques to efficiently manipulate large finite state transition systems by means of binary decision diagrams (BDDs) [8]. We consider timed Kripke structures (TKSs) as formal models as proposed in [22]. Note that the obtained TKSs have timed transitions that correspond to non-interruptible atomic actions. They can be considered for real-time verification, using the real-time temporal logic JCTL [22], as well as for further analysis purposes like WCET and BCET, using techniques like the ones presented in [25].

The distinction between micro and macro steps 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 runtime analysis is therefore that compilers do automatically determine the building blocks in terms of macro steps that can then be used for runtime analysis.

There is not much related work. In [32], an extension of the synchronous language Esterel has been presented that focuses on the runtime verification 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. A more general approach has recently been presented in [6]. There, Esterel programs are endowed with pragmas that contain the quantitative information of the runtime of particular steps. This has no effect on code generation, but allows the gen-
eration of timed automata [1] for verification of temporal properties. However, this approach requires a low-level worst-case execution time (WCET) analysis in advance to obtain the real-time constraints. In other words, [6] can only refer to execution times concerning WCET analysis results about longest execution paths and can not consider exact execution times. This reduces significantly the accuracy of the verification results obtained by [6].

The outline of this paper is as follows: In the next section, we describe some theoretical background in more detail. In section 3, we then describe the exact low-level runtime analysis and the construction of formal models that preserve mathematical operations as atomic actions. We then conclude with preliminary experimental results.

2 Background

2.1 Synchronous Languages

Synchronous languages [15] like many Esterel-variants [3, 5, 11, 20, 30, 29] are becoming more and more attractive for the design and the verification of reactive real-time systems. These languages have a discrete model of time, i.e. time is modelled by natural numbers \( \mathbb{N} \). The execution of a synchronous program from one point of time \( t \) to \( t + 1 \) is called a macro step and involves the execution of several, but always finitely many, micro steps\(^1\). Hence, the execution of micro steps does not take time (in the programer’s model), and the execution of a macro step requires always the same amount of a logical time (in the programmer’s model). Consumption of time, i.e., the beginning of a new macro step, must be explicitly programmed with special statements like the \texttt{pause} statement in Esterel.

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 \( \mathbb{N} \rightarrow \alpha \) for some type \( \alpha \). 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 [4]). An important matter of fact for runtime analysis is that by the semantics of synchronous languages, there will be only finitely many micro steps in a macro step.

The entire semantics of a synchronous program \( P \) can be given as a finite state transition system \( A_P \): the states of \( A_P \) 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 satisfied. Execution of a transition will then invoke some manipulations of the data values. Hence, the semantics can be represented by a finite state control flow that interacts with a data flow of finitely many variables of possibly infinite data types.

\(^1\) It is important to note here 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.
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 odd(y) then
       next(c) := c + x
     end;
     next(x) := 2 \cdot x;
     next(y) := y/2;
   end while
 end loop
end module

(y \neq 0)/\{(odd(y), next(c) := c + x),
(1, next(x) := 2 \cdot x),
(1, next(y) := y/2)\}

Fig. 1. Russian Multiplication

For example, consider the Quartz program given in Figure 1 (it implements a Russian multiplication algorithm). The semantics is the transition system given in Figure 1. The three states correspond with the situations where the control flow is either outside the program or at one of the locations labelled with \ell 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 \Phi holds at that point of time. Taking the transition means that those assignments or signal emissions \alpha_i are executed whose guard \gamma_i holds at that point of time.

Beneath the comfortable programmer’s model given by the macro step abstraction, synchronous languages like the Esterel-variant 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. For more details, the reader is referred to [30, 29] and to the Esterel primer, which is an excellent introduction to synchronous programming [5].

2.2 Timed Kripke Structures (TKSs)

To model real-time systems we briefly introduce in this section timed Kripke structures (TKS) as proposed in [22]. Formally, a TKS over some set of variables \mathcal{V} is defined as follows:

Definition 1 (Timed Kripke Structure (TKS)). A timed Kripke structure over the variables \mathcal{V} is a tuple \((I, S, R, L)\), such that \(S\) is a finite set of states, \(I \subseteq S\) is the set of initial states, and \(R \subseteq S \times \mathbb{N} \times S\) is the set of transitions. For any state \(s \in S\), 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 modelled by a TKS. In the sense of [22], we use the following interpretation: A transition from state \( s \) to state \( s' \) with label \( k \in \mathbb{N} \) means that at anytime \( t_0 \), 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 \( t_0 < t < t_0 + k \).

The translation from a finite state representation like the one given in figure 1 is possible when only finite data types occur in the program. In a first step, however, the transitions are not labelled and simply correspond to transitions of the automaton. In the following, we call such a special case of a timed Kripke structure a unit delay structure (UDS), since we may assume that each transition is labelled with the time consumption 1. In the next section, we describe how we determine runtimes to label the transitions of such an intermediate structure to finally obtain a timed Kripke structure.

3 Low-Level Runtime Analysis

In this section we present a new technique in order to perform exact low-level runtime analysis and construct a timed Kripke structure from the automaton representation obtained from a synchronous program [30].

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 verified at a logical level.

For this purpose, we first construct a transition system (as UDS) and obtain executable code for the given synchronous program. 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 labelling 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 efficiently manipulate large finite state transition systems by means of binary decision diagrams (BDDs) [8].

Note further that the generated transition systems have timed transitions that correspond to non-interruptible atomic actions. For verification purposes of such systems, the real-time temporal logic JCTL [23] can be considered.

Our method to generate code for synchronous programs considers equation systems based on hardware synthesis, i.e. the method is based on the encoding of the states with
Boolean state variables. For example, the automaton of Fig. 1 yields the following state transition equations:

\[
\begin{align*}
\text{next}(\text{rdy}) & := (\neg \text{rdy} \land \neg \ell \land \text{st} \lor \\
& \quad (\text{rdy} \land (\neg \text{req} \lor (y = 0)) \lor \\
& \quad \ell \land (y = 0)) \\
\text{next}(\ell) & := (\text{rdy} \land \text{req} \lor \ell) \land (y \neq 0) \\
\text{next}(x) & := \left( \text{if } \ell \land (y = 0) \land \text{odd}(y) \text{ then } 2 \cdot x \right) \\
\text{next}(y) & := \ldots \\
\text{next}(c) & := \ldots
\end{align*}
\]

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 [7, 4], 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.

3.1 Considering native CPU-instructions for mathematical functions

Model-checking algorithms used for formal verification 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 transition system (ALU Kripke-structure) out of the boolean Kripke-structure, which preserves mathematical operations as atomic actions. After that, the transitions can be easily labelled by their physical execution times required by the target CPU.

It can clearly be seen, that such a Kripke-structure must have equivalent corresponding transitions to the boolean one, so that the formal verification by means of model-checking can still be performed. For simple mathematical functions which can be expressed by combinations of boolean logic, like additions, subtractions or boolean functions for n-bit integers, this holds trivially, since these operators represent in both models atomic transitions. More interesting are mathematical operators which can’t be stated as atomic transitions at the logical level, like a division of some n-bit integers which is mapped as a sequential calculation of subtractions and comparisons, realized by boolean operators. Such a calculation represents a set of states in the control-flow and consumes logical time. On a microprocessor on the other hand, this is an atomic
operation which consumes physical time.

As an example, Figure 2 shows a sequential division-algorithm for two 4-Bit Integers. For this example we reduced the algorithm to the division of two source numbers by two dividers (so we have four possible paths through the Kripke-structure). The state \( \text{init} \) leads to states \( c_i, i \in \{0, 1, 2, 3\} \), representing the possible start configuration of source number and divider. The states \( s_{i,j}, i \in \{0, 1, 2, 3\}, j \in \{0, 1, 2\} \) represent the steps of the division-algorithm, and lead to the corresponding \( \text{rdy}_i, i \in \{0, 1, 2, 3\} \), where the result is calculated.

![Diagram of sequential division for four 4-Bit Integers]

**Fig. 2.** sequential division for four 4-Bit Integers

Figure 3 shows the resulting ALU Kripke-structure, where the sequential steps are transformed into atomic transitions that correspond to atomic instructions using native CPU-instructions. The key idea for the generation of the ALU Kripke-structure is to transform the transition sequences that represent sequential calculations in the boolean Kripke-structure into atomic transitions. For this purpose, we first use the techniques presented in [24] to declare the locations of a synchronous program representing arithmetic operations as abstract ones. The result of the compilation leads then into a Kripke-structure, where all states involved in arithmetic operations are marked to be abstract.

After that, we must determine all paths of marked states that are allowed to be transformed into atomic transitions. There are two main factors that must be considered here:

Fig. 3. division for four 4-Bit Integers preserving arithmetic operations

- the termination of the arithmetic operations
- the parallel execution of processes

The first point is trivial to understand i.e., the existence of a suspend-statement pausing the calculation of a sequential operation dependent on external inputs leads to endless paths in the formal model for this operation. Furthermore, if abort statements are used, the operation in the boolean Kripke-structure can be interrupted before the end of the calculation. This will not happen in the CPU, since the instruction is an atomic one. In other words, we must check that all computation paths starting at the beginning of an arithmetic operation will finally reach the end of the operation. This is easily done by the checking the following CTL formula:

$$\forall G(S_{\text{start}} \rightarrow AF S_{\text{end}})$$

Regarding the parallel execution of processes, we must consider the following: if an ALU operation (requiring more than one unit of time in the programmer’s view) is running in parallel to some other operations, the states of an ALU Kripke-structure will not be synchronized anymore (with respect to the boolean Kripke-structure), if we transform the sequence of states representing the ALU operation into an atomic transition.

In other words, we must check that the ALU operation runs exclusively in the synchronous program, so that the transformation into atomic transitions will not lead to synchronization problems. For this purpose we first introduce new sets defined over the variables of the structure:

- $S_{\text{atomic}_{\text{div}}}$: These sets represent the control-flow-states of the different ALU operations, included in paths which should be transformed into atomic transitions. For example, for several divisions (div) and square root calculations (sqrt), which have equivalent atomic instructions on the target CPU, we define the sets $S_{\text{atomic}_{\text{div}}}$ and
\( \mathcal{S}_{\text{atomic,ort}} \) which represent the sets, where the control-flow is inside the arithmetic operation.

- \( \mathcal{S}_{\text{ALU}} \): This set represents the control-flow-states of all ALU operations which should be atomized: \( \mathcal{S}_{\text{ALU}} := \bigcup_{j=0}^{j} \mathcal{S}_{\text{atomic},j} \).

Hence, to identify the set of states of the structure that can be transformed into atomic transitions, we must first isolate the set of states, where some processes are executed in parallel to the ALU operations. This set is easily determined by

\[
\mathcal{S}_{||} := (\mathcal{S}_{\text{reach}} \cap \mathcal{S}_{\text{atomic},i}) \cap \neg \mathcal{S}_{\text{ALU}}
\]

After that, all pre- and post-states in \( \mathcal{S}_{\text{atomic},i} \) reachable from \( \mathcal{S}_{||} \) must be determined. This will identify all computation paths that can not be transformed into atomic transitions. It is performed by the sequence algorithm shown in Figure 4.

```
function sequence(\( \mathcal{U}, \mathcal{S}_{||}, \mathcal{S}_{\text{atomic},i} \))
    \( \mathcal{S}_0 := \mathcal{S}_{||} \)
    do
        \( \mathcal{S}_{\text{old}} := \mathcal{S}_0 \)
        \( \mathcal{S}_0 := \mathcal{S}_0 \cup \{ s \in \mathcal{S} \mid \exists s' \in \mathcal{S}_{\text{old}}, (s, s') \in \mathcal{U} \land s \in \mathcal{S}_{\text{atomic},i} \} \)
        \( \mathcal{S}_0 := \mathcal{S}_0 \cup \{ s' \in \mathcal{S} \mid \exists s \in \mathcal{S}_0, (s, s') \in \mathcal{U} \land s' \in \mathcal{S}_{\text{atomic},i} \} \)
        while \( \mathcal{S}_0 \neq \mathcal{S}_{\text{old}} \)
        return \( \mathcal{S}_0 \)
    end function
```

```
function atomize(\( \mathcal{U}, \mathcal{S}_{\text{atomic},1}, \ldots, \mathcal{S}_{\text{atomic},j} \))
    \( \mathcal{S}_{\text{cut}} = \{ \} \)
    forall \( i := 1 \) to \( j \) do
        \( \mathcal{S}_{||} := (\mathcal{S}_{\text{reach}} \cap \mathcal{S}_{\text{atomic},i}) \cap \neg \mathcal{S}_{\text{ALU}} \)
        \( \mathcal{S}_{\text{seq}} := \text{sequence}(\mathcal{U}, \mathcal{S}_{||}, \mathcal{S}_{\text{atomic},i}) \)
        \( \mathcal{S}_{\text{cut}} := \mathcal{S}_{\text{cut}} \cup (\mathcal{S}_{\text{atomic},i} \setminus \mathcal{S}_{\text{seq}}) \)
    end
    \( \mathcal{U} := \text{ShortCut}(\mathcal{U}, \mathcal{S}_{\text{cut}}) \)
    return \( \mathcal{U} \)
end function
```

Fig. 4. Transforming computation paths of arithmetic operations into atomic transitions

As an example, consider the control-flow of a simple program, shown in Figure 5. In this example, all \( a_i \) represent two ALU operations. \( a_i, i \in \{0, 1, 2\} \) is the control-flow of operation 1 and \( a_i, i \in \{3, 4, 5\} \) is the control-flow of operation 2. The \( b_i \) represent arbitrary states, which do not correspond to an atomic ALU operation. As we can see, operation 1 (\( a_i, i \in \{0, 1, 2\} \)) can be transformed into an atomic transition, while operation 2 (\( a_i, i \in \{3, 4, 5\} \)) cannot. With the given formula for \( \mathcal{S}_{||} \) we get \( \mathcal{S}_{||} = \{b2\} \).

Applying the sequence algorithm for \( b2 \) returns the set \( a_i, i \in \{3, 4, 5\} \).
Finally, the resulting transition relation is constructed by the atomize algorithm shown in Figure 4. Please note that on some platforms the time consumption of arithmetic operations depends on the operands. The quantitative information of these instructions has to be determined for all possible operands, if there is a variance of needed clock-cycles on a specific platforms. atomize uses the algorithm ShortCut shown in Figure 6 to transform the selected paths into atomic transitions.

```plaintext
function ShortCut(\mathcal{U}, S_\varphi)
\mathcal{U}_{\text{short}} := \{(s, s') \in \mathcal{U} \mid \{s, s'\} \cap S_\varphi = \{\}\};
\mathcal{U}_{\text{out}} := \{(s, s') \in \mathcal{U} \mid s \in S_\varphi \land s' \not\in S_\varphi\};
repeat
\mathcal{U}_0 := \{(s, s') \mid \exists s_1. (s, s_1) \in \mathcal{U} \land (s_1, s') \in \mathcal{U}_{\text{out}}\};
\mathcal{U}_{\text{short}} := \mathcal{U}_{\text{short}} \cup \{(s, s') \mid (s, s') \in \mathcal{U}_0 \land s \not\in S_\varphi\};
\mathcal{U}_{\text{out}} := \{(s, s') \in \mathcal{U}_0 \mid s \in S_\varphi\};
d until \mathcal{U}_{\text{out}} = \{\};
return \mathcal{U}_{\text{short}};
end function
```

Fig. 6. ShortCut Algorithm

Now, consider how ShortCut works. Recall, that we may assume that all computation paths starting at the beginning of an ALU operation 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 to be abstract. To compute the further transitions, we have to first compute the transitions $\mathcal{U}_{\text{out}}$ that exit from a marked sequence. In the following loop, the algorithm performs a short cut of all predecessor transitions of $\mathcal{U}_{\text{out}}$. It is easily seen that the loop in ShortCut will be repeated at most $d_{\text{len}}$ times, where $d_{\text{len}}$ is the maximal length of a finite sequence of states in a computation path representing an arithmetic operation. An example for the execution of ShortCut is given in Figure 7.

### 3.2 Generating timed Kripke-structures

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 [17] in order to handle this problem. Techniques like the ones presented in [17] can be easily endowed in our tool and are part of our current implementation work.

We assume that we already have a function QuartzCompileUDS for the code generation, that computes an equivalent unit delay structure (UDS) $K_{\mathcal{U}}$ of a given Quartz program $\mathcal{P}$. The states of this structure correspond with the states of the program $\mathcal{P}$ and are labelled with Boolean variables. Such a function is essentially implemented by any compiler, like the ones described in [29, 30].

To finally obtain an equivalent TKS $K_{\mathcal{K}}$, 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. Remember that we need to process the UDS before the code generation to identify which of the sequential arithmetic algorithms can be translated into native arithmetic CPU code.

We first use a function QuartzCompileC which translates the obtained UDS $K_{\mathcal{U}}$ into C-Code according to the method described above. The TKS is constructed by the algorithms given in figure 8. To explain these algorithms, we first want to emphasize,
function QuartzCompileTKS(P)
(I, S, U) := QuartzCompileUDS(P);
C := QuartzCompileC(U);
R := \{\};
while U \neq \{\} do
  S_{ia} := \{s \in S | \exists s' \in S, (s, s') \in U \land L_{ia}(s) \neq false\};
  s_{time} := choose any of S_{ia};
  time := RuntimeC(s_{time}, C);
  U_{time} := \{(s, s') \in U | \exists s, s' \in S, L_{ia}(s) = L_{ia}(s_{time})\};
  U := U \setminus U_{time};
  R := R \cup \{U_{time} \times \{time\}\};
end;
return R;
end function

function RuntimeC(s, C)
  time := execution time(C(s));
  return time;
end function

Fig. 8. Runtime analysis and TKS construction

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 $|S|^2$ transitions.

Fig. 9. Applying physical times on Kripke structures
Recall that the transition system is derived from a synchronous program. Such a transition system contains boolean operators and arithmetic instructions. On an ALU boolean operations consume identical amount of time regardless of the values of their operands. A special case of boolean operators which needs to be considered here are If-Then-Else-statements. These are handled by microprocessors by means of jump-instructions. Their runtimes therefore depend on the configuration of their arguments. In addition to this, on some architectures the runtime of arithmetic instructions also depends on their arguments.

As an example consider the If-Then-Else-statement shown in Fig. 9. The two transitions refer to the statement \( \text{next}(a) = b?c:d \) and consume different times \( t_1 \) and \( t_2 \). On the other hand the statement \( \text{next}(a) = b \lor c \) of Fig. 9 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 same effect can be noticed with arithmetic instructions on some architectures.

The key idea is to use an efficient technique to obtain the variables occurring inside of arguments of If-Then-Else and arithmetic operators and determine the execution times for performing transitions with valid configurations of these variables. For this purpose the set of variables is given by

\[
\mathcal{V} := \mathcal{V}_{ia} \cup \mathcal{V}_{nonia}
\]

The set \( \mathcal{V}_{ia} \) consists of the variables which occur inside the arguments of if-statements or arithmetic instructions, while the set \( \mathcal{V}_{nonia} \) consists of the variables which occur outside the arguments of if-statements and arithmetic instructions. We assign the labelling functions \( \mathcal{L}_{ia} \) and \( \mathcal{L}_{nonia} \) to \( \mathcal{V}_{ia} \) and \( \mathcal{V}_{nonia} \) respectively, so that

\[
\mathcal{L}(s) = \mathcal{L}_{ia}(s) \land \mathcal{L}_{nonia}(s), \quad \forall s \in S.
\]

Hence, if \( \mathcal{V}_{ia} = \{\} \) then it follows that \( \mathcal{L}_{ia}(s) = \text{true}, \forall s \in S \) and also if \( \mathcal{V}_{nonia} = \{\} \) then it follows that \( \mathcal{L}_{nonia}(s) = \text{true}, \forall s \in S \). In the following, we will consider the main algorithm \texttt{QuartzCompileTKS} given in Fig. 8 more in detail.

Our goal 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.

For this purpose, our method takes advantage from the fact that boolean operations consume identical amount of time regardless of the values of their operands, i.e. regardless of the configuration of the variables \( \mathcal{V}_{nonia} \).

Hence, \texttt{QuartzCompileTKS} must consider all different variable configurations of \( \alpha \in \mathcal{V}_{ia} \), but only one single configuration \( \beta \in \mathcal{V}_{nonia} \) and this only under the condition that there exist a reachable state meeting such a configuration, i.e. \( \exists s \in S_{\text{reach}}, \mathcal{L}(s) = \alpha \land \beta \), where \( S_{\text{reach}} \subseteq S \) is the set of reachable states. If there is no such \( \beta \), 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 configuration of $V_{ia} \land V_{nonia}$, the set of all transitions $U_{time}$ that are represented by all possible variable configurations of $V_{nonia}$ are labelled with the physical time $t$. The function $\text{RuntimeC}$ determines the runtime of the generated code for a given set of transitions. $\text{RuntimeC}$ performs dynamic runtime measurements for given configurations of $V_{ia} \land V_{nonia}$ by means of so-called profiling tools like gprof [14] or VTune [19].

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

The algorithm terminates when all states of $U$ were considered in the runtime analysis, i.e. $U = \{\}$. Note that if $V_{ia} = \{\}$, i.e., if there are no If-Then-Else statements or arithmetic instructions included in the code, $L_{ia}$ 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 configurations, and hence the entire TKS $R$ will be labelled with the same time $t$.

As an example, left side of Fig. 10 shows a TKS, obtained from the Quartz program given on the right side.

**Theorem 1 (QuartzCompileTKS Iterations).** The algorithm $\text{QuartzCompileTKS}$ terminates in maximum $2^n$ iterations, where $n = |V_{ia}|$. 

![Fig. 10. Quartz and resulting TKS](image-url)
Proof:

The theorem is easily proven by induction on n:

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

If we assume that for \( n = |V_{ia}| \) the algorithm QuartzCompileTKS will terminate after maximum \( 2^n \) steps, then for \( n + 1 \) we have:

If \( |V_{ia}| = n + 1 \), then for \( x \in V_{ia} \) we separate \( \mathcal{U} \) as follows:

\[
\mathcal{U} := \mathcal{U}_x \cup \mathcal{U}_{\neg x}, \quad \text{where} \\
\mathcal{U}_x := \{(s, s') \in \mathcal{U} : x \in L_{ia}(s)\}, \\
\mathcal{U}_{\neg x} := \{(s, s') \in \mathcal{U} : x \notin L_{ia}(s)\} \\
\text{and } \mathcal{U}_x \cap \mathcal{U}_{\neg x} = \{\}.
\]

If we apply QuartzCompileTKS to \( \mathcal{U}_x \) and \( \mathcal{U}_{\neg x} \) separately, then we have

for \( \mathcal{U}_x : x \in L_{ia}(s), \forall s \in S_{ia} \) and

for \( \mathcal{U}_{\neg x} : x \notin L_{ia}(s), \forall s \in S_{ia} \).

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 \( V_{ia} \)-variables. The variable set of \( V_{ia} \) can then be expressed as \( V_{\neg x} = V_{ia} \setminus \{x\} \), where \( |V_{\neg x}| = n \).

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

4 Experimental results

We have implemented the algorithms in our tool framework Equinox. Figure 11 shows the flow of the techniques presented in this paper.

In this section, we present experimental results that we have obtained with some benchmarks of the current implementation. Tables 1, 2, and 3 show the experimental results that we have obtained for different benchmarks (Arbiter, Fischer’s Mutex Protocol, and Euclid’s algorithm) and different microprocessors (Pentium 4 2 GHz, Pentium 3 933 MHz and UltraSPARCIII 900 MHz). Table 1 contains the results of an arbiter that administrates the access of \( n \) processes to a shared resource. Note that the minimum and maximum execution times here are identical, since this benchmark contains no If-Then-Else-statements. Table 2 contains the results for Euclid’s algorithm to compute the greatest common divisor of two given \( n \) bit broad numbers. Finally, table 3
shows the results for Fischer’s mutual exclusion protocol [21] for \( n \) processes. The source codes of the programs can be found in [31, 25].

**Table 1. Results for Arbiter**

<table>
<thead>
<tr>
<th>( n )</th>
<th>variables state+time</th>
<th>BDD nodes</th>
<th>runtime [sec]</th>
<th>runtime [sec]</th>
<th>sec x ( 10^{-6} ) min,max</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>24 + 10</td>
<td>736</td>
<td>0.16</td>
<td>1.21</td>
<td>980, 980</td>
</tr>
<tr>
<td>5</td>
<td>29 + 11</td>
<td>1101</td>
<td>0.21</td>
<td>1.50</td>
<td>1238, 1238</td>
</tr>
<tr>
<td>6</td>
<td>34 + 11</td>
<td>1624</td>
<td>0.35</td>
<td>2.23</td>
<td>1824, 1824</td>
</tr>
<tr>
<td>7</td>
<td>39 + 11</td>
<td>2386</td>
<td>0.57</td>
<td>2.50</td>
<td>1982, 1982</td>
</tr>
<tr>
<td>8</td>
<td>44 + 12</td>
<td>3658</td>
<td>0.60</td>
<td>2.93</td>
<td>2065, 2065</td>
</tr>
<tr>
<td>9</td>
<td>49 + 12</td>
<td>3820</td>
<td>0.64</td>
<td>3.34</td>
<td>2481, 2481</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n )</th>
<th>variables state+time</th>
<th>BDD nodes</th>
<th>runtime [sec]</th>
<th>runtime [sec]</th>
<th>sec x ( 10^{-6} ) min,max</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>24 + 9</td>
<td>823</td>
<td>0.14</td>
<td>0.63</td>
<td>424, 424</td>
</tr>
<tr>
<td>5</td>
<td>29 + 10</td>
<td>1099</td>
<td>0.17</td>
<td>0.79</td>
<td>570, 570</td>
</tr>
<tr>
<td>6</td>
<td>34 + 10</td>
<td>1623</td>
<td>0.30</td>
<td>1.15</td>
<td>700, 700</td>
</tr>
<tr>
<td>7</td>
<td>39 + 10</td>
<td>3308</td>
<td>0.41</td>
<td>1.61</td>
<td>908, 908</td>
</tr>
<tr>
<td>8</td>
<td>44 + 10</td>
<td>3501</td>
<td>0.52</td>
<td>1.78</td>
<td>879, 879</td>
</tr>
<tr>
<td>9</td>
<td>49 + 11</td>
<td>3674</td>
<td>0.65</td>
<td>2.14</td>
<td>1075, 1075</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n )</th>
<th>variables state+time</th>
<th>BDD nodes</th>
<th>runtime [sec]</th>
<th>runtime [sec]</th>
<th>sec x ( 10^{-6} ) min,max</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>24 + 8</td>
<td>822</td>
<td>0.08</td>
<td>0.35</td>
<td>216, 216</td>
</tr>
<tr>
<td>5</td>
<td>29 + 9</td>
<td>1097</td>
<td>0.09</td>
<td>0.43</td>
<td>274, 274</td>
</tr>
<tr>
<td>6</td>
<td>34 + 9</td>
<td>1622</td>
<td>0.18</td>
<td>0.63</td>
<td>340, 340</td>
</tr>
<tr>
<td>7</td>
<td>39 + 10</td>
<td>3307</td>
<td>0.24</td>
<td>0.86</td>
<td>403, 403</td>
</tr>
<tr>
<td>8</td>
<td>44 + 9</td>
<td>3459</td>
<td>0.35</td>
<td>1.04</td>
<td>417, 417</td>
</tr>
<tr>
<td>9</td>
<td>49 + 9</td>
<td>3762</td>
<td>0.42</td>
<td>1.21</td>
<td>497, 497</td>
</tr>
</tbody>
</table>
The columns of the tables are as follows: the first column denotes the instantiation of the parameter of the benchmark (bitwidth or number of processes). The second column shows how many Boolean variables were necessary to encode the state transition diagram and the runtimes on the transitions. Column three shows how many BDD nodes were necessary to analyze the benchmark, which is a measure for memory consumption. Columns four and five show the determined runtimes for code-generation and for TKS-generation respectively. The last column, finally shows the determined runtimes for the minimal and maximal macro steps on the target machine (Pentium 4 2 GHz, Pentium 3 933 MHz and UltraSPARCIII 900 MHz).

<table>
<thead>
<tr>
<th>UltraSPARCIII</th>
<th>n</th>
<th>variables</th>
<th>state+time</th>
<th>BDD nodes</th>
<th>runtime [sec]</th>
<th>runtime [sec]</th>
<th>sec x 10^-6</th>
<th>min,max</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>13 + 10</td>
<td>534</td>
<td>0.03</td>
<td>22.56</td>
<td>418, 845</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18 + 11</td>
<td>2465</td>
<td>0.07</td>
<td>153.03</td>
<td>819, 1737</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>23 + 12</td>
<td>4274</td>
<td>0.22</td>
<td>936.59</td>
<td>1319, 2910</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>28 + 13</td>
<td>10150</td>
<td>0.27</td>
<td>5355.31</td>
<td>1966, 4510</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>33 + 10</td>
<td>30134</td>
<td>0.33</td>
<td>2942.20</td>
<td>271, 753</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>38 + 11</td>
<td>87980</td>
<td>0.78</td>
<td>15596.32</td>
<td>363, 1083</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pentium3</th>
<th>n</th>
<th>variables</th>
<th>state+time</th>
<th>BDD nodes</th>
<th>runtime [sec]</th>
<th>runtime [sec]</th>
<th>sec x 10^-6</th>
<th>min,max</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>13 + 9</td>
<td>505</td>
<td>0.02</td>
<td>12.63</td>
<td>238, 478</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18 + 11</td>
<td>2186</td>
<td>0.02</td>
<td>100.35</td>
<td>530, 1155</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>23 + 12</td>
<td>4332</td>
<td>0.14</td>
<td>661.26</td>
<td>935, 2086</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>28 + 12</td>
<td>29792</td>
<td>0.21</td>
<td>3910.56</td>
<td>1431, 3385</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>33 + 10</td>
<td>72839</td>
<td>0.25</td>
<td>2240.44</td>
<td>213, 542</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pentium4</th>
<th>n</th>
<th>variables</th>
<th>state+time</th>
<th>BDD nodes</th>
<th>runtime [sec]</th>
<th>runtime [sec]</th>
<th>sec x 10^-6</th>
<th>min,max</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>13 + 7</td>
<td>431</td>
<td>0.01</td>
<td>3.33</td>
<td>64, 124</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18 + 8</td>
<td>2001</td>
<td>0.02</td>
<td>23.20</td>
<td>145, 246</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>23 + 9</td>
<td>3532</td>
<td>0.09</td>
<td>140.48</td>
<td>226, 498</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>28 + 10</td>
<td>7963</td>
<td>0.14</td>
<td>774.26</td>
<td>318, 810</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>33 + 9</td>
<td>20937</td>
<td>0.17</td>
<td>435.25</td>
<td>48, 276</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Results for Euclid
### UltraSPARCIII

<table>
<thead>
<tr>
<th>n</th>
<th>variables</th>
<th>BDD runtime</th>
<th>state+time runtime</th>
<th>sec x 10^-6</th>
<th>min, max</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>15 + 10</td>
<td>455</td>
<td>0.05</td>
<td>8.08</td>
<td>556, 856</td>
</tr>
<tr>
<td>3</td>
<td>21 + 11</td>
<td>724</td>
<td>0.13</td>
<td>21.92</td>
<td>892, 1311</td>
</tr>
<tr>
<td>4</td>
<td>28 + 15</td>
<td>998</td>
<td>0.22</td>
<td>453.35</td>
<td>14661, 22717</td>
</tr>
<tr>
<td>5</td>
<td>34 + 12</td>
<td>1224</td>
<td>0.37</td>
<td>91.67</td>
<td>1838, 2779</td>
</tr>
<tr>
<td>6</td>
<td>40 + 12</td>
<td>1497</td>
<td>0.35</td>
<td>320.54</td>
<td>2220, 3338</td>
</tr>
</tbody>
</table>

### Pentium3

<table>
<thead>
<tr>
<th>n</th>
<th>variables</th>
<th>BDD runtime</th>
<th>state+time runtime</th>
<th>sec x 10^-6</th>
<th>min, max</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>15 + 9</td>
<td>426</td>
<td>0.01</td>
<td>3.94</td>
<td>259, 449</td>
</tr>
<tr>
<td>3</td>
<td>21 + 10</td>
<td>640</td>
<td>0.08</td>
<td>10.99</td>
<td>424, 673</td>
</tr>
<tr>
<td>4</td>
<td>28 + 14</td>
<td>998</td>
<td>0.21</td>
<td>219.87</td>
<td>6307, 11107</td>
</tr>
<tr>
<td>5</td>
<td>34 + 11</td>
<td>1068</td>
<td>0.29</td>
<td>49.80</td>
<td>785, 1382</td>
</tr>
<tr>
<td>6</td>
<td>40 + 11</td>
<td>1472</td>
<td>0.26</td>
<td>225.01</td>
<td>950, 1655</td>
</tr>
</tbody>
</table>

### Pentium4

<table>
<thead>
<tr>
<th>n</th>
<th>variables</th>
<th>BDD runtime</th>
<th>state+time runtime</th>
<th>sec x 10^-6</th>
<th>min, max</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>15 + 8</td>
<td>411</td>
<td>0.01</td>
<td>1.48</td>
<td>107, 133</td>
</tr>
<tr>
<td>3</td>
<td>21 + 8</td>
<td>456</td>
<td>0.06</td>
<td>3.76</td>
<td>151, 210</td>
</tr>
<tr>
<td>4</td>
<td>28 + 12</td>
<td>1098</td>
<td>0.13</td>
<td>79.94</td>
<td>2682, 3890</td>
</tr>
<tr>
<td>5</td>
<td>34 + 9</td>
<td>1085</td>
<td>0.20</td>
<td>24.14</td>
<td>328, 454</td>
</tr>
<tr>
<td>6</td>
<td>40 + 10</td>
<td>1430</td>
<td>0.15</td>
<td>134.25</td>
<td>397, 550</td>
</tr>
</tbody>
</table>

#### Table 3. Results for Fischer’s Mutex Protocol

### References

11. ECL Homepage. Website. http://www-cad.eecs.berkeley.edu/