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

G. Logothetis, K. Schneider*, and C. Metzler

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

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

Abstract

Synchronous programming languages are well-suited for the implementation and verification of real-time systems. The main benefit for the estimation of real-time constraints is thereby that the macro steps provided by the semantics of synchronous languages can be directly used as building blocks for runtime analysis. We describe a new approach to determine the execution times of the macro steps of a synchronous program wrt. given microprocessors and generate real-time formal models endowed with notions of physical time for formal verification purposes. The execution times are exact, since we consider all possible input sequences.

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, 2, 7, 18] 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 [9] has important advantages for the analysis and verification of the real-time behavior, due to its clean formal semantics. Synchronous languages distinguish between micro and macro steps [13]. 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. In [19], 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. 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 [23]. Low-level WCET analysis is done on the object code, and hence depends on the chosen hardware/software partitioning and the chosen architecture. 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 [14, 10, 21, 22].

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 highly inaccurate 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 [18]), 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 [14] 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 labeled with notions of physical time, denoting the exact execution times of the macro steps that correspond to the transitions.

Our approach takes advantage of established symbolic techniques to efficiently manipulate large finite state transition systems by means of binary decision diagrams (BDDs) [6]. We consider timed Kripke structures (TKSs) as formal models as proposed in [18]. 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 [18], as well as for further analysis purposes like WCET and BCET, using techniques like the ones presented in [20].

There is not much related work. In [25], 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 [4]. 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 generation 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, [4] 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 [4].

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. We then conclude with preliminary experimental results.

2 Background

2.1 Synchronous Languages

Synchronous languages [12] like many Esterel-variants [3, 8, 16, 24] 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 modeled 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. Hence, the execution of micro steps does not take time (in the programmer’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 \( \mathbb{N} \to \alpha \) 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 causal-ity problems [3]). 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

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

\[
\text{req} \land (y \neq 0)/\{(1, x := a), (1, y := b), (1, c := 0)\}
\]

\[
\text{rdy} \lor (y = 0)/\{}
\]

\[
\bar{\pi}/\{}
\]

Figure 1. Russian Multiplication Automaton
that interacts with a data flow of finitely many variables of possibly infinite data types. For example, consider the transition system given in Figure 1. It shows the semantics of a Quartz program, implementing a Russian multiplication algorithm (the source code can be found in [20]). The three states correspond with the situations where the control flow is either outside the program or at one of the locations labeled with \( t \) 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.

### 2.2 Timed Kripke Structures (TKSs)

To model real-time systems we briefly introduce in this section timed Kripke structures (TKS) as proposed in [18]. 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 \((I, S, R, \mathcal{L})\), such that \( S\) is a finite set of states, \( T \subseteq S \) is the set of initial states, and \( R \subseteq S \times N \times S \) 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 \((s, t, s') \in R\), we have \( t > 0 \) and that for any \( s \in S \), there must be a \( t \in N \) and a \( s' \in S \) such that \((s, t, s') \in R \) holds.

It is crucial to understand what is modeled by a TKS. In the sense of [18], we use the following interpretation: A transition from state \( s \) to state \( s' \) with label \( k \in N \) 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 labeled 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 labeled with the time consumption 1.

### 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 [24]. 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. determine and capture the execution times required for all actions of the code. This is done during the execution of the code. Simultaneously, our method constructs a low-level timed Kripke structure by labeling the transitions of the system by the determined execution times of the corresponding code actions for the given microprocessor. Our technique 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:

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

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 [5, 3], 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 particularly for applications involved in embedded systems, where the memory size is usually limited.

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

Our generated program is one static block which is executed in an endless loop. The writing to cache data and also the execution of instructions is performed in the same order in each loop. By executing this block several times for a given input we obtain a cache-configuration which is very similar to the configuration in a real environment. A runtime-analysis for this block can then be directly per-
formed 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 [14] in order to handle this problem. Techniques like the ones presented in [14] can be easily endowed in our tool and are part of our current implementation work.

Recall that the transition system is derived from a synchronous program. Such a transition system contains only boolean operators. Consequently, this is also the case for the generated C-Code by QuartzCompileC. 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.

As an example consider the If-Then-Else-statement shown in Fig. 3. The two transitions refer to the statement $next(a) = b?c : d$ and consume different times $t_1$ and $t_2$. On the other hand the statement $next(a) = b \lor c$ of Fig. 4 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 key idea is to use an efficient technique to obtain the variables occurring inside of arguments of If-Then-Else 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 $V := V_{if} \cup V_{nonif}$. The set $V_{if}$ consists of the variables which occur inside the arguments of if-statements, while

\textbf{Figure 2. Runtime analysis - TKS construction}

We assume that we already have a function QuartzCompileUDS for the code generation, that computes an equivalent unit delay structure (UDS) $K_U$ of a given Quartz program $P$. The states of this structure correspond with the states of the program $P$ and are labeled with Boolean variables. Such a function is essentially implemented by any compiler, like the ones described in [24].

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

\begin{verbatim}
function QuartzCompileTKS(P)
    {I, S, U} := QuartzCompileUDS(P);
    C := QuartzCompileC(U);
    R := {};
    while U \neq {} do
        S_if :=
            \{ s \in S \mid \exists s' \in S.s(s, s') \in U \land L_if(s) \neq false \};
        s_time := choose any of S_if;
        time := RuntimeC(s_time, C);
        U_time :=
            \{ (s, s') \in U \mid \exists s, s' \in S.L_if(s) = L_if(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
\end{verbatim}

\textbf{Figure 3. Example If-Then-Else Statement}

\textbf{Figure 4. Example Boolean Operator}
The theorem is easily proven by induction on \( n \): if \( \mathcal{V}_{if} = \{ \} \) and \( \mathcal{V}_{nonif} \neq \{ \} \), then according to the definition of \( L_{if} \) we have \( L_{if}(s) = \text{true}, \forall s \in \mathcal{S} \). But then \( \mathcal{U}_{time} = \mathcal{U} \) holds trivially and hence also \( \mathcal{U} \setminus \mathcal{U}_{time} = \{ \} \), which terminates the algorithm after one iteration, i.e. \( 1 = 2^{\lvert \mathcal{V}_{if} \rvert} \). If we assume that for \( n = \lvert \mathcal{V}_{if} \rvert \) the algorithm QuartzCompileTKS will terminate after maximum \( 2^n \) steps, then for \( n + 1 \) we have: If \( \lvert \mathcal{V}_{if} \rvert = n + 1 \), then for \( x \in \mathcal{V}_{if} \) we separate \( \mathcal{U} \) as follows: \( \mathcal{U}_x := \mathcal{U}_x \cup \mathcal{U}_{x,x} \), where \( \mathcal{U}_x := \{ (s, s') \in U \in L_{if}(s) \}, \mathcal{U}_{x,x} := \{ (s, s') \in U \in L_{if}(s) \} \) and \( \mathcal{U}_x \cap \mathcal{U}_{x,x} = \{ \} \). If we apply QuartzCompileTKS to \( \mathcal{U}_x \) and \( \mathcal{U}_{x,x} \) separately, then we have for \( \mathcal{U}_x : x \in L_{if}(s), \forall s \in S_{if} \) and for \( \mathcal{U}_{x,x} : x \notin L_{if}(s), \forall s \in S_{if} \). In other words, \( x \) has fixed values in \( \mathcal{U}_x \) and \( \mathcal{U}_{x,x} \) and hence, it must not be considered for determining the possible configurations of \( \mathcal{V}_{if} \). The variable set of \( \mathcal{V}_{if} \) can then be expressed as \( \mathcal{V}_{x} = \mathcal{V}_{if} \setminus \{ x \} \), where \( \lvert \mathcal{V}_{x} \rvert = n \). According to the induction’s assumption, \( \mathcal{U}_x \) and \( \mathcal{U}_{x,x} \) can then be computed in maximum \( 2^n = 2^{\lvert \mathcal{V}_{x} \rvert} \) iterations and hence, for the entire problem we have a maximum of \( 2 \cdot 2^n = 2^{\lvert \mathcal{V}_{if} \rvert} \) iterations.

4 Experimental results

We have implemented the algorithms in our tool framework and have tested several benchmarks. Due to lack of space, we present in this section experimental results that we have obtained with one benchmark, Fischer’s mutual exclusion protocol [17], for different microprocessors (Pentium 4 2 GHz, Pentium 3 933 MHz and UltraSPARCIII 900 MHz).

<table>
<thead>
<tr>
<th>UltraSPARCIII</th>
<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}</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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium3</td>
<td>n</td>
<td>variables state-time</td>
<td>BDD nodes</td>
<td>runtime [sec]</td>
<td>runtime [sec]</td>
<td>sec x 10^{-6}</td>
<td>min max</td>
</tr>
<tr>
<td>2</td>
<td>15 + 9</td>
<td>426</td>
<td>0.01</td>
<td>3.94</td>
<td>259, 449</td>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium4</td>
<td>n</td>
<td>variables state-time</td>
<td>BDD nodes</td>
<td>runtime [sec]</td>
<td>runtime [sec]</td>
<td>sec x 10^{-6}</td>
<td>min max</td>
</tr>
<tr>
<td>2</td>
<td>15 + 8</td>
<td>411</td>
<td>0.01</td>
<td>1.48</td>
<td>107, 133</td>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>46 + 10</td>
<td>2146</td>
<td>0.26</td>
<td>1322.47</td>
<td>441, 633</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Results for Fischer’s Mutex Protocol

Table 1 shows the results for Fischer’s Mutex protocol [17] for \( n \) processes. The columns of the tables are as follows: the first column denotes the instantiation of the parameter...
of the benchmark (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.

References


