Generating Formal Models for Real-Time Verification by Exact Low-Level Runtime Analysis of Synchronous Programs

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

University of Karlsruhe
Institute for Computer Design and Fault Tolerance
Department of Computer Science
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 synchronous programs can be directly used for runtime analysis: If synchronous circuits are generated from these descriptions, the macro steps are implemented by combinatorial circuits, and if software is generated, they correspond to basic building blocks that do not contain loops. In this paper, we describe methods to generate timed transitions systems from a synchronous program by taking the final architecture into account. For software synthesis, this requires to consider different microprocessors and compilers, and for hardware synthesis, this requires to consider a hierarchy of clocks to optimize the clock speed.

1 Introduction

The design of embedded systems requires design methods to realize systems partly in software and hardware. Often, only software is generated and the hardware design reduces to the choice of a particular microcontroller that is on the one hand, powerful enough to meet the required real-time constraints, and on the other hand, cheap enough to reduce production costs. For this reason, there is a need for methods and tools to help designers to make good decisions about the hardware-software partition and the choice of the microprocessors for the final realization of the system.

Unfortunately, the languages used in hardware and software design are rather different. Hardware description languages like VHDL [3, 42] or Verilog [41] describe the timing behavior of the circuits in an event-oriented style that is useful to implement efficient simulators. However, even for hardware synthesis, this requires difficult algorithms to obtain an efficient hardware circuits. Generating software is even more difficult, and maybe this is one of the reasons why new approaches like SystemC [17] gear towards programming languages.

An interesting approach to achieve realization independent descriptions is followed by synchronous languages [13] like Esterel [6, 11]. Synchronous languages have 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 thread interactions like preemption and suspension. It is very comfortable to be able to either generate hardware or software from the same synchronous program.
- Synchronous languages have clean formal semantics that allow an easy integration of verification tools [24, 35], e.g., by the definition of control flow predicates [35].
- Synchronous languages distinguish between micro and macro steps [14, 36]. 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 its execution. Consequently, all threads run in lockstep and automatically synchronize after each macro step. The important fact for real-time verification is that the languages are designed in a way that macro steps can not contain loops.
In [24], it has been shown how timed transition systems can be generated from synchronous programs for real-time verification. To this end, a new statement \textbf{abstract S end abstract} has been introduced to declare the macro steps of $S$ as uninteresting for timing analysis. This means that the intermediate states of $S$ are marked, so that a post-processing step can remove them. The post-processing eliminates marked states and redirects transitions that are instead labeled with the number of macro steps that they represent. Hence, the techniques introduced in [24] may be viewed as 'hyper steps' that correspond to one transition of the transition system, but contain a finite number of macro steps. Note, however, that this has no effect on code generation and is only used to simplify runtime analysis.

In this paper, we refine this approach by the introduction of a clock hierarchy similar to [32, 33]. This is achieved with the new statement \texttt{newclock c in S end abstract} that declares the signal $c$ as the new clock signal so that $c$ determines the execution of the macro steps. Macro steps of other threads may run at another clock speed, and they may even influence the clock signal $c$ of other threads. In particular, this is useful for hardware synthesis, when it is not a priori clear which parts of the system are implemented as combinatorial or sequential circuits. The clock hierarchy can be used to balance the macro steps, i.e., to achieve that all macro steps require a similar amount of physical time. Related work used multiclocks for other purposes: multiclocks have been used in [32, 33] to implement event-oriented systems as modeled by VHDL with Esterel programs. In other languages like Lustre, multiclocks are a fundamental part of the language.

Besides the use of clock hierarchies for hardware implementation, we also consider the runtime analysis of a software realization. While in the programmer’s view, the micro steps of a synchronous program are executed in zero time, this is, of course, not the case for final program. To estimate the execution time of a given program, [31] introduced the notion of worst case execution time (WCET) analysis. In general, two levels of WCET analysis can be distinguished: the high-level and the low-level phase.

High-level WCET analysis is applied to an architecture independent description of the system and has the task to compute path information [30] like unfeasible computations or bounds on the maximal number of loop iterations. Unfortunately, high-level WCET analysis is undecidable when infinite data types are used, and therefore only limited automation can be achieved. State of the art approaches use abstract interpretation [10], symbolic execution [21, 28], or special restrictions on the loops [16]. A major problem of high-level WCET analysis is that the maximal number of computation steps of a statement (like a loop) may heavily depend on the input data, but some of the approaches do simply compute WCET bounds for substatements and add these afterwards. However, simply adding the bounds clearly yields highly pessimistic bounds. Recently, an automatic WCET analysis approach based on symbolic state space exploration was introduced in [25]. There, the minimal and maximal numbers of macro steps between given program locations are computed. This computation is exact, because it considers all possible input sequences.

Low-level WCET analysis is done on the object code, and hence, it depends on the chosen hardware/software partitioning and on the chosen architecture. For simple microcontrollers like the still mainly used 8-bit processors, this is relatively simple, but it becomes significantly more complicated for modern architectures that require the consideration of caches, pipelines, branch prediction, interrupts, and other effects. There exist several approaches to estimate the worst- or best-case execution time of a given program on architectures with caches or superscalar execution by runtime analysis [1, 12, 15, 29].

Synchronous languages lend themselves also well for a low-level runtime analysis. The macro steps can be directly used as building blocks since they can not contain data-dependent loops. In [40], 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 [7]. 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 [2] for verification of temporal properties.

Considering only worst- or best-case execution times 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 [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 [15] can perform a so-
called “point-to-point” analysis, but they are not inte-
grated in a formal framework for a further analysis.

In this paper, we present a technique that per-
forms an exact and detailed low-level (architecture-
dependent) runtime analysis of synchronous pro-
grams. Our approach computes the exact execution
times of all possible single transitions of a system and
simultaneously generates a real-time formal model,
whose transitions are labeled with numbers denoting
the exact execution times of the macro step that cor-
responds to the transition. These formal models can
then be directly used for architecture-dependent real-
time verification, as well as for further analysis pur-
poses like WCET and BCET, using techniques similar
to the ones presented in [24, 25]. To this end, we use
timed Kripke structures (TKSs) as formal models as pro-
posed in [22]. Hence, we propose the following design
flow, starting from a synchronous program:

1. verify desired specifications at a logical level to
find design errors [22, 37] and to estimate the run-
time in terms of macro steps [24, 25]
2. choose appropriate hardware to realize the em-
bedded system and automatically derive code for
software or hardware design [34, 35]
3. perform a low-level runtime analysis using the
methods of this paper to determine the actual re-
action time of the embedded real-time system.
4. verify desired specifications and real-time con-
straints at a physical level

The main idea is thereby to first construct a transition
system and obtain executable code for the given syn-
chronous program. After that, an architecture depen-
dent low-level runtime analysis is used to label the
transitions with the obtained execution time. The out-
line 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 run-
time analysis and the construction of formal models.
We then conclude with preliminary experimental re-

tults.

2 Background

2.1 Synchronous Languages

Synchronous languages [13] like many Esterel-
variants [4, 6, 18, 20, 34, 35] 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 nat-
ural numbers \( \mathbb{N} \). The execution of a synchronous pro-
gram 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 al-
ways the same amount of a logical time (in the pro-
gerammer’s model). Consumption of time, i.e., the be-

inning of a new macro step, must be explicitly pro-
grammed with special statements like the \texttt{pause} state-
ment in Esterel. 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.

Concerning the data flow, every variable, and
hence, every data expression has a unique value for
every macro step. Hence, the semantics of a data type
expression is a function of type \( \mathbb{N} \rightarrow \alpha \) for some type \( \alpha \).
The variable’s values are manipulated by micro
steps of a macro step, so that their ordering is irrele-
vant for the semantics. For code generation, the order-
ing is however very important, since one can circum-
vent so called causality problems [5] by a good ordering.

The entire semantics of a synchronous program \( \mathcal{P} \)
can be given as a finite state transition system \( A_\mathcal{P} \): the
states of \( A_\mathcal{P} \) reflect the possible combinations of con-


roll flow locations of the program (a control flow lo-
cation is a point in the program text, where the con-

rol flow might rest for one unit of time, i.e., the \texttt{pause}
statement). As the language allows the implementa-
tion 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. Execu-
tion of a transition will then invoke some actions of
the data values. Hence, the semantics can be repre-
sented by a finite state control flow that interacts with
a data flow of finitely many variables of possibly infi-
nite data types.

For example, consider the Quartz\(^1\) program given
in Figure 1 (it implements a Russian multiplication
algorithm). The semantics is the transition system
given in Figure 2. The three states correspond with
the situations where the control flow is either outside
the program or at one of the locations labeled with \( \ell \)
or \( \texttt{rdy} \). The labels of the transitions are of the form
\( \Phi / \{(\gamma_1, \alpha_1), \ldots, (\gamma_n, \alpha_n)\} \) with the follow-
ing 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.

\(^1\)Quartz is an Esterel variant [34, 35].
module RussMult:
input req, a : I[n], b : I[n];
output c : I[n];
local x : I[n], y : I[n]
label rdy;
loop
  rdy : await req;
  x := a; y := b; c := 0;
  while y ≠ 0 do
    if odd(y) then
      next(c) := c + x
    end;
    next(x) := 2 · x;
    next(y) := y/2;
  end while
end loop
end module

Figure 1. Russian Multiplication as Quartz Code

\begin{align*}
(y ≠ 0) &/\{(odd(y), next(c) := c + x), \\
(1, next(x) := 2 · x), \\
(1, next(y) := y/2)\}
\end{align*}

Beneath the comfortable programmer’s model given by the macro step abstraction, synchronous languages provide a rich set of statements, in particular, for manipulating the execution of concurrent threads: 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 [34, 35] and to the Esterel primer, which is an excellent introduction to synchronous programming [6].

In [24], we have introduced a new statement of the form \texttt{abstract S end}. The statement has no effect for code generation, i.e., the code is generated for the substatement \(S\). However, for the generated transition system, the states inside \(S\) are marked. This allows us to eliminate these states and to redirect the transitions to their successor states. Repeating this, finally eliminates all marked states. During the elimination of marked states, we label the redirected transitions by the number of transitions they represent, so that we do not lose the runtime information that is required for a high-level runtime analysis (where the number of executed macro steps is measured).

This approach can be extended to a clock hierarchy as follows: we introduce a further new statement \texttt{newclock c in S end}, where \(c\) is a new signal. The meaning of this statement is as follows: code is generated for the statement \(S'\) that is obtained from \(S\) by replacing every \texttt{pause} statement by \texttt{await} \(c\). Hence, the next macro step is not necessarily executed at the next (synchronous) point of time, but only if the clock signal \(c\) allows it. This allows us to run threads at a different clock speed, which has many applications: For example, consider the program of Figure 1. Using clock signals, we can achieve that the sequential computation of the multiplication is hidden for other threads, if their clock signal is such that between two clocks a complete multiplication can be performed. Note that clock signals to control the granularity of macro steps allows us to hide even data-dependent loops as the one used in Figure 1. An application of this statement is to explore the design space for a hardware implementation in that we can (1) either implement the different clocks or (2) decide which parts should be implemented with combinatorial or sequential circuits. This can also be used to achieve a good balanced program, where all macro steps require a similar amount of time, so that an efficient hardware implementation is obtained. Another application is to mimic instructions that are not available on a microprocessor, but that can be ‘outsourced’ to a special hardware unit, or to define a suitable instruction set for a microprocessor for a particular system.
2.2 Timed Kripke Structures (TKSs)

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

**Definition 1 (Timed Kripke Structure (TKS))** A timed Kripke structure over the variables \( V \) 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 \( L(s) \subseteq V \) 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 an \( a \in \mathbb{N} \) and a \( s' \in S \) such that \((s, t, s') \in R\).

It is crucial to understand what is modeled 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 the following, there is no information about the intermediate points of time with \( t_0 < t < t_0 + k \).

The translation from a finite state representation like the one given in Figure 2 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. 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 technique to perform exact low-level runtime analysis and to construct a timed Kripke structure from the automaton representation obtained from a synchronous program. We use these timed Kripke structures for real-time verification, so that we can reason about real-time formal specifications. To this end, we use the verification tools described in [22], and the methods for determining path information [24, 25].

Our goal is to generate executable code from 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. To this end, 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 determine and capture the execution times required for all actions of the code. This is done during the execution of the code. Simultaneously, our method constructs a low-level timed Kripke structure by labeling the transitions of the system by the determined execution times of the corresponding code actions for the given microprocessor. Our approach takes advantage of established symbolic techniques to efficiently manipulate large finite state transition systems by means of binary decision diagrams (BDDs) [9].

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. 2 follows the following state transition equations:

\[
\text{next}(r\text{dy}) := \begin{cases} \neg r\text{dy} \land \neg \ell \land s\text{t}\lor r\text{dy} \land (\neg \text{req} \lor (y = 0)) \lor \\ \ell \land (y = 0) \end{cases} \\
\text{next}(\ell) := \begin{cases} r\text{dy} \land \text{req} \lor \ell \lor (y \neq 0) \\ \text{if } \ell \land (y = 0) \land \text{odd}(y) \text{ then } 2 \cdot x \\ \vdots \end{cases} \\
\text{next}(x) := \ldots \\
\text{next}(y) := \ldots \\
\text{next}(c) := \ldots 
\]

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, 8], 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, and in theory at most quadratic in the size of the synchronous program).

The exact runtime analysis of single instructions used in a sequential program is a complicated task due to architecture-dependent features like 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: Note that our generated program is a single 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 iteration. 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 [15] in order to handle this problem. Techniques like the ones presented in [15] 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 code generation, that computes a unit delay structure (UDS) \( K_\mathcal{U} \) from a given Quartz program \( P \). The states of this structure correspond to 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 [34, 35].

function QuartzCompileTKS(\( P \))
\( (I, S, U) := \) QuartzCompileUDS(\( P \));
\( C := \) QuartzCompileC(\( U \));
\( R := \{\} \);
while \( U \neq \{\} \) do
\( S_f := \{ s \in S \mid \exists s' \in S, (s, s') \in U \land L_{if}(s) \neq false\};
\( s_{time} := \) choose any of \( S_f \);
\( time := \) 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

**Figure 3. TKS construction**

To finally obtain an equivalent TKS \( K_\mathcal{R} \), we have to endow the transitions by their required amount of execution time. These labels are obtained by measuring the runtime for generated platform-specific code for the micro steps that are related to the transitions. We first use a function QuartzCompileC which translates the obtained UDS \( K_\mathcal{U} \) into C-Code according to the method described above. The TKS is constructed by the algorithms given in Figure 3. 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.

An important observation for this analysis is to consider the state transition equations that are generated by the translation of the Quartz program to an equation system. 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, and therefore lead to different execution times than other Boolean operations.

For this reason, we distinguish for the construction of the TKS between variables that occur in conditional statements and others, hence, the set of variables is partitioned as \( V := V_{if} \cup V_{nonif} \). We also distinguish between these sets in the labeling function of the TKS, i.e., we have \( L_{if} \) and \( L_{nonif} \) that correspond to \( V_{if} \) and \( V_{nonif} \), respectively. A detailed explanation of the algorithm is given in [26, 27], where we also proved the following fact:

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

**4 Experimental results**

We have implemented the algorithms in our tool framework Equinox. Figure 4 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 [19] for \( n \) processes. The source codes of the programs can be found in [25, 38].

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).

5 Conclusions

In this paper, we have extended our tool Equinox to an exact and detailed low-level runtime analysis which allows us to use it for low-level real-time formal verification. We presented a technique for analyzing the execution times of all single transitions of a synchronous program. This allows Equinox to generate low-level timed Kripke structures so that their transitions are labeled with the physical times required to execute the code on the transitions. The generated transition systems have timed transitions that correspond to non-interruptible atomic actions. For verification purposes of such real-time systems, the real-time temporal logic JCTL was introduced in [22] as an extension of the CTL temporal logic. We have evaluated the tools by

Table 1. Results for Arbiter

<table>
<thead>
<tr>
<th>n</th>
<th>variables</th>
<th>state+time</th>
<th>BDD nodes</th>
<th>time [sec]</th>
<th>time [sec]</th>
<th>sec x 10^{-6}</th>
<th>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>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>39 + 11</td>
<td>2386</td>
<td>0.57</td>
<td>2.5</td>
<td>1982, 1982</td>
<td></td>
<td></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>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Results for Euclid

<table>
<thead>
<tr>
<th>n</th>
<th>variables</th>
<th>state+time</th>
<th>BDD nodes</th>
<th>time [sec]</th>
<th>time [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>
</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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

Table 4. Results for Arbiter

<table>
<thead>
<tr>
<th>n</th>
<th>variables</th>
<th>state+time</th>
<th>BDD nodes</th>
<th>time [sec]</th>
<th>time [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>
</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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

Figure 4. Information flow

C Code

GenerateCode

Run

Applytime

TKS

Compile

qif

JERRY

UDS

QRZ

Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS'03)
0-7695-2044-8/03 $ 17.00 © 2003 IEEE
Table 3. Results for Fischer’s Mutex Protocol

<table>
<thead>
<tr>
<th>n</th>
<th>variables state+time</th>
<th>BDD nodes</th>
<th>time [sec]</th>
<th>time [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>5.563</td>
<td>856</td>
</tr>
<tr>
<td>3</td>
<td>21 + 11</td>
<td>724</td>
<td>0.13</td>
<td>21.92</td>
<td>892</td>
<td>1311</td>
</tr>
<tr>
<td>4</td>
<td>28 + 15</td>
<td>998</td>
<td>0.22</td>
<td>453.35</td>
<td>14661</td>
<td>22717</td>
</tr>
<tr>
<td>5</td>
<td>34 + 12</td>
<td>1224</td>
<td>0.37</td>
<td>91.67</td>
<td>1838</td>
<td>2779</td>
</tr>
<tr>
<td>6</td>
<td>40 + 12</td>
<td>1497</td>
<td>0.35</td>
<td>320.54</td>
<td>2220</td>
<td>3338</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>variables state+time</th>
<th>BDD nodes</th>
<th>time [sec]</th>
<th>time [sec]</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</td>
<td>449</td>
</tr>
<tr>
<td>3</td>
<td>21 + 10</td>
<td>640</td>
<td>0.08</td>
<td>10.99</td>
<td>424</td>
<td>673</td>
</tr>
<tr>
<td>4</td>
<td>28 + 14</td>
<td>998</td>
<td>0.21</td>
<td>219.87</td>
<td>6307</td>
<td>11107</td>
</tr>
<tr>
<td>5</td>
<td>34 + 11</td>
<td>1068</td>
<td>0.29</td>
<td>49.80</td>
<td>785</td>
<td>1382</td>
</tr>
<tr>
<td>6</td>
<td>40 + 11</td>
<td>1472</td>
<td>0.26</td>
<td>225.01</td>
<td>950</td>
<td>1655</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>variables state+time</th>
<th>BDD nodes</th>
<th>time [sec]</th>
<th>time [sec]</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</td>
<td>133</td>
</tr>
<tr>
<td>3</td>
<td>21 + 8</td>
<td>456</td>
<td>0.06</td>
<td>3.76</td>
<td>151</td>
<td>210</td>
</tr>
<tr>
<td>4</td>
<td>28 + 12</td>
<td>1098</td>
<td>0.13</td>
<td>79.94</td>
<td>2682</td>
<td>3890</td>
</tr>
<tr>
<td>5</td>
<td>34 + 9</td>
<td>1085</td>
<td>0.20</td>
<td>24.14</td>
<td>328</td>
<td>454</td>
</tr>
<tr>
<td>6</td>
<td>40 + 10</td>
<td>1430</td>
<td>0.15</td>
<td>134.25</td>
<td>397</td>
<td>550</td>
</tr>
<tr>
<td>7</td>
<td>46 + 10</td>
<td>2146</td>
<td>0.26</td>
<td>1322.45</td>
<td>405</td>
<td>633</td>
</tr>
</tbody>
</table>

At the moment, our tools for code generation do not exploit the arithmetic instructions of the microprocessors. Instead, the arithmetic operations of a synchronous program are mapped to the Boolean level and are implemented as bit-operations on the microprocessors. There is no need to proceed like this, and we plan in our very next future work to extend our tool set so that also the arithmetic instructions of the processors are taken into account. A first attempt in this direction has been made in [26].

Another approach that has been recently given in [39] treats arithmetic operations in a symbolic way, i.e., variables of the program that hold integer data correspond to single integer variables in the TKS. This requires to have more powerful representations for state sets, since these may now contain infinitely many configurations. The WCET analysis method given in [39] was applied to assembler programs, but it may obviously be used for synchronous programs as well.

Unfortunately, it is not possible to perform runtime analysis by replacing the actions of the transitions in the control-dataflow graph (as shown in Figure 2) with the corresponding runtimes. We need to additionally know how often a cycle may be executed, and this information is only available when we know about the actions.

Our method does not aim at analyzing large systems which consist of thousands of lines of code. We believe that a complete and accurate runtime analysis of large systems can only be achieved by combining different methods for each level of abstraction. Nevertheless, our method is well-suited for core routines which are frequently called and contribute a large part to the total execution time.

References


