Exact High Level WCET Analysis of Synchronous Programs by Symbolic State Space Exploration

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

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

Abstract

In this paper, a novel approach to high-level (i.e. architecture independent) worst case execution time (WCET) analysis is presented that automatically computes exact bounds for all inputs. To this end, we make use of the distinction between micro and macro steps as usually done by synchronous languages. As macro steps must not contain loops, a later low-level WCET analysis (architecture dependent) is simplified to a large extent.

Checking exact execution times for all inputs is a complex task that can nevertheless be efficiently done when implicit state space representations are used. With our tools, it is not only possible to compute path information by exploring all computations, but also to verify given path information.

1. Introduction

Only about 2% of the microprocessors that have been produced in 1999 were contained in desktop computers, while the vast majority was contained in so-called embedded systems [16, 25]. These systems do not have a user interface, but interact directly with their environment. Embedded systems are parts of aircrafts, automobiles, domestic appliances, consumer electronics, etc., and are as such very often real-time systems. As they often occur in safety-critical applications, their correctness is mandatory.

Designing a real-time system is a relatively error-prone task, especially when the system consists of several interacting processes, which is the usual case. Decreasing time-to-market and the overall design costs requires to check as early as possible in the design flow whether the desired specifications are met.

For real-time systems, the essential task is to guarantee that certain actions are executed within some strict deadlines or that they will start only after some point of time. To avoid expensive redesigns, it is important to check as soon as possible in the design flow whether the real-time constraints are met. For this reason, the estimation of *worst case execution time (WCET)* [27] was proposed that usually consists of two phases: the *low-level* and the *high-level* WCET analysis. Low-level analysis is done on the object code, and hence depends on the chosen hardware/software partitioning and the chosen architecture (microcontrollers). For simple microcontrollers like the still mainly used 8-bit processors, this is a straightforward task, but it is more complicated for modern architectures that require the consideration of caches, pipelines, branch prediction, interrupts, etc.

In contrast, *high-level* WCET analysis is applied to an architecture independent description of the system and has the task to compute *path information* [26] like unfeasible computations or bounds on the maximal number of loop iterations. Clearly, 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 [13], symbolic execution [24, 21], or special restrictions on the loops [18]. A major problem of the high-level WCET analysis is that the maximal number of computation steps of a statement (like a loop) may heavily depend on the input data, but some of the approaches do simply compute the WCET bounds for the substatements and add these afterwards. However, simply adding the maximal bounds for all substatements clearly yields highly pessimistic bounds. To estimate tighter bounds, [1] proposed not to compute constants, but functions that depend on inputs as WCET bounds. This approach is able to compute much tighter bounds, but is certainly a deeply manual task, although guided by computer algebra systems.

In this paper, we propose a new and completely automatic approach to high-level WCET analysis. For a given program, we compute the best and worst runtime in terms of macro steps for all inputs at once. We are even able to compute the input sequences that require these bounds. For this

1530-1591/03 $17.00

purpose, we have to assume that all data types were finite[1], so that the overall problem becomes decidable.

Nevertheless, checking all input sequences is still a highly complex task. A key idea of this paper is therefore to use modern techniques, namely symbolic state space exploration techniques [5, 8], developed for the verification of temporal properties of reactive systems. These techniques allow us to explore state spaces with more than 10^{20} states, in some cases even with more than 10^{200} states, or after suitable abstractions even infinitely many states. These methods are essentially based on the implicit representation of the system by propositional formulas which is often called a *symbolic representation*[2].

Beneath the symbolic state space traversal, the other essential key ingredient to our solution is the use of *synchronous programming languages* like Esterel [2, 4] or Quartz [28, 29] to achieve realisation independent descriptions of the system. These languages are well-suited for a high-level WCET analysis 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 preemption and suspension.
- Synchronous languages have a clean formal semantics. In particular, the definition of control flow predicates in [29] supports static runtime analysis. Moreover, there are already tools for verifying synchronous programs by symbolic model checking [29, 23].
- Synchronous languages distinguish between micro and macro steps [17]. 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 WCET analysis 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.

Our procedure works as follows: We start with a synchronous program and translate this program as described in [29] into a finite-state automaton (representing the control flow) whose transitions are labeled with a set of conditional assignments (representing the data flow). Each transition directly corresponds to a macro step of the program. This description is given in an implicit form, so that we can use it for symbolic state space exploration [5, 8]. The algorithm presented in this paper (cf. Figure 3) is then used to compute the minimal and maximal numbers of macro steps necessary to reach a set of control flow states $\mathcal{S}_\gamma$ from another set of control flow states $\mathcal{S}_\alpha$. Additionally, we can count the number of visits of a third set of states $\mathcal{S}_\beta$ while the control flow moves from $\mathcal{S}_\alpha$ to $\mathcal{S}_\gamma$. It is also possible to compute the input sequences that lead to these number of iterations. To specify sets $\mathcal{S}_\alpha$, $\mathcal{S}_\beta$, and $\mathcal{S}_\gamma$, it is convenient to make use of the control flow predicates given in [29] (cf. section 3).

Hence, we are able to compute the exact minimal and maximal reaction times in terms of macro steps. In particular, we can compute path information like loop bounds, the minimal/maximal number of macro steps required to reach a certain program location from another one, and infeasible paths for a later low-level WCET analysis. Using our algorithms for real-time model checking [22, 23] allows us furthermore to efficiently *verify given path information*.

There is some related work like [9] where similar algorithms for runtime analysis on transition systems have been considered. In contrast to [9], our approach is integrated in a design flow: the transition systems we analyse are obtained from synchronous programs that are used for automatic hardware and software generation. Furthermore, our approach has an obvious interface to a low level analysis that has to determine the runtime of the macro steps.

Alternative integrations of runtime estimation algorithms into the design flow have recently been presented in [6, 11, 30]. In contrast to our approach, these approaches construct a timed automaton that can be used for real-time verification of given constraints. For this purpose, they require however a low-level WCET analysis in advance to determine the physical time necessary to execute a macro step on a particular architecture. While having the same aim, our approach is located at the high-level of the design flow, and therefore independent of a particular architecture. Having computed a high-level WCET analysis in advance, we can then apply the results at different architectures and hardware/software partitions, avoiding repetitions of expensive real-time verification analysis.

The paper is organized as follows: In the next section, we explain the basics of synchronous languages. In Section 3, we then explain the basics of our real-time verification methods as introduced in [22]. Section 3.2 contains the main results of the paper: *Using symbolic state space exploration, we compute for all inputs the lengths of all computations from given program locations $\mathcal{S}_\alpha$ to other program locations $\mathcal{S}_\gamma$.* Beneath computing the execution times, we can also use the real-time verification algorithms given in [22] to verify given path information.

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

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

```
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;
        ℓ : pause
      end while
    end loop
end module
```

Figure 1. Russian Multiplication

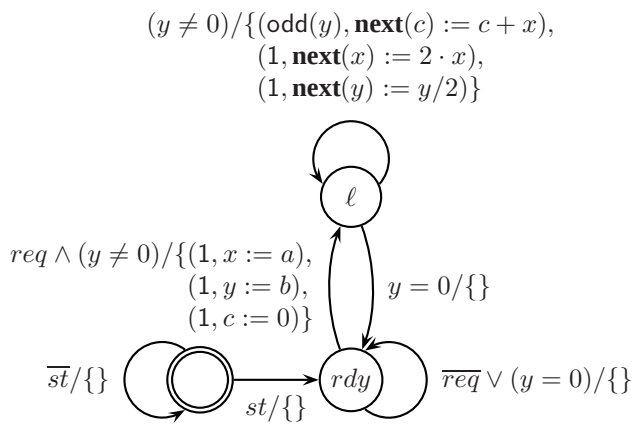

Figure 2. Semantics of module RussMult

2. Synchronous Languages

Synchronous languages [15] like many Esterel-variants [2, 4, 12, 19, 29, 28] 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[3]. 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 **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 α. The manipulations of the variables of a program are performed as micro steps of a macro step. These assignments or signal emissions determine the values of the variables at the current and the next macro step (this may result in so-called *causality problems* [3]). *An important matter of fact for WCET 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 $\mathcal{P}$ can therefore be given as a finite state transition system $\mathcal{A}_\mathcal{P}$: the states of $\mathcal{A}_\mathcal{P}$ reflect the possible combinations of control flow locations of the program (a control flow location is a point in the program text, where the control flow might rest for one unit of time). As the language allows the implementation of parallel threads, there might be more than one current position of the control flow in the program. A transition between two control states is enabled if some condition on the data values is 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.

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

Beneath the comfortable programmer's model given by the macro step abstraction, synchronous languages like our 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 [29, 28] and to the Esterel primer, which is an excellent introduction to synchronous programming [4].

The semantics of Quartz and Esterel can be defined in several ways that lead all to the same transition system. In particular, there is a semantics based on process-algebraic transition rules, and a direct translation into hardware circuits [3]. Recently, we have defined the semantics of a statement S by the following control flow predicates [29]

[3] 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.

and the set of guarded commands $\mathsf{guardcmd}(\varphi, S)$.

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

$\mathsf{instant}(S)$ holds iff the control flow can not stay in S when S would now be started. This means that the execution of S would only execute micro steps.

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

$\mathsf{terminate}(S)$ describes all conditions where the control flow is currently somewhere inside S and wants to leave S.

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

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

For example, for the body statement of module *RussMult* given in Figure 1, we obtain the following results ($\mathsf{X}\varphi$ means that φ holds at the next point of time):

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

Using the above predicates, one can easily define the control and the data flow of a program [29]. *What is more interesting for WCET analysis is that we can describe with these control flow predicates situations that are relevant for gathering path information, as we will explain in the next section.*

3. WCET Analysis of Synchronous Programs

The presentation of the semantics in the form indicated in Figure 2 is the basis of our execution time analysis. However, the key problem in execution time analysis, namely to determine how many transitions can be taken from a state set $\mathcal{S}_\alpha$ to another state set $\mathcal{S}_\gamma$, is still an undecidable problem. However, if we assume that all data types used in the program are finite, then we can compile the program to a classical finite state machine (fsm). Clearly, the obtained fsm will normally suffer from the enormous state explosion. For this reason, we use a symbolic representation in the sense of symbolic state space exploration [5, 8]. Symbolic representations have lead to a breakthrough in the verification of finite-state transition systems [5, 8]. The key idea is thereby that sets are not explicitly stored; instead the characteristic function is represented as a Boolean formula that is itself stored in a canonical normal form (BDDs [7]).

3.1. Verifying Given Path Information

Using a symbolic representation, it is straightforward to compute fixpoints, for example to determine the set of reachable states. Furthermore, arbitrary temporal properties specified in a temporal logic can be verified. In particular, we have defined in [22] an extension called JCTL of the well-known temporal logic CTL by real-time contraints. In contrast to other real-time temporal logics, JCTL is a consequent extention of CTL and therefore still allows the use of already available symbolic model checking algorithms.

Using JCTL, we can describe many interesting temporal properties with constraints on the number of macro steps taken to satisfy a condition. For example, the following are JCTL formulas, provided that φ and ψ were JCTL formulas, and a and b are natural numbers:

- any atomic formula
- $\neg\varphi$ and $\varphi \wedge \psi$
- $\mathsf{EX}^{[a,b]}\varphi$ and $\mathsf{EX}^{\geq a}\varphi$
- $\mathsf{E}[\varphi \,\underline{\mathsf{U}}^{[a,b]}\, \psi]$ and $\mathsf{E}[\varphi \,\underline{\mathsf{U}}^{\geq a}\, \psi]$
- $\mathsf{EG}^{[a,b]}\varphi$ and $\mathsf{EG}^{\geq a}\varphi$

Intuitively, $\mathsf{EX}^{[a,b]}\varphi$ holds in a state s iff s has a direct successor state s' that satisfies φ and can be reached in time $t \in [a, b]$. $\mathsf{EX}^{\geq a}\varphi$ holds in a state s iff s has a direct successor state s' that satisfies φ and can be reached in time $t \geq a$.

$\mathsf{E}[\varphi \underline{\mathsf{U}}^{[a,b]}\, \psi]$ holds in a state s iff there is a path π starting in s and a number $i \in \mathbb{N}$ so that for the first i states of π the property φ holds, and ψ holds on the $(i+1)$-th state of π, and the time t *required to reach* the $(i+1)$-th state stems from the interval $[a, b]$. $\mathsf{E}[\varphi \,\underline{\mathsf{U}}^{\geq a}\, \psi]$ is defined analogously, with the difference that $a \leq t$ has to be satisfied instead $t \in [a, b]$.

$\mathsf{EG}^{[a,b]}\varphi$ holds in a state s iff there is a path π starting in s, such that any state $\pi^{(i)}$ on π that is reached within a time $t \in [a, b]$ satisfies φ. $\mathsf{EG}^{\geq a}\varphi$ is defined analogously.

In [22], a symbolic model checking algorithm for JCTL has been presented that computes for a given JCTL formula Φ and a TKS $\mathcal{K}$ the set of states of $\mathcal{K}$ where Φ holds. Manually given path information like the infeasibility of computation paths can be specified in JCTL, and hence, can be verified by the algorithms given in [22]. To this end, it is

convenient to make use of the control flow predicates that have been discussed in the previous section (see also the following section).

3.2. Determining Path Information

In this section, we show how our tool computes WCET and BCET for a given program. In particular, we explain how the lower and upper bounds of loop iterations can be efficiently calculated. As in the previous section, we assume that we have already compiled the program to a finite state machine.

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

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

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

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

In the first place, we must ensure that the given program is correctly implemented, i.e. that all computation paths starting at $\mathcal{S}_\alpha$ will finally reach $\mathcal{S}_\gamma$. This correctness property can be easily verified by checking the following JCTL property, that states that all computation paths that start in $\mathcal{S}_\alpha$ must finally reach $\mathcal{S}_\gamma$: $\mathsf{AG}^{\geq 0}(\alpha \rightarrow \mathsf{AF}^{\geq 0}\gamma)$.

For the above properties α, β, and γ this means that the loop will terminate for all inputs. The final WCET/BCET analysis together with the calculation of bounds for loop iterations is then performed by the function EHLA *(*E*xact* H*igh*-L*evel* A*nalysis)*, shown in Figure 3. Arguments of the algorithm are the transition relation $\mathcal{U}$ of the fsm, the source set of states $\mathcal{S}_\alpha$, the set of target states $\mathcal{S}_\gamma$, and the

```
function EHLA(U, S_α, S_β, S_γ)
  i := 0;
  T := {};
  bound := 0;
  repeat
    if S_α ∩ S_γ ≠ {} then
      T := T ∪ {i};
      S_α := S_α \ S_γ;
    endif ;
    if S_α ∩ S_β ≠ {} then
      bound := bound + 1;
    endif ;
    S_α := {s' ∈ S | (s, s') ∈ U ∧ s ∈ S_α};
    i := i + 1;
  until S_α := {};
  return (T, bound);
end function
```

Figure 3. Computing minimal and maximal paths from $\mathcal{S}_\alpha$ to $\mathcal{S}_\gamma$ and counting the number of visits in $\mathcal{S}_\beta$.

set of states $\mathcal{S}_\beta$. As an invariant, the algorithm stores in $\mathcal{S}_\alpha$ the set of states that can be reached within i steps from the originally given set $\mathcal{S}_\alpha$. Using a breadth first search, the algorithm successively computes all successor states of the current set $\mathcal{S}_\alpha$. A variable i is used to count the number of macro steps taken so far, and a set $\mathcal{T} \subseteq \mathbb{N}$ stores the lengths of paths from $\mathcal{S}_\alpha$ to $\mathcal{S}_\beta$. Two main things must be checked within the loop:

- If a path reaches the target set of states $\mathcal{S}_\gamma$ (checked by $\mathcal{S}_\alpha \cap \mathcal{S}_\gamma \neq \{\}$), then the current number of macro steps i is added to the set $\mathcal{T}$, and the path is removed from the current set $\mathcal{S}_\alpha$ (so it will not be considered further). The algorithm terminates when the set $\mathcal{S}_\alpha$ is empty, i.e. if no more paths are left. Then, $\mathcal{T}$ will contain lengths of all paths from $\mathcal{S}_\alpha$ to $\mathcal{S}_\gamma$. The minimum and maximum of $\mathcal{T}$ are then the BCET and WCET, respectively. For compact storage, one can also easily represent $\mathcal{T}$ by means of a BDD.
- If some path reaches the set of states $\mathcal{S}_\beta$, then the loop bound variable $bound$ is incremented. At the end of the calculation, $bound$ will deliver exactly how many times $\mathcal{S}_\beta$ can be visited by computations from $\mathcal{S}_\alpha$ to $\mathcal{S}_\gamma$.

Algorithm EHLA given in Figure 3 is our main procedure for runtime analysis. The algorithm works on finite-state

```
module Euclid :
  input a : int[n], b : int[n];
  output x : int[n], y : int[n];
  x := a;
  y := b;
  do
    if x ≥ y then
      next(x) := x − y
    else
      next(y) := y − x
    end;
    ℓ : pause
  while (x ≠ 0) ∧ (y ≠ 0);
  if x = 0 then next(x) := y end;
  rdy : pause
  /* x is the gcd of a and b */
end
```

Figure 4. Euclid's algorithm.

transition systems, but gathers runtime information about the program locations described by α, β, and γ. Previous algorithms like [9, 10] have, in contrast to our approach, no relationship to the program source, and hence, are not able to compute program related properties like the minimal/maximal numbers of loop iterations. Note that the compiler used to generate executable C-code or hardware circuits is the same that is used to compute our fsms. Hence, the overall design work flow assures that our runtime analysis determines the correct information for the later low-level analysis.

4. Experimental Results

We have implemented the algorithms in our tool framework consisting of a compiler for the Quartz language and a BDD-based real-time model checker. In this section, we present experimental result that we have obtained with some benchmarks of the current implementation.

The first example is Euclid's algorithm to compute the greatest common divisor of two given numbers. We have instantiated the Quartz program given in Figure 7 with various numbers for the bitwidth of the numbers n, and obtained the results given in table 1. The column 'possible states' contains the size of the transition system that we obtained from the program, the next column contains the computed WCET in terms of macro steps, the third column the memory requirement of our tool in terms of BDD nodes (one node is 16Bytes), and the fourth row is the runtime of our tool on a Pentium 3 with 1GHz and 512 MBytes of main memory.

b	poss. states	WCET [steps]	memory [nodes]	runtime [min:sec]
4	2^{23}	16	4600	0:00.390
5	2^{28}	32	9253	0:00.890
6	2^{33}	64	26154	0:03.080
7	2^{38}	128	45178	0:06.820
8	2^{43}	256	96740	0:26.160
9	2^{48}	512	296869	2:37.270
10	2^{53}	1024	833931	11:36.510
11	2^{58}	2048	2433534	68:11.360
12	2^{63}	4096	7670831	695:20.780

Table 1. WCET analysis of Euclid's algorithm.

It can be observed that Euclid's algorithm for b bits has a WCET of 2^b macro steps. The crucial macro step is the one that corresponds to the loop body that consists of a subtraction, a comparision and two test on equality to 0. If a low-level analysis for a hardware implementation could tell us that the program for 8 bits can be implemented with a hardware circuit with a clock speed of 40 MHz, then we know that a gcd computation will be done in at least 256/40 10^{-6} seconds, and for 12 bits, the circuit would require about $2^{12}/4010^{-6}$ seconds (≈ 0.1024 milliseconds).

Another benchmark that we have tested was checking primality of a given number. The algorithm is given in Figure 5. The idea is to first check if the number is even, and if not, to divide the number by all odd number starting from 3 up to the half of the number to be tested. As division is hard to implement by a combinatorial circuit, we have chosen a sequential algorithm that requires b steps for a division of two b-bit numbers.

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

The columns are the same as in Table 1, the additional column '# Div.' contains the maximal number of divisions, i.e. loop iterations. As $2^{b-2} - 1$ is 15, 31, and 63 (for 6,7, and 8 bits, respectively), we see that our analytical estimation was too pessimistic. For the WCET, the analytical prediction would yield the numbers 121, 280, and 631 (for 6,7, and 8 bits, respectively) which is also too pessimistic. We see, that our exact bounds are better than the pessimistic

```
module Primality :
  input p : int[n];
  output prime, non_prime;
  local ls, x : int[n − 1] in
    a := p;
    if a[0] then
      weak abort
        x := 3;
        if x < (a div 2) then emit ls end;
        while ls do
          d := a;
          D : run Division()(d, x, m);
          if m = 0 then emit non_prime end;
          ℓ1 : pause;
          next(x) := x + 2;
          ℓ2 : pause;
          if x < (d div 2) then emit ls end
        end
      when non_prime;
      if non_prime then emit prime end
    else
      if a = 2 then emit prime
      else emit non_prime end;
    end;
    tc : pause
  end
end
```

Figure 5. Algorithm to test primality.

b	poss. states	# Div.	WCET [steps]	memory [nodes]	runtime [min:sec]
6	2^{56}	14	113	65749	0:19.700
7	2^{64}	30	271	287387	12:17.960
8	2^{72}	61	611	678930	144:59.560

Table 2. Results for Primality test.

analytical estimation.

A last example we want to mention is Fischer's mutual exclusion protocol [20]. The benchmark consist of n processes that execute the code given in Figure 6. There is a critical section between the locations cs_1 and cs_2, i.e. at most one of the processes is allowed to be in between these locations. The access to the critical region is controlled by a shared variable x: when $x = 0$ holds, the region is free, $x \in \{1, \ldots, n\}$ means that the process with the identifier x is granted access to the region. A process tries to assign its process identifier *pid* to a shared variable x. As the processes are executed in an interleaved manner, there will be no write conflict in doing so, and it may be the case that after having written x, it may no longer have the value *pid* when the process reaches the location $wait_2$.

```
module FischerProcess :
  input running, pid : integer;
  output x : integer;
  suspend
    do
      wait1 : await (x = 0);
      next(x) := pid;
      wait2 : pause
    while (x ≠ pid);
    cs1 : pause;
    /* critical section */
    cs2 : pause;
    x := 0;
  wait3 : halt
  when running
end
```

Figure 6. A process in Fischer's Protocol.

n	poss. states	WCET [steps]	memory [nodes]	runtime [min:sec]
5	2^{34}	25	5953	0:00.950
10	2^{65}	45	34816	0:08.770
15	2^{95}	65	59714	0:22.170
20	2^{126}	85	116618	0:28.930
25	2^{156}	105	225928	3:41.430
30	2^{186}	125	281271	9:22.980

Table 3. WCET analysis of Fischer's protocol.

We have obtained the results given in Table 3 for n processes. In particular, we have determined the the largest number of macro steps it may take for a process to complete its task, i.e. to reach location $wait_3$. The example shows that our tool is able to examine the runtime of even large systems that consist of a large number of processes. According to our experimental results, we might guess that the WCET of a system with n processes is $4n + 5$, which would not be so simple to obtain analytically.

5. Conclusions

A novel approach to analyse execution times of synchronous programs has been presented that is able to com-

pute for all input sequences the number of macro steps that are executed between given program locations α and γ. In particular, using control flow predicates [29], we can determine α and β such that the algorithm can be used to compute exact bounds of loop iterations. The overall approach does not depend on a particular architecture; its results can be used for different architectures if these are determined in later design phases.

References

[1] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *IFAC Workshop on Real-Time Programming*, 2000.

[2] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[3] G. Berry. The constructive semantics of pure Esterel, July 1999.

[4] G. Berry. The Esterel v5_91 language primer. June 2000.

[5] C. Berthet, O. Coudert, and J. C. Madre. New ideas on symbolic manipulations of finite state machines. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 1990.

[6] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys = Esterel + Kronos. A tool for verifying real-time properties of embedded systems. In *Conference on Decision and Control (CDC)*, Orlando,USA, 2001. IEEE Control Systems Society.

[7] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[8] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computing*, 98(2):142–170, June 1992.

[9] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems*, 1995.

[10] S. Campos and O. Grumberg. Selective quantitative analysis and interval model checking: Verifying different facets of a system. In R. Alur and T. A. Henzinger, editors, *Conference on Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 257–268, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[11] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. TAXYS: A tool for the developpment and verification real-time embedded systems. In *Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, Paris,France, 2001. Springer Verlag.

[12] ECL Homepage. Website. http://www-cad.eecs.berkeley.edu/

[13] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *International European Conference on Parallel Processing (EuroPar)*, volume 1300 of *LNCS*, pages 1298–1307, Passau, Germany, 1997. Springer Verlag.

[14] E. Erpenbach, F. Stappert, and J. Stroop. Compilation and timing analysis of statecharts models for embedded systems. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Wyndham City Center, Washington, D.C. USA, October 1999. ACM.

[15] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, 1993.

[16] T. Halfhill. Embedded market breaks new ground, 2000. Microprocessor Report.

[17] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engenieering Methods*, 5(4), 1996.

[18] C. Healy, R. van Engelen, and D. Whalley. A general approach for the tight timing predictions of non-rectangular loops. In *IEEE Real-Time Technology and Applications Symposium*, 1999.

[19] Jester Home Page. Website. http://www.parades.rm.cnr.it/projects/jester/jester.html.

[20] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 1987.

[21] Y. Liu and G. Gomez. Automatic accurate time-bound analysis for high-level languages. In *Languages, Compilers and Tools for Embedded Systems (LCTES)*, 1998.

[22] G. Logothetis and K. Schneider. A new approach to the specification and verification of real-time systems. In *Euromicro Conference on Real-Time Systems*, pages 171–180, Delft, The Netherlands, June 2001. IEEE Computer Society.

[23] G. Logothetis and K. Schneider. Extending synchronous languages for generating abstract real-time models. In *European Conference on Design, Automation and Test in Europe (DATE)*, Paris, France, March 2002. IEEE Computer Society.

[24] T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Languages, Compilers and Tools for Embedded Systems (LCTES)*, 1998.

[25] Micro Design Ressources. Embedded processor forum. http://www.mdronline.com.

[26] P. Puschner. Worst-case execution time analysis at low cost. In *Distributed Computer Control Systems*, pages 16–21, Seoul, Korea, 1997.

[27] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1(1):159–176, 1989.

[28] K. Schneider. A verified hardware synthesis for Esterel. In F. Rammig, editor, *International IFIP Workshop on Distributed and Parallel Embedded Systems*, pages 205–214, Schloß Ehringerfeld, Germany, 2000. Kluwer Academic Publishers.

[29] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *International Conference on Application of Concurrency to System Design (ICACSD 2001)*, pages 143–156, Newcastle upon Tyne, UK, June 2001. IEEE Computer Society Press.

[30] R. Shyamasundar and J. Aghav. Realizing real-time systems from synchronous language specifications. In *Real Time Systems Symposium, Work in Progress Session*, Orlando, Florida, USA, November 2000. IEEE.