Towards Interactive Timing Analysis for Designing Reactive Systems

Insa Fuhrmann
David Broman
Steven Smyth
Reinhard von Hanxleden

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-26
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-26.html

April 3, 2014
Acknowledgement

This work was supported in part by the the PRETSY project (supported by the German Science Foundation DFG HA 4407/6-1 and ME 1427/6-1), the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies), the Swedish Research Council (#623-2011-955), and the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley (supported by the National Science Foundation, NSF awards #0720882 (CSR-EHS: PRET), #1035672 (CPS: Medium: Timing Centric Software), and #0931843 (ActionWebs), the Naval Research Laboratory (NRL #N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota).
Towards Interactive Timing Analysis for Designing Reactive Systems

Insa Fuhrmann¹, David Broman²³, Steven Smyth¹, and Reinhard von Hanxleden¹

¹ Christian-Albrechts-Universität zu Kiel, Germany
² University of California, Berkeley, CA, USA
³ Linköping University, Sweden

Abstract. Reactive systems are increasingly developed using high-level modeling tools. Such modeling tools may facilitate formal reasoning about concurrent programs, but provide little help when timing-related problems arise and deadlines are missed when running a real system. In these cases, the modeler has typically no information about timing properties and costly parts of the model; there is little or no guidance on how to improve the timing characteristics of the model. In this paper, we propose a design methodology where interactive timing analysis is an integral part of the modeling process. This methodology concerns how to aggregate timing values in a user-friendly manner and how to define timing analysis requests. We also introduce and formalize a new timing analysis interface that is designed for communicating timing information between a high-level modeling tool and a lower-level timing analysis tool.

Keywords: Worst-Case Execution Time, Reactive Systems, Worst-Case Reaction Time, Synchronous Languages, Precision Timed Machines, Sequential Constructiveness

1 Introduction

Cyber-physical systems (CPS) [12], such as automobiles and aircraft, include a large number of embedded reactive systems. Such systems typically interact with the physical environment by sensing, performing computations, and actuating output data. Reactive systems are increasingly designed with the help of

* This work was supported in part by the the PRETSY project (supported by the German Science Foundation DFG HA 4407/6-1 and ME 1427/6-1), the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies), the Swedish Research Council (#623-2011-955), and the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley (supported by the National Science Foundation, NSF awards #0720882 (CSR-EHS: PRET), #1035672 (CPS: Medium: Timing Centric Software), and #0931843 (ActionWebs), the Naval Research Laboratory (NRL #N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota).
high-level modeling tools, where real-time is not part of the model abstraction. This separation of functional and timing concerns enables formal reasoning and provides a basis for determinism, as provided for example by synchronous languages [3]. Although such strict separation of concerns is valuable, it limits the modeler’s control and ability to reason about execution time of the modeled system. In particular, modeling languages and tools provide little help for the modeler to determine where the time consuming parts are in the model. These “hot spots” often contribute significantly to the execution time, but account only for small sections of the model [4]. Moreover, state-of-the-art modeling tools offer little guidance on how to improve timing and design of the model.

To improve the situation, a tool chain may be augmented with interactive timing analysis capabilities. This means that the modeler gets direct feedback concerning execution time for the whole or parts of the system. For a model of a reactive system, the most essential timing information is the worst-case reaction time (WCRT) [5], meaning the maximal time allowed to read sensor input, perform computation, and actuate output. Such an input-output cycle is also referred to as a tick, and the computation is performed in a tick function. The interactive timing analysis problem concerns computing the worst-case execution time (WCET) [6] of parts of such a tick function and propagating the timing information back to the user at the level of abstraction of the modeling language.

Although there exists a large body of work in the area of WCET analysis [7,8,9,10,11], surprisingly little has been done with interactive timing analysis. Previous work addresses fast WCET analysis [12], interactive C code analysis [13], analysis of Java code [14] and timing analysis of Matlab/Simulink models [15]. Similar to previous work, we study the interactive timing analysis problem in the context of a concrete modeling language: SCCharts [16], a visual language based on the synchronous model of computation. However, in contrast to previous work on interactive timing analysis, we describe a generic approach with a formal interface between a modeling tool and a timing analysis tool.

More specifically, this work-in-progress paper addresses several technical challenges. The first problem concerns classification of WCET information and aggregation of timing values from hierarchical models. Aggregation of timing information is particularly difficult when there is no direct mapping between elements at different levels of abstraction in the tool chain. The second problem concerns efficiency of the WCET analysis itself. The actual computation time of the WCET analysis must be short and responsive, so that the user can interactively use the timing information when developing a model. In contrast to previous work on efficient WCET analysis [12], our approach advocates separation of concerns between the analysis of the tick function and of functions it calls.

Contributions:

– We propose a design methodology where detailed interactive timing analysis information is an integral part of the design processes. We introduce a classification for timing values that defines the categories deep and flat for
hierarchical model elements as well as the classes of *fractional* and *local* timing values according to different methods of value aggregation (Sec. 2).

- We present work-in-progress of a tool chain, where timing information is computed and propagated between various levels—from machine code level up to high level modeling languages. We also propose the concept of *timing program points*, specific annotations that are used for passing timing analysis requests between optimization and translation phases (Sec. 3).

- We formally define a *timing analysis interface* that establishes interactive communication between modeling and timing analysis tools. The interface allows to separate the concerns of timing analysis for external function calls and for the tick function, thus supporting efficiency, as the WCET of external functions can be computed outside of the interactive timing analysis loop. We present a formal interface description that also leverages the concept of timing program points (Sec. 4).

Our work is positioned in comparison to related work in Sec. 5 and we give an outlook on planned future work in Sec. 6.

![Model Diagram](image)

**Fig. 1.** An example of a small robot model, in which the timing values (right upper edge of region, flat WCET / deep WCET) change with the model revision.
2 Interactive Timing Analysis from the User Perspective

In this section we motivate our proposal for interactive timing analysis in reactive system design from the user perspective.

Fig. 1(a) shows a simple robot control model expressed in SCCharts [16]. The robot drives, takes images, stops upon hitting an obstacle, and then writes a log file. The SCChart contains three different threads of execution, each expressed as a region of the model. Regions are depicted as white rectangles, with the region name in the upper left corner. The main thread initializes the output variable and is then in state Driving or Stopped. Driving is a superstate that contains two regions (threads): region HandleEmergency sets the boolean flag stop if the flag bumper indicates hitting an obstacle; concurrently, region HandleMotor first calls getImage and enters the Ready state. When stop turns true, it writes the log file and moves to the Stop state; alternatively, when the accelerator pedal is pressed, it moves to Start and starts over. Note that we assume no true parallelism in the execution of concurrent threads for our example implementation, but our timing analysis interface is not limited in that respect.

Our proposed interactive timing analysis augments regions with timing annotations, shown in the upper right corner. In our prototype, they have the format $\langle$Flat WCET$\rangle / \langle$Deep WCET$\rangle$, with the following meanings.

Flat timing values denote the WCET (BCET) of a region, but the execution time of enclosed regions is not included in this value. In the robot example this means that the flat timing value for main will include the execution time for initialization and termination, but not the execution time for the threads HandleEmergency and HandleMotor.

Deep timing values for a region on the contrary take the included regions into account. The execution time of the regions HandleEmergency and HandleMotor belong to the deep execution time value of region main.

In our prototype, the values are given as generic time units (more precisely, elements of $\mathbb{N}_+$, see Sec. 4.2), abbreviated $tu$, with their interpretation depending on processor clock rates. For fixed clock rates, they may also indicate concrete physical time units, such as $\mu$sec.

The WCRT of the robot model, meaning the maximal time between reading sensor values (such as bumper) and writing actuators (such as motor), corresponds to main’s deep WCET. For reactive systems, the maximal permitted WCRT is typically part of the design specification. In the following, let’s assume for example that the WCRT must not exceed 400 $tu$. The actual WCRT of the model version 1 in Fig. 1(a) exceeds this constraint by 130 $tu$. However, this value alone does not help in locating the most costly parts of the model, the “hot spots” where the revision of the model should start.

Here the modeler is guided by the more detailed values for individual regions. The flat WCET for Main, 30 $tu$, is so small that a reduction would not make a sufficient difference. However, the WCET of HandleMotor alone (410 $tu$) exceeds the timing specification. Now the modeler may revise the model to version 2 shown in Fig. 1(b), where getImage() and writeLog() will not be executed in the same
tick anymore. The automatically updated timing annotations confirm success; the WCRT is now 330 tu.

In addition to the distinction of timing values into deep and flat, we propose the following classification, which we explain with the help of the robot model version 2.

**Fractional** WCET or BCET of a model element denote its share of the overall WCET or BCET of the model. For a region, this is the execution time cost of the part of the critical (or least critical for BCET) path that lies in the region. In our example, suppose the critical path corresponds to the case that the input from the bumper sensor is true. Then the part of the critical path that lies in region **HandleMotor** means taking the transition with the priority 1, which is active, when **stop** is true. This transition will call **writeLog()**, which costs 180 tu; with some additional execution time for the conditional, say 30 tu, this yields 210 tu as fractional WCET for **HandleMotor**.

**Local** WCET or BCET of a model element is the cost of the most (or least for BCET) costly execution time path that lies in this element. In **HandleMotor**, the transition with the priority 2, which is not part of the overall WCET path, calls **getImage()**, which costs 200 tu; with some additional execution time, here 40 tu, this results in 240 tu. This local WCET path of **HandleMotor** is not part of the overall WCET path, but more costly than the path including the transition with priority 1. The corresponding WCET of 240 tu is the local WCET of **HandleMotor**.

Both timing values are zero if we can determine that the model element will never be executed. For a formal definition of these terms, please see Section 4.

The two types of timing values serve different purposes. The fractional WCET highlights the critical path, as a model element that is not on the critical path has the fractional WCET zero. The local WCET values indicate the potential cost of a model element even if it is not on the critical path in the current model context. Our current tool prototype shows fractional WCETs, this could be replaced by or augmented with local WCETs.

### 3 Timing Information Chain and Program Points

In this section we explain the concept of timing information flow along the chain of compilation steps. We address the tracing of model element representations during model transformations, and how program points allow to link the structure of the original model to generated code.

#### 3.1 The Timing Information Chain and Model Element Tracing

We have to trace model element representation for all compilation steps, and we must be able to map back timing values. Fig. 2 shows the timing information flow for our work-in-progress tool chain. Though the concrete choice of modeling language and processor has an effect on efficiency and tightness of interactive analysis, the proposed timing information propagation concept is generic.
In the following we discuss the different compilation steps.

1) **Model transformation in the same metamodel:** Transforming a model to its representation in a simple kernel subset of the modeling language, for example from Extended SCCharts to Core SCCharts \[16\], eases further compilation and is easy to trace, as the set of structural elements is reduced and the model of computation is kept.

2) **Model transformation from one metamodel to another:** Even transformations with source and target models that belong to different metamodels can be easy to trace if the chosen intermediate representation adheres to the same model of computation, and its structural description matches as directly as possible the kernel structure set of the modeling language. This is the case for SCCharts and its intermediate representation, the SC Graph (SCG) \[16\]. Our prototype traces the model element representation relations with every transformation. This mapping allows us to determine which elements of the final model belong to the regions in the original model.

3) **Host code generation:** Usually the next step is the generation of host language code. For our tool chain this is C code. To convey our preserved model element mapping to the timing analysis tool, we propose to insert *(timing)* program points *(TPP)* in the code between any two statements that belong to different threads (see also Sec. 3.2). For our concrete implementation this is simple, as the code synthesis preserves the structure of the SCG.

4) **Translation to an intermediate low level language:** We propose to use an intermediate low level language, for example LLVM assembly. This shifts the timing analysis for specified code segments to an intermediate level and separates it from low level basic block timing analysis. We also propose to use an extended intermediate format that maintains the inserted program points.

5) **Translation to machine code:** Step 4 is followed by a normal translation to machine code, for which a basic block timing analysis is performed. We here use the FlexPRET processor \[17\] with RISC-V ISA. The approach presented in this paper is not specific to PRET architectures; however, the more “compositional” the timing properties of the target architecture are, the more accurate is the thread-specific timing information we convey to the user. In that respect, it is very desirable if the timing properties for some sequence of instructions are mostly independent from their execution context (*e.g.*, cache and pipeline states), as is the case for the FlexPRET.
Fig. 3. The tick function for the improved robot example with inserted and implicit timing program points.

3.2 Timing Program Points

Fig. 3 shows the tick function for the SCChart of Fig. 1(b). The code is divided into a number of scheduling blocks. Their execution depends on boolean guards, prefixed with `g`. The scheduling logic is less important here (we refer the interested reader to [16]), essential is only that the different threads can be mapped to scheduling blocks as marked in the listing. To retrieve timing information on thread level, we insert timing program points wherever one statement belongs to another thread than its predecessor, see again Fig. 3. There are implicit program points at the start and the end of the tick function, called entry, short `p_e`, and exit, short `p_x`. Each explicit program point has a unique number. We specify a list of program point pairs to convey for which parts of the code we request timing information. Any pair of program points can be listed, but analysis will yield the unknown value, if there is no controlflow path from the first to the second point. The pairs are passed to timing analysis in a file that can include information about desired time value types and about assumptions specified by the modeler.

4 Timing Analysis Interface

In this section we formally define a timing analysis interface used for communicating timing information between a high-level modeling tool and a timing analysis tool.

4.1 Interface Formalization

The interactive timing analysis problem can be defined as follows.
Definition 1 (Interactive Timing Analysis). Given a program consisting of a set of functions $F$, a set of global variables $G$, and a timing analysis request $t_{req}$, return a timing response $t_{res}$.

By function we mean a function in the sense of the C language, although the problem formulation itself is not limited to C. Global variables may be of any primitive type (e.g., integer or floating point) and be given initial values.

The essential component of the timing analysis interface is the syntax and semantics of the timing analysis request $t_{req}$ and the timing response $t_{res}$. More specifically, a timing analysis request is a 6-tuple

$$t_{req} = (f, a, g, e, P, R).$$

The first element $f \in F$ of the tuple is the function to be analyzed; $a$, $g$, and $e$ state assumptions for the analyzer; $P$ is the set of timing program points in function $f$; $R$ is the set of requested analyses. We now detail the assumptions ($a$, $g$, $e$), followed by the program points ($P$) and analyses requests ($R$).

4.2 Assumptions

The assumptions $t_{req}$ may be used by the timing analyzer to compute tighter execution bounds. For instance, if only a specific set of values can be supplied as arguments to function $f$, the analyzer may exclude infeasible paths, thus providing tighter WCET or BCET. These assumptions are optional; by not providing assumptions, the analysis may have to be more conservative.

Consider now again elements two to four of $t_{req}$. Assumption $a : \mathbb{N} \rightarrow A$ is a function that specifies assumptions for the arguments that may be applied to function $f$. That is, expression $a(n)$ returns, for argument $n \in \mathbb{N}$, an abstract value $v \in V_a$. In this formalization, we do not specify which abstract domain value $v$ should be in, but for an integer type, a typical value could be represented as an integer interval. Similarly, function $g : G \rightarrow V_a$ specifies the assumption for a value $g(x)$ of a global variable $x \in G$. Finally, function $e : F \rightarrow \mathbb{N}_{\perp} \times \mathbb{N}_{\perp}$ specifies assumptions on execution time for functions that may be called by $f$.

More specifically, for a function $f_1 \in F$, $e(f_1)$ denotes a tuple $(t_b, t_w)$, where $t_b$ and $t_w$ specify the assumptions of safe lower and upper bounds of the execution time for $f_1$, respectively. Note that we represent execution time as a natural number that includes elements $\perp$ and $\epsilon$, that is $\mathbb{N}_{\perp} = \mathbb{N} \cup \{\perp\} \cup \{\epsilon\}$. Element $\perp$ indicates that the function is non-terminating and $\epsilon$ that a safe bound cannot be or has not been determined. For instance, if $e(f_1) = (200, \perp)$, we can assume that 200 is a safe lower bound, but that it does not exist any safe upper bound because at least one path in the function is infinite (the function does not terminate for some input).

4.3 Timing Program Points and Analyses Requests

The objective of $t_{req}$ is to specify precisely what timing information the high-level modeling tool is interested in. To enable more precise specification than only at
a function level, the modeling tool can insert timing program points within a function. Using pairs of these program points, the tool can then request timing analysis information about parts of the function.

Element $P$ of the timing request tuple $t_{req}$ specifies a set of timing program points. A tool may specify any finite number of timing points, but two program points $p_e$ and $p_x$ representing the function entry point and exit point, respectively, are always part of $P$.

Set $R$ specifies the requested analyses. Each element of $R$ is a triple $(y, p_a, p_b)$, where $y \in Y$ is the type of requested analysis value, $p_a \in P$ the starting program point for the analysis, and $p_b \in P$ the ending point. There are six possible types of requests:

$$Y = \{\text{WCP, BCP, LWCET, LBCET, FWCET, FBCET}\}.$$  

WCP and BCP stand for worst-case path and best-case path, respectively. These are the execution paths between timing program points that result either in longest or shortest timing bound. The other four types request the worst-case execution time (LWCET and FWCET) and best-case execution time (LBCET and FBCET). The prefixes L and F stands for local and fractional, respectively (see Sec. 2 for an informal description). The precise meaning of the different timing requests are defined next.

### 4.4 Timing Response

The timing response $t_{res}$ for a specific timing request $t_{req}$ is a function

$$t_{res} : R \rightarrow \mathbb{N}_{\epsilon} \cup \mathcal{P}(\bar{p})$$

where $r \in R$ is an analysis request and the resulting value is either an execution time value $t \in \mathbb{N}_{\epsilon}$ (for $r \in \{\text{LWCET, LBCET, FWCET, FBCET}\}$ or $r \in \{\text{WCP, BCP}\}$ for an undefined or infinite path) or a finite path $\bar{p} = \langle p_1, p_2, \ldots, p_n \rangle$ (for $r \in \{\text{WCP, BCP}\}$).

By $\mathcal{P}(\bar{p})$ we mean the set of all possible finite paths.

We now formalize the meaning of the different types of timing requests. Let $G = (V, E)$ be a directed graph, representing an extended control-flow graph (CFG) for a function $f$ that is being analyzed. The set of vertexes $V = B \cup P$ is the union of basic blocks $B$ and timing program points $P$ in $f$. From $G$, we can derive a timing program points graph $G_p = (P, E_p)$, where all vertices are program points, and edges $E_p = \{(v, w) \mid v, w \in P$ and $w$ is reachable from $v$ in $G\}$. We require that $G_p$ is a directed acyclic graph (DAG) with the motivation that timing estimates should have a simple clear meaning. Note, however, that $G$

1 As a consequence, timing program points cannot exist inside loops. Although enabling timing program points inside loops would make the interface more expressive, its formal meaning becomes significantly more complicated. For instance, what is the meaning of computing the FWCET between two points inside a loop, where the order that the program points are visited is different depending on input to the main function? That is, between which points do we compute the WCET if the trace for one input is $p_1, p_2, p_2, p_1, p_1$, and for another input $p_2, p_2, p_1$? We consider this interesting problem as future work.
A path in $G$ of length $n$ is a sequence of vertices $(v_1, v_2, \ldots, v_n)$. $T(\bar{p})$ denotes a path in $G_p$ that is derived by removing all vertices $v \notin P$ from $\bar{p}$. Because $G_p$ is required to be a DAG, $T(\bar{p})$ is a simple path (all vertices are distinct).

We assume there exist functions $c_w: E \to \mathbb{N}$ and $c_b : E \to \mathbb{N}$ stating the worst-case and best-case execution times for executing block $v \in V$ and transition to block $w \in V$, where $(v, w)$ is an edge in $E$. If $v$ is a basic block and contains function calls, the timing analysis tool should use function $c$ (assumptions of execution time for function calls) as defined in (1). The execution time is always zero for an edge that leaves from a timing program point vertex. We also assume functions $c_w$ and $c_b$ are non-negative, and if $v$ and $w$ are both timing program points, the transition time $e_{v,w}$ is equal to 1. In general, edges $(v, w)$ are not necessarily timing program point vertices.

Let $\mathcal{F}_{p_1, p_2}(\bar{p})$ be a subpath of $\bar{p}$, that contains the contiguous sequence of vertices between and including $p_1$ and $p_2$. That is, for a sequence $\bar{p} = \langle v_1, v_2, \ldots, v_n, v_{n+1}, v_{n+2}, \ldots, v_{n+k} \rangle$, $\mathcal{F}_{p_1, p_2}(\bar{p}) = \langle v_1, v_2, \ldots, v_{n+k} \rangle$. If $\bar{p}$ does not contain $p_1$ or $p_2$, the empty path is returned, even though $v_1$ and $v_{n+k}$ may still contain elements of $\bar{p}$. If $\bar{p}$ is equal to the empty sequence, then $\mathcal{F}_{p_1, p_2}(\bar{p})$ is returned, respectively. Note that the path is a simple path (all vertices are distinct), such subpaths are uniquely defined.

The worst-case execution time is defined as $E(\bar{p}) = c_w(v_1, v_2) + \ldots + c_w(v_{n-1}, v_n)$, where $\bar{p} = \langle v_1, v_2, \ldots, v_n \rangle$ is a path in $G$. The best-case execution time is defined in the same way using $c_b$. If $\bar{p}$ is the empty path, $E(\bar{p}) = 0$. We also define $E(\perp) = 1$ and $E(\epsilon) = \epsilon$. The timing response time function is then defined as follows:

$$t_{\text{res}}(r) = \begin{cases} T(\bar{v}_w) & \text{if } r = (\text{WCP}, p_1, p_2) \\ T(\bar{v}_b) & \text{if } r = (\text{BCP}, p_1, p_2) \\ E(\bar{v}_w) & \text{if } r = (\text{LWCET}, p_1, p_2) \\ E(\bar{v}_b) & \text{if } r = (\text{LBCET}, p_1, p_2) \\ E(\mathcal{F}_{p_1, p_2}(\bar{v}_w)) & \text{if } r = (\text{FWCET}, p_1, p_2) \\ E(\mathcal{F}_{p_1, p_2}(\bar{v}_b)) & \text{if } r = (\text{FBCET}, p_1, p_2) \end{cases}$$

The different requested execution times are best illustrated using an example. Consider Fig. 4 that shows a CFG for a function $f$. The graph has seven basic blocks ($b_1$ to $b_7$) and six timing program points $\{p_c, p_1, p_2, p_3, p_4, p_2\}$ (shown in small filled circles). In the example, we assume that the timing analysis tool determines that the loop between $b_5$ and $b_6$ is executed at most ten times and at least two times (either by using external flow facts or by computing the loop bounds). We may then observe the following:
Fig. 4. An extended control flow graph (CFG) that includes basic blocks (b₁ to b₇) and timing program points (pₑ, p₁, p₂, p₃, p₄, and pₓ). White arrows represent true-branches and black arrows false-branches and unconditional branches. The graph is cyclic (nodes b₅ and b₆), but a derived graph between timing program points is acyclic.

- If the timing analysis tool cannot handle infeasible paths, the true-branches at b₁ and b₄ are in worst-case taken. Consequently, $t_{\text{res}}(\text{WCP}, pₑ, pₓ) = (pₑ, p₁, p₂, p₃, pₓ)$ and $t_{\text{res}}(\text{LWCET}, pₑ, pₓ) = 10 + 200 + 5 + 15 + 10 \times (100 + 20) + 20 + 5 = 1455$. Note that LWCET and FWCET are the same when computed for the whole function.

- If the timing analysis tool can handle infeasible paths, the tool can detect that the path containing b₂ and b₅ is infeasible because variables x and y cannot be equal and not equal at the same time (assuming x and y in the if-expressions are not modified in basic blocks b₁, b₂, b₃, or b₄.). As a consequence, $t_{\text{res}}(\text{LWCET}, pₑ, pₓ) = 10 + 5 + 15 + 10 \times (100 + 20) + 20 + 5 = 1255$ (the false-branch is taken at b₁ and the true-branch is taken at b₄).

- If the tool handles infeasible paths, $t_{\text{res}}(\text{LWCET}, pₑ, p₁) = 10 + 200 + 5 = 215$, but $t_{\text{res}}(\text{FWCET}, pₑ, p₁) = 10 + 5 = 15$. Note that fractional WCET states how much the path between pₑ and p₁ contributes to the global WCET between pₑ and pₓ. Because the tool is assumed to handle infeasible paths, the longest global path contains edge (b₁, b₃). Local WCET does not consider the global analysis.

- Note that $t_{\text{res}}(\text{FBCET}, p₂, p₃) = 2 \times (100 + 20) + 20 = 260$, but $t_{\text{res}}(\text{FBCET}, p₂, p₄) = 0$; the latter because there is no feasible path between p₂ and p₄ (the path returned by $t^b_{pₑ,pₓ}$ does not contain p₄).

5 Related Work

During the last decades, a significant amount of work has been done in the area of WCET analysis. This includes, for instance, loop and infeasible paths analyses [7,8], cache analysis [9], and path enumeration techniques [10]. There exist several commercial tools, such as aiT² and Bound-T³ as well as academic

² http://www.absint.com/
³ http://www.bound-t.com/
implementations, such as SWEET\textsuperscript{4} from Mälardalen University. A comprehensive overview of the research field is given by Wilhelm et al. \textsuperscript{6}.

To overcome the complexity of analyzing unpredictable hardware, several research groups have been developing predictable hardware platforms. The PRET initiative \textsuperscript{18,19} focuses on both compilers and hardware to achieve tight bounds on WCET. Previous work exists on, for instance, ARM based predictable processors \textsuperscript{20} and Java processors \textsuperscript{21}. Our current work-in-progress uses FlexPRET \textsuperscript{17}, a processor platform designed for mixed-criticality systems. Special WCET analysis techniques have been designed for these kinds of predictable hardware platforms. For instance, Kim et al. \textsuperscript{22} show how Scratchpad memory management can be combined with WCET analysis and Schoeberl et al. \textsuperscript{23} present how to perform WCET analysis on Java processors. Falk and Lukuciejewski \textsuperscript{11} present a compiler for improving WCET and Seshia and Rakhlin \textsuperscript{24} develop methods for performing WCET analysis in a game theoretic setting.

Most work within the WCET analysis area focus on techniques for computing safe and tight bounds of WCET; less attention has been given to how such techniques fit in a development environment and only a few attempts exist on performing interactive timing analysis. Harmon et al. \textsuperscript{12} have developed a tool chain for interactive WCET analysis, where the performance of the analysis time is favored before tightness of the WCET bound. Their approach is implemented in a Java development environment, where the user can obtain timing values directly on functions and program statements. Instead of using the IPET \textsuperscript{10} (the most commonly used path enumeration technique), they use a tree based approach that does not handle infeasible paths. Kirner et al. \textsuperscript{15} show how interactive timing analysis can be incorporated in the Matlab/Simulink environment. They describe how start and stop markers are inserted into C code, but do not give a precise formal meaning. Persson and Hedin \textsuperscript{25} present an interactive timing environment for Java, where WCET analysis is performed at the byte-code level. Ko et al. \textsuperscript{13} have developed an interactive timing analysis environment for C programs, where portions of the program can be selected and analyzed. Bernat et al. \textsuperscript{4} show how a commercial tool RapiTime can be used to identify worst-case hot-spots. The RapiTime tool only uses static analysis for program analysis and makes use of measurements to obtain execution time numbers when executing on the processor. The tool aiT is integrated into SCAD\textsuperscript{5} for timing feedback, but to our knowledge there is no publication of a general formal interface.

In all above described previous work on interactive timing analysis, the focus has been to develop interactive and efficient analysis techniques for specific environments. By contrast, our work in this paper focuses on the interface for interactive timing analysis between a high level tool and the timing analysis tool. In particular, none of the related work formalizes this interface and discusses the difference between different kinds of WCET/BCET values, as discussed in this paper.

\textsuperscript{4} http://www.mrtc.mdh.se/projects/wcet/sweet/
\textsuperscript{5} http://www.esterel-technologies.com/products/scade-suite/
There have also been some investigations of WCET/WCRT analysis for synchronous languages, for example for Esterel by Bertin et al. [26] and for the Esterel-like Quartz language by Logothetis et al. [27,28]. Boldt et al. [29] propose a flow-based method for WCRT analysis of Esterel programs running on a reactive processor with fully predictable timing [30]; the WCRT analysis results are then used to annotate a program with timing annotations that the processor uses to check timing overruns due to hardware failures. Mendler et al. [31] propose an algebraic approach for the WCRT analysis for Esterel programs running on a reactive processor. Raymond et al. [32] propose an approach to improve WCET analysis by checking feasibility of computed longest paths using model checking at the modeling level. Want et al. propose an ILP-based approach that exploits concurrency explicitly in the ILP formulation to avoid state space explosion [33]. These techniques could be combined with our proposal regarding WCET-feedback at the modeling level. Perhaps closest in spirit to our work is the work by Ju et al. [34], which back-annotate an Esterel program with information regarding the timing-critical path. However, they don’t break down specific timing information as we propose here. Our current usage of timing program points is related to the control points of the Saxo-RT compiler [35] in that both indicate possible context switches; however, control points are finer grain since they also express scheduling properties within a thread, not only across threads.

6 Conclusion

In this paper, we have briefly described a methodology for interactive timing analysis as well as introduced a formal timing analysis interface that connects high-level modeling tools to execution time analysis tools. The interface has the potential of enabling efficient interactive timing analysis loops because it separates the concerns of analysis of tick functions from WCET analysis of external functions. We have also discussed our work-in-progress of a tool chain that compiles SCCharts models into RISC-V machine code and performs interactive timing analysis. This tool chain enables us to provide timing feedback not only for the model as a whole, but also for specific model elements.

As for future work, we plan to experimentally study which types of timing information and at what granularity timing feedback are most valuable for the modeler. Additionally, we intend to explore a model checking problem, where the user wants to improve execution time and at the same time guarantee the preservation of functional behavior.

References


13