This is the accepted version of a paper presented at 24th International Conference on Real-Time Networks and Systems (RTNS).

Citation for the original published paper:

https://doi.org/10.1145/2997465.2997467

N.B. When citing this work, cite the original published paper.

© Owner/Author | ACM 2016. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in In Proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS 2016), http://dx.doi.org/10.1145/2997465.2997467.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-199041
ABSTRACT

Modeling tools typically provide no information about timing properties and costly parts of the system under development. In this paper, we propose a generic approach to integrate timing analysis and modeling tools. This approach includes visual highlighting to guide the user to worst-case execution time hotspots, detailed timing information for specific model elements, and the separation of different types of timing values. Our solution includes both a way to keep track of model elements subject to timing analysis during the compilation process, and a flexible and formally defined timing analysis interface for communicating timing information between a high-level modeling tool and a lower-level timing analysis tool. We present a complete open-source, Eclipse-based prototype tool chain that is evaluated both using a systematic benchmark suite and a user study.

1. INTRODUCTION

Cyber-physical systems (CPS) [18], 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 high-level modeling tools, where real-time is not part of the model abstraction. Such separation of concerns on the one hand is valuable as it facilitates formal reasoning and determinism, as leveraged for example by the synchronous languages [2]. On the other hand, it limits the modeler’s control and ability to reason about execution time of the modeled system. This is often problematic as embedded reactive systems are typically real-time systems, where

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

RTNS ’16, October 19 - 21, 2016, Brest, France
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4787-7/16/01...$15.00
DOI: http://dx.doi.org/10.1145/2997465.2997467

correctness not only depends on functional behavior, but also on the timeliness of computations. The typical lack of timing abstractions makes it hard for the modeler to locate the time consuming parts of the model. These “hot spots” often contribute significantly to the execution time, but account only for small sections of a model [3].

In this paper, we introduce an interactive timing analysis method together with a complete tool chain. This approach makes it possible for the modeler to get direct feedback concerning the execution time of the model. As high-level reactive system models are typically compiled to sequential host code, like C, this means that the feedback concerns the execution time of the corresponding generated tick function, which reads sensor input, performs computations, and activates output. A single such execution cycle is called a tick. The interactive timing analysis problem concerns computing the worst-case execution time (WCET) [28] 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 [8,11,14,19,28], relatively little has been done with interactive timing analysis. Previous work addresses fast WCET analysis [12], interactive C code analysis [17], analysis of Java code [23] and timing analysis of Matlab/Simulink models [15]. Also, as a proprietary solution, the timing analysis tool aiT is integrated into SCADE [7]. In contrast to previous work, we present the following contributions:

- We propose an interactive timing analysis method that includes hotspot highlighting of the worst-case path, and the separation of deep, flat, local and fractional timing values (Sec. 2).
- We present a complete Eclipse-based open source tool chain where we augment an open source modeling tool with interactive timing analysis feedback. This includes a method for tracing model elements within the compilation process and the corresponding backwards mapping of the retrieved time values to the model editor (Sec. 3).
- We formally define a generic timing analysis interface that establishes a contract between modeling and timing analysis tools. To support good response time of the analysis, the interface separates the concerns of timing analysis for external function calls and for the tick function (Sec. 4).
In Section 5, we present our evaluation, in Section 6, we position our paper with regard to related work and we give a conclusion and an outlook in Section 7.

2. INTERACTIVE TIMING ANALYSIS

In this section, we demonstrate our approach to interactive timing analysis by showing how a concrete model example can be analyzed and improved. This is followed by an overview of the analysis phase and the proposed interface.

2.1 User Modeling Level

The design flow for our approach to interactive timing analysis is shown in Fig. 1. On the top level there is the modeling tool, based on a graphical modeling language, with a compiler that yields a low level language representation, for example in C. The modeling tool requests detailed timing information of the interactive timing analysis tool, specifying code parts with the help of Timing Program Points (TPPs) and adding optional assumptions. The timing analysis provides individual analysis results for these code parts, which the modeling tool then aggregates and maps back to the model. The safety and tightness of the timing information conveyed to the user of course depends on the safety and tightness of the underlying timing analysis tool and the complexity of the target architecture. Furthermore, we are conservative during aggregation and e.g. with respect to the calling context of functions. In addition to interactive timing analysis, one can still use a traditional WCET tool for overall analysis of the generated code before final deployment. As a consequence, the interactivity of the actuator motor, is shown in the top-level SCCChart, again in the upper right corner.

For reactive systems, the maximal permitted WCET is typically part of the design specification. In the following, let's assume for example that the WCET must not exceed 250 ms. The actual WCET of the model version 1 in Fig. 2(a) exceeds this constraint by 132 ms. 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. Also, the hotspots of the model are highlighted automatically with a background color shade corresponding to their relative timing criticality. The modeler is thus quickly pointed to the region HandleMotor, whose WCET of 382 ms alone exceeds the timing specification. The modeler can now turn to this region to reconsider this part of the design and revise it, either by semantical changes or by using different function calls or different constructs with identical meaning, but different timing characteristics. For example the modeler may find that it is not necessary for the robot to take a new image when it is not moving, so that the call to getImage() can be skipped in case the accelerometer button is not pressed. Now the modeler may revise the model to version 2 shown in Fig. 2(b), where getImage() and writeLog() will not be executed in the same tick anymore.

The WCET of the robot model, meaning the maximal time between reading sensor values (such as bumper) and writing the actuator motor, is shown in the top-level SCCChart, again in the upper right corner.

In this feedback example, we focus on fractional WCET values as opposed to local WCET, a distinction that we propose as follows (also for best-case execution time (BCET)):

- **Flat timing values** denote the WCET 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 not include the execution time for the regions HandleEmergency and HandleMotor.
- **Deep timing values** for a region on the contrary take the included regions into account. Here the values are given as milliseconds. In the following, we explain the time values for the robot model and an example work flow from the user perspective.

The WCET of the robot model, meaning the maximal time between reading sensor values (such as bumper) and writing the actuator motor, is shown in the top-level SCCChart, again in the upper right corner.

For reactive systems, the maximal permitted WCET is typically part of the design specification. In the following, let's assume for example that the WCET must not exceed 250 ms. The actual WCET of the model version 1 in Fig. 2(a) exceeds this constraint by 132 ms. 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. Also, the hotspots of the model are highlighted automatically with a background color shade corresponding to their relative timing criticality. The modeler is thus quickly pointed to the region HandleMotor, whose WCET of 382 ms alone exceeds the timing specification. The modeler can now turn to this region to reconsider this part of the design and revise it, either by semantical changes or by using different function calls or different constructs with identical meaning, but different timing characteristics. For example the modeler may find that it is not necessary for the robot to take a new image when it is not moving, so that the call to getImage() can be skipped in case the accelerometer button is not pressed. Now the modeler may revise the model to version 2 shown in Fig. 2(b), where getImage() and writeLog() will not be executed in the same tick anymore.

The WCET of the robot model, meaning the maximal time between reading sensor values (such as bumper) and writing the actuator motor, is shown in the top-level SCCChart, again in the upper right corner.

In this feedback example, we focus on fractional WCET values as opposed to local WCET, a distinction that we propose as follows (also for best-case execution time (BCET)):

- **Fractional WCET or BCET** of a model element denotes its share of the overall or global WCET of the model. For a region, this is the execution time cost of the part of the critical path that lies in the region.
- **Local WCET or BCET** of a model element is the cost of the most costly execution time path that lies in this element regardless of whether it contributes to the global WCET.

For a more formal definition of what response we expect if we poll the analysis tool for a fractional or local time value we refer to Section 4.4.

1 In the SCADE/aiT integration, this corresponds to the CWCET [7].
In the proposed design flow, the modeling tool translates the reactive model into a low level intermediate language, such as C (see lower part of Fig. 1). Between the modeling tool and the interactive timing analysis tool, we propose to have a formally defined timing analysis interface. The key concept of this interface is the notion of timing program points (TPPs), that is, markers in the intermediate language where the modeling tool can request timing information. In particular those markers can be inserted to mark code parts representing a certain model element.

Figure 3: The tick function for the improved robot example (Fig. 2(b)) with timing program points.

For a better practical understanding of the terms we look at the `HandleEmergency` region in Fig. 2(a). A call to `errorLog()` and a write to the output `motor` together have a cost of 80 ms. Hence the local WCET of the region `HandleEmergency` is 80 ms. In contrast, the fractional WCET value as shown in the figure is zero, as the overall WCET path is given when `HandleMotor` is active and `HandleEmergency` is not. We focus on the fractional WCET, because it is suited best for hot spot highlighting in relation to overall WCET and can also be used to display time values as a ratio of the overall WCET. Hence the local WCET of the region in Fig. 2(a). A call to `errorLog()` and `getImage()` are mutually exclusive, presupposed that not both of them are initiated to 1; note that they are updated on lines 16-18 which have initial values and are always updated in the same tick. However, guard `g2` (or `errorLog()` (lines 9-11) and guard `g6` for `getImage()` (lines 16-18) cannot be derived to be mutually exclusive by only observing one execution of `tick()`: `g2` depends on `PRE_g1` and `g6` depends on `PRE_g4` which are independent and not defined earlier in the function. Thus, in a traditional analysis, both `PRE_g1` and `PRE_g4` have to be assumed to have arbitrary input. As a consequence, a traditional WCET tool cannot find any infeasible paths for these two if-conditions. This is a special infeasible path problem that is not necessarily handeled by tools that are otherwise able to detect infeasible program paths, as the notion of state is part of the required information, as we explain in the following.

A reactive `tick()` function is basically a Mealy machine where the state update function and the output function are combined into one tick function. If the tool can have as assumptions that `PRE_g1` and `PRE_g4` are state variables, which have initial values and are always updated in the tick before it is used, we can see that `PRE_g1` and `PRE_g4` are mutually exclusive, presupposed that not both of them were initiated to 1; note that they are updated on lines
traditional WCET analysis tools that perform WCET analysis can be performed if the tool can get as input the set of state variables (in this case PRE.g1 and PRE.g4).

A practical challenge in interactive timing analysis is to keep the round trip time from model change to updated timing results sufficiently short for an interactive design flow. We therefore distinguish between a tool that is doing the interactive timing analysis of the reactive tick function, and "traditional" WCET analysis tools that perform WCET analysis of called C functions (for instance errorLog) on line 10 or getImage) on line 17. The former takes a timing analysis request as input, which is based on the formalized timing analysis inference (Sec. 4). This request asks about timing information (WCET or BCET) between timing points. The key point is that the interactive timing analysis tool does not need to do traditional WCET analysis of complex external functions, it only computes the timing between timing program points. The WCET numbers for the called functions are instead computed offline by a traditional WCET tool, and are then used as assumptions in the interactive timing analysis. By separating these two concerns, the interactive timing analysis performance can be significantly accelerated. However, it should also be noted that the calling context of the analyzed external functions will not be part of the offline analysis, which can lead to pessimistic but still sound WCET estimates.

3. TOOL CHAIN

Our approach is not limited to a specific modeling tool or architecture. However, in this section we present a concrete, complete example tool chain for a specific architecture. This toolchain is also used to test the soundness of our concepts and the timing interface, as detailed in Section 5. As modeling tool (recall Fig. 1), we used the SCCharts modeling tool of KIELER\(^2\). We chose KIELER as it is open software, has integrated automatic layout, comes with a growing benchmark collection and supports SCCharts \(^3\) as a graphical modeling language that can model state-based systems. For low-level timing analysis, we have developed a simple experimental WCET analysis tool\(^4\), specifically designed to experiment with timing analyses between timing points. We describe the implementation on the modeling tool side further in Section 3.1 and give details on the handling of TPP on the analysis side in Section 3.2.

3.1 Modeling Tool

The KIELER SCCharts modeling tool is a textual modeling tool that offers automatically generated graphical views of the model under construction. Previously to this work, KIELER did not offer any timing analysis feedback. We augmented the KIELER SCCharts modeling tool with the feedback of the overall WCET value and also with detailed flat and deep fractional time values for each region. We implemented a display of the time values directly in the graphical view. Additionally, we added hotspot highlighting by automatically coloring the regions with different shades of red in relation to their respective share of the overall WCET. This kind of time value feedback can be seen in Figure 2, which has been created automatically with the augmented tool. Note that detailed time values could have been retrieved for arbitrary model elements, to implement this for regions is just an exemplary choice.

A main technical problem to solve on the modeling tool side is to keep track of the information which parts of the generated code belong to which regions of the original model. The modeling tool has to trace this information down the compilation chain. This allows for the automatic marking of the correct code parts with TPPs for the timing analysis request. Our approach is not restricted to any particular way of solving this problem, but as an example we explain now in more detail how we did this in KIELER.

The compilation of SCCharts in KIELER follows the Single-Pass Language-Driven Incremental Compilation (SLIC) approach presented by Motika et al. \(^22\). Thus, the compilation consists of a chain of modular transformation steps between model representations, of which each transformation is dedicated to handling specific features of the modeling language. During the SCCharts compilation up to 27 transformations may be performed, depending on the number of used SCCharts language features, until the Sequentialized Sequentially Constructive Graph (Sequentialized SCG) is reached, from which the code will be generated. The compilation of the robot example has seven intermediate model representations. The transformation chain will be passed exactly once during the compilation. Thus, the SLIC approach facilitates model element tracing, which can be performed modularly on each transformation. The resulting tracing mappings between model elements can be combined by a transitive closure to yield the overall mapping between model elements of the original model and the parts of the Sequentialized SCG. From this overall mapping, we can derive an allocation of parts of the Sequential SCG (and thus of generated code parts) to model elements, for example to regions.

Based on this concept we developed a tracing framework that also provides a view for traced models to inspect and debug the tracing, illustrated in Fig. 4. The view displays the graphical model representation of the SCChart and the generated Sequentialized SCG. The arrows visualize the tracing relations of the selected elements. The sidebar on the right side provides options for customizing the view, such as activating tracing visualization of selected elements.

As an example we illustrate a part of the tracing information for the improved robot example in Fig. 4. The screenshot shows the tracing of the transitions in regions HandleEmergency and HandleMotor. The presented SCG is zoomed to represent the lines 8 to 24 of the code in Fig. 3. For our example in Fig. 4, we can easily identify that all nodes down to motor = false are associated with region HandleEmergency and all nodes shown from g5 = pre(g4) on are associated with HandleMotor. Consequently the tool inserts TPP(2) between these nodes. TPP(3) is added at the end of the block of nodes traced to region HandleMotor. The newly created TPPs are preserved in the final code generation step. We now also

\(^2\)http://rtsys.informatik.uni-kiel.de/kieler
\(^3\)https://github.com/timed-c/kta
know which region relates to which TPP or TPPs. Thus we can request the according timing information from the timing analysis tool, aggregate the retrieved timing values for each region and display them in its graphical view in the model, thus closing the interactive feedback cycle.

3.2 Timing Analysis

The modeling tool described in the previous section generates C code as output, where timing program points are directly inserted as part of the C code. The next task of the tool chain is to compile these C code files into machine code and then to perform WCET and BCET analyses.

In contrast to conventional WCET analysis tools, such as aiT, SWEET, OTAWA, or Chronos, a timing analysis tool for interactive timing analysis needs to perform timing analysis between pairs of timing program points, not just on a specific function.

To the best of our knowledge, no existing tool can perform such timing analysis between timing points. To enable the evaluation of our approach of interactive timing analysis, we have developed an exhaustive simulation-based timing analysis tool. We will leave it as future work to design a complete WCET analysis tool that can handle the combination of interactive timing analysis, timing points, and analysis of complex hardware that includes pipelines and caches. The main steps performed by the timing analysis tool that is used for evaluation are as follows.

First, it uses a GCC-based cross-compiler targeting a MIPS32 architecture to compile the C program into an ELF-binary. Our TA tool then parses the binary, extracts MIPS instructions, and generates an internal data structure that is later used for cycle-accurate simulation. In this process, one key challenge is to make the TPPs in the C code pass through the C compiler in a way that the timing analysis tool can read out the correct locations of the timing program points from the binary. We have solved this problem by representing timing program points as assembly labels. In contrast to C labels, these labels are preserved throughout the compiler phases of the C compiler and are available in the symbol table of the ELF-binary file. We then use the addresses in the binary that define the exact positions of the timing points. Note that this approach implements a barrier semantics, where the barriers in this case are the timing points. This means that the C compiler cannot perform optimizations over these barriers. Hence, inserting timing points may affect the optimization of the C compiler, but they do not otherwise result in any other performance cost since they are represented as pure addresses in the binary. In the current experiments, we have disabled optimization of the C code.

The second step is to read the timing analysis (..ta) file, which the modeling tool has generated. This file includes all the relevant pairs of TPPs that the tool is interested in, as well as assumptions on WCET time for functions. The interactive timing analysis tool then performs cycle-accurate exhaustive simulation of all possible input combinations. Again, we note that this approach is not intended as a complete scalable solution, but it is a good way to experiment with the approach presented in this paper. To improve scalability, possible research directions could be to base the analysis on implicit path enumeration [19], abstract interpretation, explicit path analysis [16], or a combination of these techniques.

The modeled processor is a 32-bit single-cycle MIPS pro-
cessor where all the program code is assumed to fit into a fast local scratchpad memory. This is similar to a processor that is specifically designed for timing predictability [29]. During the search, the tool keeps track of the WCET path and visited timing points in such a way that the timing requests from the tool can be answered, see the next section for a formal definition of the actual timing interface and timing requests. To enable reactive timing analysis, the tool first uses a specific initialization function to initialize the defined function states. When all input values are explored, the tool computes the set of all possible output states of the analyzed function. This set of possible states is then used again as possible inputs to the function, and another simulation round is performed. The procedure then performs a fixed-point iteration and terminates when there are no more state/input combinations left to explore. Finally, the timing analysis tool reports back local and fractional timing information to the modeling tool, which in turn reports back and visualizes the results to the end user.

4. TIMING ANALYSIS INTERFACE

In this section, we formally define an interface for communicating timing information between a high-level modeling tool and a timing analysis tool. Note that this interface should only be seen as a specification; a tool can be implemented in different ways, as long as the specification is followed.

4.1 Interface Formalization

The 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 and be given initial values. A timing analysis request is a 7-tuple

\[
t_{req} = (f, a, g, S, e, P, R).
\]

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

4.2 Assumptions

The assumptions of \( t_{req} \) may be used by the interactive timing analyzer to compute tight 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.

Assumption \( a \) is a \( \mathbb{N} \to A \) 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 \( g : G \to V_S \) specifies the assumption for a value \( g(x) \) of a global variable \( x \in G \). Since the set of state variables, the variables that can be used by the interactive timing analysis tool to compute reactive infeasible paths, the interface offers an option to customize the representation for state based systems without limiting its application area to them. Finally, function \( e : F \to \mathbb{R} \times \mathbb{R} \) 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 \( (t_e, t_w) \), where \( t_e \) and \( t_w \) specify the assumptions of safe lower and upper bounds of the execution time for \( f_1 \), respectively. We represent execution time as \( \mathbb{R} = \mathbb{R} \cup \{\bot\} \), where \( \bot \) indicates that the function is non-terminating or that a safe bound has not been determined. For instance, if \( e(f_1) = (200, \bot) \), we can assume that 200 is a safe lower bound, but that we cannot prove any safe upper bound because at least one path in the function cannot be proven to terminate. Time values are given in clock cycles.

4.3 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 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, including program points \( p_0 \) and \( p_n \) that represent the function entry point and exit point, respectively. Set \( R \) specifies the requested analyses. Each element of \( R \) is a triple \((y, p_0, p_n)\), where \( y \in Y \) is the type of requested analysis value, \( p_0 \in P \) the starting program point for the analysis, and \( p_n \in P \) the ending point. We define six types of requests:

\[
Y = \{\text{WCET, BCET, LWCET, FWCET, FBET, FBCET}\}.
\]

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 the 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 FBET). The prefixes L and F stands for local and fractional, respectively. The precise meanings of the different timing requests are defined next.

4.4 Timing Response

The timing response \( t_{res} \) is a function

\[
t_{res} : R \to \mathbb{R} \cup \{\bar{p}\}
\]

where \( r \in R \) is an analysis request and the resulting value is either an execution time value \( t \in \mathbb{R} \) for \( r \in \{\text{LWCET, LBCET, FWCET, FBET}\} \), a finite path \( p = (p_1, p_2, \ldots, p_n) \) for \( r \in \{\text{WCP, BCP}\} \) or \( \bot \) for an undefined response. By \( \bar{p} \) 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 a control-flow graph (CFG) for a function \( f \) that is being analyzed. The set of vertices \( V = B \cup P \cup F \cup J \) is the union of basic blocks \( B \), timing program points \( P \), and fork \( F \) and join \( J \) nodes that express parallel execution paths. We write \( \bar{v} \) to denote a path within \( G \). From \( G \), we can derive a timing points graph \( G' = (P, E_p) \), where all vertices are program points, and edges \( E_p = \{(v, w) \mid v, w \in P\} \).
The best-case execution time is defined in the same way as the etime.

**Definition 2 (Timing Program Point Path).** For a path $\vec{v}$ in $G$, the corresponding timing program point path, denoted $\vec{p} = tpath(\vec{v})$, is derived by removing all vertices $v \not\in P$ from $\vec{v}$.

From the specification point of view, suppose there exist functions $c_v : E \to \mathbb{N}$ and $c_e : E \to \mathbb{N}$ 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 $e$ (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 that there exist functions for computing the worst-case execution time $\overline{\text{etime}}(v_1, v_2, \ldots, v_n)$, and a best-case execution time path $\overline{\text{etime}}(v_1, v_2, \ldots, v_n)$, between two timing program points $p_1$ and $p_2$, respectively. The empty path is returned if $p_2$ is not reachable from $p_1$ or $p_2$ is returned if the path between $p_1$ and $p_2$ cannot be proven as finite (the execution may be non-terminating). Note that the worst-case execution time path $\overline{\text{etime}}(v_1, v_2, \ldots, v_n)$ contains both basic block and timing program point vertices, but the path returned by requesting $\text{WCP}$ only contains timing program points.

**Definition 3 (Subpath).** Let $\text{spath}_{p_1, p_2}(\overline{v})$ be a subpath of $\overline{v}$, which contains the contiguous sequence of vertices between and including $p_1$ and $p_2$. That is, for a sequence $\overline{v} = (v_1, v_2, \ldots, v_n)$, $\text{spath}_{p_1, p_2}(\overline{v}) = (v_{i_1}, v_{i_2}, \ldots, v_{i_m})$, where $i_1$ is the first vertex in $\overline{v}$ that is a path in $G$. The worst-case execution time is defined in the same way using $c_v$. If $\overline{v}$ is the empty path, $\text{spath}_{p_1, p_2}(\overline{v}) = 0$. Moreover, \text{etime}(\perp) = 0.

Note that this path-related definition together with $\overline{\text{etime}}$ and $\text{etime}$ covers also fork and join nodes. There is no restriction in the interface on how the timing analysis tool implements those functions, for example this could involve a max-plus algebra calculation, see [1] for a survey, for parallel execution of concurrent code parts.

The timing response time function is then defined as follows:

$$t_{\text{res}}(r) = \begin{cases} t\text{path}(\text{spath}_{p_1, p_2}(\overline{v}_{p_1, p_2})), & \text{if } r = \{\text{WCP}, p_1, p_2\} \\ t\text{etime}(\overline{v}_{p_1, p_2}), & \text{if } r = \{\text{BCP}, p_1, p_2\} \\ t\text{etime}(\overline{v}_{p_1, p_2}), & \text{if } r = \{\text{LBCET}, p_1, p_2\} \\ t\text{etime}(\overline{v}_{p_1, p_2}), & \text{if } r = \{\text{FCET}, p_1, p_2\} \\ t\text{etime}(\overline{v}_{p_1, p_2}), & \text{if } r = \{\text{FCET}, p_1, p_2\} \\ t\text{etime}(\overline{v}_{p_1, p_2}), & \text{if } r = \{\text{FCET}, p_1, p_2\} \end{cases}$$

**4.5 Example**
The different requested execution times are best illustrated using an example. Fig. 5 shows a CFG for a function $f$. The graph has seven basic blocks ($b_1$ to $b_7$) and six timing program points ($p_1$, $p_2$, $p_3$, $p_4$, $p_5$, and $p_6$). White arrows represent true-branches and black arrows represent false-branches and unconditional branches. The graph is cyclic (nodes $b_5$ and $b_6$), but a derived graph between timing program points is acyclic. Time values are given in clock cycles.

![Figure 5: A CFG that includes basic blocks (b1 to b7) and timing program points (p1, p2, p3, p4, p5, and p6). White arrows represent true-branches and black arrows represent false-branches and unconditional branches. The graph is cyclic (nodes b5 and b6), but a derived graph between timing program points is acyclic. Time values are given in clock cycles.](image)

**5. Evaluation**
We used a test suite to validate our concepts, as detailed in the following. Also we conducted a small user study, which will be introduced in Section 5.2.

**5.1 Test Suite**
With the help of the example implementation introduced in Section 3, we employed a test suite to check the soundness of the concepts and interface of the interactive timing analysis. Note that the validation does not focus on the quality of timing analysis approximations, as we do not claim the

As a second part of the evaluation we conducted a study, with no requirement to pass the exam with full score. The students could get partial credit for an elaboration of the study and were evenly distributed over the four groups. The participants had to solve the task to revise the model in order to reduce the overall WCET below a given threshold. To do this they were allowed to exchange host code calls for calls to alternative functions. The participants received a list of available functions, without timing information.

The working time of the participants was measured. In case a participant took longer than 25 minutes, the attempt was aborted to limit the potential strain for each participant. The consumed working times are shown in Figure 6, with an approximated working time of 25 minutes for the aborted attempts. This applies to eight of the eleven members of Group 1 and three out of eleven for Group 2. In both of the groups with hotspot highlighting, all participants were able to finish in time. An interesting result is that the time values for Group 1, with least timing information, and Group 4, with most available feedback, are completely disjoint. That is, all participants of Group 1 took longer time to finish compared to all participants of Group 4. This strengthens the indication that the full interactive timing analysis feedback was helpful in this setting. Except for a single value this is also true for Group 3. For Group 2, the majority of the time values are improved as well as the number of aborted attempts. The time values for Group 4 show the lowest standard deviation, possibly pointing to an increased degree of effective guidance. For Group 3 a cluster of time values is observable, although the results deviate more overall. The two hotspot-highlighting-based configurations are distinguished as the only ones in which no participant broke the time limit. The results indicate that hotspot highlighting alone might not be far inferior in effect to hotspot highlighting with detailed region time values.

It seems likely that the detailed interactive timing analysis actually can help improve model revision productivity. An elaborate experiment on interactive timing analysis in an industrial setting with a larger number of benchmarks and higher degrees of possible model changes would be a research endeavour of its own and is left to future work.

### Table 1: Selected validation test cases

<table>
<thead>
<tr>
<th>Model</th>
<th>Nodes</th>
<th>Regions</th>
<th>TPP</th>
<th>Focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>CircleWithCalls</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>State</td>
</tr>
<tr>
<td>Controller</td>
<td>28</td>
<td>7</td>
<td>11</td>
<td>State</td>
</tr>
<tr>
<td>FunParc2</td>
<td>150</td>
<td>40</td>
<td>49</td>
<td>State</td>
</tr>
<tr>
<td>MedicalAid</td>
<td>50</td>
<td>13</td>
<td>25</td>
<td>LWCET</td>
</tr>
<tr>
<td>Feeder</td>
<td>21</td>
<td>7</td>
<td>18</td>
<td>Hotspot</td>
</tr>
<tr>
<td>MultiWait</td>
<td>198</td>
<td>99</td>
<td>103</td>
<td>TPP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>Nodes</th>
<th>Regions</th>
<th>TPP</th>
<th>Focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>CircleWithCalls</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>State</td>
</tr>
<tr>
<td>Controller</td>
<td>28</td>
<td>7</td>
<td>11</td>
<td>State</td>
</tr>
<tr>
<td>FunParc2</td>
<td>150</td>
<td>40</td>
<td>49</td>
<td>State</td>
</tr>
<tr>
<td>MedicalAid</td>
<td>50</td>
<td>13</td>
<td>25</td>
<td>LWCET</td>
</tr>
<tr>
<td>Feeder</td>
<td>21</td>
<td>7</td>
<td>18</td>
<td>Hotspot</td>
</tr>
<tr>
<td>MultiWait</td>
<td>198</td>
<td>99</td>
<td>103</td>
<td>TPP</td>
</tr>
</tbody>
</table>

Figure 6: Times for the four groups with different types of timing information.
6. RELATED WORK

During the last decades, a significant amount of work has been done in the area of WCET analysis. A comprehensive overview of the research field is given by Wilhelm et al. [28]. Most works 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. [12] have developed a tool chain for interactive WCET analysis, where the performance of the analysis time is favored over 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. Kirner et al. [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 [24] present an interactive timing environment for Java, where WCET analysis is performed at the byte-code level. Ko et al. [17] have developed an interactive timing analysis environment for C programs, where portions of the program can be selected and analyzed. The tool aiT is integrated into SCADe for timing feedback, but their analysis granularity is limited to function calls. This precludes inlining, and also does not give timing feedback to semantic elements of the original model that correspond to multiple, unconnected regions in the generated code. 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.

There have also been some investigations of WCET/WCRT analysis for synchronous languages. Mendler et al. [20] propose an algebraic approach for the WCRT analysis for Esterel programs. Raymond et al. [25] have been concerned with infeasible paths in the binary code under timing analysis that are implied by high-level functional properties given in synchronous languages, the concrete language in their design being the dataflow language Lustre [5]. This work is related to our concept of reactive infeasible paths, however the authors use an existing model checker to verify the feasibility of paths according to high-level semantics and to trace the functional properties from Lustre to C and from C to binary code. In contrast we are working with statebased modeling systems and identify the state representation variables in the C code for the analysis tool to enable a statebased timing analysis. Also, we trace the representation of model elements for a graphical control-flow based language, whose compilation involves a chain of model transformation steps down to code generation and use the concept of timing program points to identify the corresponding code parts. Wang et al. propose an ILP-based approach that exploits concurrency explicitly [27]. 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. [13], who back-annotate an Esterel program with information regarding the timing-critical path. However, they do not 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 [6] 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.

For textual programming languages such as C, the general concept of quickly guiding users to timing hot spots using visual notations has been applied in the context of profiling by long-established tools such as IBM’s Rational Quantify. For model-driven engineering, the user story on interactive timing analysis advocated in this paper fits into the general idea of modeling pragmatics that strives to enhance user productivity by making the best possible use of visual models [9].

Parts of this work have been presented at a workshop before, but without formal publication. An earlier version of the work is available as technical report [10].

7. CONCLUSION AND OUTLOOK

In this paper, we have explained and demonstrated how a complete tool chain can be augmented with interactive timing analysis capabilities. Key properties of our approach are i) hotspot highlighting at the model level ii) clear display of timing annotations in the form of deep, flat, and fractional timing values, iii) a formally defined timing analysis interface that clearly explains the meaning of timing requests and timing responses, iv) the tracing of model elements for which timing is requested during compilation and v) the introduction of state variable assumptions to address a special infeasible path problem, the reactive infeasible paths. For future work, we want to use the detailed timing information for optimizations in parallel code compilation. Also, we plan to further investigate the combination of interactive timing analysis with traditional WCET analysis tools. As our approach is not limited to WCET analysis, it may be transferred to the analysis of average time behavior as well.

Acknowledgment

We thank Christian Motika and Steven Smyth for their help and ideas for tracing in the SLIC approach. This work has been funded in part by the German Research Foundation within the Precision-Timed Synchronous Reactive Processing project (PRETSYS2, DFG HA 4407/6-2). This work was also supported by the Swedish Research Council #623-2013-8591.

8. REFERENCES


