WCRT Algebra and Interfaces for Esterel-Style Synchronous Processing

Michael Mendler
Faculty of Inform. Sys. and Appl. Comp. Sciences
The University of Bamberg
michael.mendler@uni-bamberg.de

Reinhard von Hanxleden and Claus Traulsen
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
{rvh|ctr}@informatik.uni-kiel.de

***Abstract*—The synchronous model of computation together with a suitable execution platform facilitates system-level timing predictability. This paper introduces an algebraic framework for precisely capturing worst case reaction time (WCRT) characteristics for Esterel-style reactive processors with hardware-supported multithreading. This framework provides a formal grounding for the WCRT problem, and allows to improve upon earlier heuristics by accurately and modularly characterizing timing interfaces.**

I. Introduction

Reconciling performance and predictability in embedded systems is a challenge that spans all layers of hard- and software development. However, as observed by Edwards and Lee [1], these abstractions typically limit themselves to encapsulate and guarantee functionality, not timing. Hence, even though there is a significant body of work that addresses timing predictability at different abstraction layers, considering for example schedulability, worst case execution times (WCET), or circuit timing, it is very difficult to transfer results across layers. However, end users are not interested in results that apply only to one layer—they care about timing guarantees for complete systems.

The choice of the model of computation—and its model of time—has a profound influence on how easy or difficult it is to provide timing guarantees across abstraction layers. From the predictability point of view, a very appealing candidate in the embedded systems domain is the synchronous model of computation [2]. Furthermore, languages built on that model generally have a well-established formal semantics that allows reasoning about functional as well as timing properties from the ground up.

In this paper, we first give an overview on how the synchronous model together with a suitable execution platform can provide system-level timing predictability (Section III), and we illustrate this with the case of multi-threaded execution of synchronous programs written in Esterel-like languages [3]. The main contribution of this paper (Section IV) then is the introduction of an algebraic framework for performing Worst Case Reaction Time (WCRT) analyses. These analyses aim to give conservative yet close estimates on the time from capturing inputs to determining outputs, in embedded real-time systems developed with the synchronous model. First experimental results are reported in Section V.

Due to space considerations, this presentation is fairly condensed; a more complete development can be found in a separate report [4].

II. Related Work

Most interface models in synchronous programming are restricted to causality issues, *i. e.*, dependency analysis without quantitative time. On the other hand, there exist numerous approaches to classical WCET analysis [5] but only few on WCRT analysis [6], [7].

Logothetis *et al.* [8] have employed model checking to perform a precise timing analysis for the synchronous language Quartz, which is similar to Esterel. However, their problem is WCET since they are interested in computing the number of logical ticks required to perform a certain transformational computation, such as a primality test.

The modules of André *et al.* [9] do not permit instantaneous interaction. Such a model is not suitable for WCRT. Hainque *et al.* [10] use a topological abstraction of the underlying circuit graphs (or syntactic structure of Boolean equations) to derive a fairly rigid component dependency model with the effect that multi-threaded execution cannot be modeled compositionally. The interface model also does not cover data dependencies and thus cannot deal with dynamic schedules and does not support WCRT, either.

The causality interfaces of Lee *et al.* [11] are more flexible. These are functions associating with every pair of input and output ports an element of a *dependency domain* D, which expresses if and how an output depends on some input. Causality analysis is then performed by multiplication on the global system matrix. Using an appropriate dioid structure D, one can perform the analyses of Hainque et. al. [10] as well as restricted forms of WCRT. However, Lee's interfaces cannot express the difference between an output depending on the joint presence of several values as opposed to depending on each input individually. Thus they do not support full AND- and OR-type synchronization dependencies and hence cannot represent neither multi-threading nor multi-processing. The work reported here can be seen as an extension of [11] to address these deficiencies.

Similar restrictions apply to recent work [12], [13] combining network calculus [14], [15] with real-time interfaces. These works are concerned with the compositional modeling of regular execution patterns rather than stabilization processes inside each execution cycle of a synchronous program. Existing interface theories [11], [12], [13], which aim at the verification of resource constraints for real-time scheduling, handle timing properties such as task execution latency, arrival rates, resource utilization, throughput, accumulated cost of context switches, and so on. However, the dependency on data and control flow is largely abstracted. For instance, since the task sequences of Henzinger and Matic [13] are independent of each other, their interfaces do not model concurrent forking and joining of threads. The causality expressible there is even more restricted than that by Lee *et al.* [11] in that it permits only one-to-one associations of inputs with outputs. The interfaces of Wandeler and Thiele [12] for modular performance analysis in real-time calculus are like those of Henzinger and Matic [13] but without sequential composition of tasks and thus do not model control flow as we do here.

In so far as WCRT analysis aims to obtain exact bounds on the duration of stabilization processes with synchronous feedback, it is related to the timing analysis of combinational circuits (see, e.g., [16], [17], [18], [19]) which is known to be NP-complete. Although WCRT analysis for single or multi-threaded synchronous processing can mostly be performed in max-plus as opposed to min-max-plus algebra, the inherent data dependency still makes it computationally intractable without sophisticated heuristics. The work presented here fits into a general and expressive interface theory [20] for stabilization processes which has been developed to provide a semantical foundation for such heuristics. It supports modularization and hierarchical abstraction and systematizes earlier work on combinational timing analysis.

III. Synchronicity and Timing Predictability

The synchronous model of computation divides physical time into a sequence of discrete *ticks*, or *instants*. The abstraction is that at each tick, outputs are synchronous with the inputs. In other words, computations take place instantaneously, interspersed with durations of inactivity between ticks. Synchronous languages generally do not permit unbounded computations within a tick. For example, the language Esterel provides a loop construct [3], however, instantaneous loops are forbidden; *i. e.*, each loop iteration must include at least one tick-delimiting instruction, and the compiler must be able to verify this. This simplifies the problem of determining the maximal number of instructions per tick, which leads to the worst case reaction time (WCRT). The situation is quite different for imperative languages such as Java or C, which permit unbounded loops and unbounded recursion, and thus only language subsets (*e. g.*, with statically bounded loop iterations) are amenable to WCET analyses. Another helpful characteristic of (strict) synchrony is that the statuses of signals, which are basic communication means in synchronous programs, evolve monotonically. There can be no oscillations between signal presence and absence, thus guaranteeing convergence after a finite number of computations. This contrasts, for example, with Harel's original Statecharts dialect [21], which assumes a weaker form of synchrony in which computations are also assumed to not consume any time, but signal statuses are allowed to oscillate and computations within a tick are unbounded. Finally, the synchronous paradigm also supports concurrent and preemptive control flow, with a deterministic semantics regarding both functionality and timing characteristics. This again contrasts with classical imperative languages, which either do not support non-sequential control flow at all (*e. g.*, C relegates this to the OS level, subject to run-time scheduling decisions), or support it only in a rather haphazard fashion (*e. g.*, Java threads [22]).

Synchronous programs may be compiled into hardware or software. The traditional software design flow is to first compile the synchronous program into a classical imperative language, such as C, and to run the resulting program on a standard micro processor [23]. This approach preserves the nice semantical properties of synchronous programs at a functional level. The timing properties, however, are only partially preserved with this approach. Computations are still finite per tick, and the synthesized C-code has no unbounded loops, for example. However, depending on the synthesis approach used, the control flow may still be rather complex and difficult to analyze (*e. g.*, computed gotos). Furthermore, standard processors typically employ various techniques that improve average execution time, at the expense of worst case execution time and predictability [24].

An alternative, more recent approach for executing synchronous programs is to run them on processors that directly support reactive control flow. This *reactive processing* approach builds on instruction set architectures (ISAs) that can express concurrency and preemption and preserve functional determinism [25]. Note that this approach does typically not strive for high-performance processing that maximizes average execution times and uses hardware acceleration techniques such as caching and pipelining [1]. Instead, the focus here is on predictable architectures with fixed instruction cycle times.

There have been various proposals on how to support concurrency in reactive processing, including sequentialization, parallel execution, and, most recently, multi-threading [26], [27], [28]. The latter one appears to be the most effective at this point, and significantly outperforms classical software-based execution strategies while using minimal resources.

In the multi-threaded reactive processing approach, timing determinism is assured by a combination of static scheduling, hardware-supported context switching, and fixed machine instruction execution times. This has been exploited in a compiler which translates Esterel programs into multi-threaded assembler code for the Kiel Esterel Processor (KEP), and as part of the compilation process analyzes the WCRT in terms of instruction cycles [7].

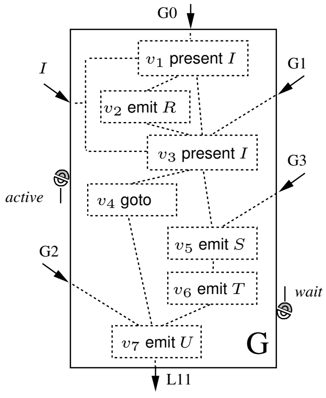
The WCRT analysis technique developed by Boldt *et al.* [7] already provide fairly promising results with a reported accuracy typically in the 30–40% range. However, this heuristics still makes conservative and simplifying assumptions and is not grounded in a formal timing model. To illustrate this,

```
// Esterel Program G:
present I then
    emit R end;
present I else
    emit S;
    emit T end;
emit U;
```

(a)

```
// Assembler Fragment G
L04: G0: PRESENT I,G1
L05:       EMIT R
L06: G1: PRESENT I,G3
L07:       GOTO G2
L08: G3: EMIT S
L09:       EMIT T
L10: G2: EMIT U
```

(b)



(c)

Fig. 1. Example program G: Esterel source (a), KEP assembler (b), control flow graph (c).

consider the small program G in Fig. 1(a) and the corresponding assembler (b). Considering that each of the basic instructions present, emit, goto takes 1 instruction cycle (ic) regardless how it is entered or exited, the longest-path heuristic implemented in the KEP compiler [7] will compute a WCRT of 6 ics. This, however, is overly conservative, as the longest path makes contradictory assumptions (signal I present and absent at the same time). Furthermore, the existing WCRT algorithms are neither compositional nor scalable in terms of precision. They are global analyses on the complete and fully-expanded control-flow graph of a monolithic program and run at the ground level of atomic program statements rather than hierarchical sub-systems.

In this paper we propose a theory of WCRT interfaces for synchronous programming and show how it can be employed to obtain type-directed and modular WCRT analyses which (1) give precise statements about exactness and coverage of timing values, supporting a variety of timing abstractions, (2) are dedicated to express the imperative synchronous programming languages, and (3) are scalable across component hierarchies and the software-hardware abstraction boundary. As an interface theory our WCRT algebra operates on matrices of delay values characterizing whole sub-systems rather than individual nodes like existing graph-theoretic WCRT algorithms do. Like the propositional stabilization theory presented in [20] it combines max-plus algebra $(\mathbb{N}, \mathit{max}, +, 0, -\infty)$ [14] with an intuitionistic refinement of Boolean logic to reason about implicit control-flow.

IV. The WCRT Algebra

An *execution* σ is a finite and monotonically increasing sequence of sets of *control signals* $A \in \mathbb{S}$, which can be data-signals or control-flow labels. We also use activation controls $\mathit{active}(v) \in \mathbb{S}$ for nodes v in the hierarchical decomposition of the program. An execution σ models the micro-sequence of instruction cycles (ic) which are executed by a given thread within a single synchronous instant. Each step $\sigma(i) \mapsto \sigma(i+1)$ records the change of controls between two successive activations of the thread. The *empty execution* $\sigma = \emptyset$ is included as a degenerated case. The difference $\Delta_i = \sigma(i+1) \setminus \sigma(i)$ may be an arbitrary subset of $\mathbb{S}$. It will encompass more than one signal when the thread forks into concurrent sub-threads or if other concurrent threads get executed between the two activations i and $i+1$ of the thread represented by σ.

A. Scheduling Types

A set of executions S defines a *schedule*. The possible schedules of a program are specified by a *scheduling type*

$$\begin{aligned}\phi \;::=\;& A \mid \mathit{true} \mid \mathit{false} \mid \phi \wedge \phi \mid \neg\phi \mid \phi \supset \phi \mid \\ & \phi \vee \phi \mid \phi \oplus \phi \mid \phi \parallel \phi \mid \circ\phi.\end{aligned}$$

We write $S \models \phi$ ($\sigma \models \phi$) to say that schedule S (execution σ) *validates* the type ϕ. As a type, each signal $A \in \mathbb{S}$ represents the statement that "*A is active (is present, traversed, scheduled) in all executions of the schedule*." The constant *true* is validated by all schedules and *false* only by the empty execution or the schedule which contains the empty execution only. The type operators $\neg$, $\supset$ are (intuitionistic) negation and implication. The operators $\vee$ and $\oplus$ are two forms of logical disjunction to encode internal and external non-determinism and $\wedge$, $\parallel$ are two forms of logical conjunction related to true concurrency (multi-processing) and interleaving concurrency (multi-threading), respectively. Finally, $\circ$ is the operator to express execution delays. This type syntax permits definitions provided in the following.

A *basic control type* is an expression ζ built from literals A, $\neg A$ ($A \in \mathbb{S}$) and constants *true*, *false* using conjunction $\wedge$ and disjunction $\oplus$. Basic control types satisfy $S \models \zeta$ iff $\sigma \models \zeta$ for all $\sigma \in S$, *i. e.*, they express properties of individual executions. On executions, ζ behaves like a standard Boolean combination of the atomic statements A ("*A present throughout*") and $\neg A$ ("*A absent throughout*"). For instance, $\sigma \models A \oplus \neg A$ says that signal A is constant in σ, *i. e.*, it is either present from the start, or never becomes active. Since signals which are not active initially may occur in the course of an execution, the type $A \oplus \neg A$ is not a tautology, *i. e.*, $A \oplus \neg A \ncong \mathit{true}$. This intuitionistic nature of negation is crucial to handle the semantics of synchronous languages in a compositional and fully abstract way [29]. For special signals like the activation of nodes $\mathit{active}(T)$ it is safe to assume $\mathit{active}(T) \oplus \neg \mathit{active}(T) \cong \mathit{true}$ since these state signals are decided at the start of every instant. Every basic control has an equivalent disjunctive normal form $\zeta = \bigoplus_i \bigwedge_j l_{ij}$ over literals l_{ij}. Basic controls ζ are used to specify scheduling interaction at the input and output side of a program node. When used as an output we need to express that ζ occurs delayed after some maximal number of ics, d say. We write $\sigma \models d : \zeta$ to abbreviate of $\sigma' \models \zeta$ where $\sigma' = \sigma(d)\sigma(d+1)\cdots\sigma(|\sigma|-1)$ is the suffix of σ starting after d ics. Note that if the delay is larger than the length of the execution, $d > |\sigma| - 1$, then this suffix is empty $\sigma' = \emptyset$ and thus $\sigma \models d : \zeta$ for all ζ, even $\zeta = \mathit{false}$ is validated. This is natural since by stepping beyond the final event within a thread's instant an inconsistent state is reached. This may be exploited for optimizations in WCRT analysis [4]. The specification *wait* $=_{df}$ 1 : *false* is of particular interest. It says that an execution has at most one event, i.e., $\sigma \models$ *wait* iff $|\sigma| \leq 1$. If non-empty such an execution has reached the end of the scheduling instant and is pausing in a final event $\sigma(0) \subseteq \mathbb{S}$. The reaction time of an execution σ may then be bounded by d either as $\sigma \models d$: *wait* or $\sigma \models d + 1$: *false* depending on whether we are interested in the number of steps or the number of events in σ.

An *output control* is an expression $\psi = \circ\zeta_1 \oplus \circ\zeta_2 \oplus \cdots \oplus \circ\zeta_n$ with basic controls ζ_i. $S \models \psi$ specifies that schedule S reaches at least one of the controls ζ_j after a bounded number of *instruction cycles* (ics). The selection $\oplus$ of which ζ_j is activated is an internal choice which is dynamically resolved during each execution. Each operator $\circ$ stands for a possibly different delay depending on which output ζ_j is taken. In contrast to this, an output control such as $\psi = \circ(\zeta_1 \oplus \zeta_2 \oplus \cdots \oplus \zeta_n)$ only specifies a single bound for all exits ζ_j.

An *input control* is an expression $\phi = \zeta_1 \vee \zeta_2 \vee \cdots \vee \zeta_m$ where the disjunction $\vee$ refers to the external non-determinism resolved by the environment which determines how a program node is started. There is also no delay involved which is why we do not need operator $\circ$. Formally, $S \models \phi$ if there is at least one ζ_i such that $S \models \zeta_i$.

Notice the change of quantifiers between input and output controls regarding executions: $S \models \zeta_1 \vee \zeta_2$ requires $\exists i \in \{1, 2\}. \forall \sigma \in S.\ \sigma \models \zeta_i$ which is an external choice, whereas $S \models \zeta_1 \oplus \zeta_2$ is $\forall \sigma \in S. \exists i \in \{1, 2\}. \sigma \models \zeta_i$ which expresses an internal choice.

B. Interface Types

We build *interface types* for program fragments as implications $\phi \supset \psi$ between input controls $\phi = \bigvee_{i=1}^{m} \zeta_i$ and output controls $\psi = \bigoplus_{j=1}^{n} \circ\xi_j$. The input controls ϕ capture all the possible ways in which the program fragment can be started within an instant and the output controls sum up the ways in which it can be exited during the instant. Intuitively, $S \models \phi \supset \psi$ says that whenever any set of executions from schedule S enters the program through one of the input controls ζ_i, then within some bounded number d_{ij} of ics all these executions are guaranteed to exit through one of the output controls ξ_j. The bounds d_{ij} may depend on the choice of input and output control, in general. To capture the bounds, we associate with each interface type a delay matrix of shape $n \times m$. Our type specifications then become logical expressions of the form $D : \phi \supset \psi$ consisting of a timing matrix D together with an interface type $\phi \supset \psi$. The former describes the quantitative aspect of scheduling, the latter captures the qualitative part of the interface. Formally, $\phi \supset \psi$ is a type specification for schedules S and the instrumented $D : \phi \supset \psi$ specifies a set of executions.

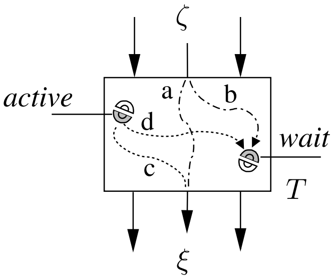


Fig. 2. The four types of thread paths: through path (a), sink path (b), source path (c), internal path (d).

Fig. 2 depicts a program fragment T abstracted into a reactive box with input and output controls. The paths inside T seen in Fig. 2 illustrate the four ways in which a reactive node T may participate in the execution of a logical tick: Threads may (a) pass straight through the node entering at some input control ζ and exiting at output control ξ; (b) enter through ζ but pausing inside, waiting there for the next instant; (c) start the tick inside the node and eventually (instantaneously) leave through some exit control ξ, or (d) start inside the node and never leave it during the current instant. These paths or rather sections of a path are called *through paths*, *sink paths*, *source paths* and *internal paths*, respectively.

The interface type for such a node T (considering only one input control ζ and one output control ξ) separates these different paths and associated WCRT values:

$$T \quad = \quad \begin{pmatrix} d_{\mathit{thr}} & d_{\mathit{src}} \\ d_{\mathit{snk}} & d_{\mathit{int}} \end{pmatrix} : (\zeta \vee \mathit{active}) \supset (\circ\xi \oplus \circ \mathit{wait})$$

If one of the paths does not exist its associated delay is set to $-\infty$. A node T can be classified according to the paths that are executable in it. We define the (not necessarily disjoint) sets of *through* nodes, $N_{\mathit{thr}} = \{T \mid d_{\mathit{thr}} \geq 0\}$, *source* nodes, $N_{\mathit{src}} = \{T \mid d_{\mathit{src}} \geq 0\}$, *sink* nodes, $N_{\mathit{snk}} = \{T \mid d_{\mathit{snk}} \geq 0\}$, and *internal* nodes, $N_{\mathit{int}} = \{T \mid d_{\mathit{int}} \geq 0\}$. A *delay node* is a node with at least one non-instantaneous path ($N_{\mathit{del}} = N_{\mathit{src}} \cup N_{\mathit{snk}} \cup N_{\mathit{int}}$). A *strong delay node* is a delay node without any through path ($N_{\mathit{sdel}} = N_{\mathit{del}} \setminus N_{\mathit{thr}}$). A *transient node* is a through node that contains only through paths, *i. e.*, $d_{\mathit{src}} = d_{\mathit{snk}} = d_{\mathit{int}} = -\infty$ ($N_{\mathit{trans}} = N_{\mathit{thr}} \setminus N_{\mathit{del}}$). Each cyclic dependency loop in the program must be broken by at least one strong delay node, which corresponds to the rule mentioned earlier that forbids instantaneous loops.

In general, the interface type of a program T will mention a number of controls $\zeta_1, \zeta_2, \ldots \zeta_m$ and $\xi_1, \xi_2, \ldots, \xi_n$ on the input and output side for which the type would be

$$T = D : (\zeta_1 \vee \zeta_2 \cdots \vee \zeta_m) \supset (\circ\xi_1 \oplus \circ\xi_2 \oplus \cdots \oplus \circ\xi_n) \quad (1)$$

with a WCRT matrix D of shape $n \times m$. A composite program will be made up of a number of program fragments T_i each with its interface $D_i : \phi_i \supset \psi_i$. The total specification is the logical conjunction $\bigwedge_i D_i : \phi_i \supset \psi_i$ in WCRT type algebra. The basic controls appearing in ϕ_i, ψ_i describe the causal dependencies between the nodes T_i. In its general form, WCRT analysis amounts to a transformation

$$\bigwedge_i D_i : \phi_i \supset \psi_i \qquad \preceq \qquad D : \phi \supset \psi \qquad (2)$$

in which the individual timing interfaces D_i are combined into a total delay matrix D for an external interface $\phi \supset \psi$ such that D is the smallest (component-wise) matrix of values such that (2) holds. The external interface $\phi \supset \psi$ determines the functional precision with which we are computing the WCRT of a composite system. For instance, instead of an interface like (1), which distinguishes m input and n output controls, a less discriminative type $\zeta \supset \circ\xi$ with $\zeta =_{df} \bigvee_{i\in I} \zeta_i$ and $\xi =_{df} \circ \bigoplus_{j\in J} \xi_j$ might consider merely subsets $I \subseteq \{1, \ldots, m\}$ and $J \subseteq \{1, \ldots, n\}$ of inputs and outputs bundled into a single control. Such an interface $\zeta \supset \circ\xi$, which specifies only one delay value is more abstract than (1). Of course, we do not expect to get an equivalence $\cong$ but only an inclusion $\preceq$ in (2) if the calculation of D involves timing abstractions. We can trade off precision and efficiency of the WCRT analysis within wide margins by choosing different types $\phi_i \supset \psi_i$ for the components and $\phi \supset \psi$ for the composite program in (2). By logical transformations of interfaces, various optimizations can be achieved including such as those employed by classic combinational timing analyses [20].

C. An Example

To illustrate the use of WCRT types consider again Fig. 1. Each node v_1–v_7 in the control-flow graph (c) of the associated Esterel program (a) is compiled into an assembler instruction (b) which is entered either sequentially through its instruction number L4–L10 or through an explicit jump to a control flow label such as G0–G3. For instance, node v_3 is accessed both through its linear instruction number L6 as well as by jump to its label G1. In contrast, node v_4 is only accessed through its line number L7 while node v_5 only by jumping to its label G3. The present nodes v_1 and v_3 are tests which branch to their two successor instructions depending on the status of signal I. If I is present then v_1 moves to instruction v_2 which immediately follows it and if I is absent then v_1 passes control to instruction v_3 by jumping to label G1.

An interface which only considers the input $G0$ and computes the longest path through G is $(6) : G0 \supset \circ L11$. A full WCRT specification encapsulating program G as a component would require mention of program labels $G1$, $G3$, $G2$ which are accessible from outside for jump statements. Therefore, the interface type of G would be $(6, 4, 3, 1) : (G0 \vee G1 \vee G3 \vee G2) \supset \circ L11$. This is still not the most exact description of G since it does not express the dependency of the WCRT on signal I. In particular, the longest path of length 6 from $G0$ to $L11$ is not executable. To capture this we consider signal I as just another control input and refine the WCRT scheduling type of G as follows: $(5, 5, 3, 4, 3, 1) : ((G0 \wedge I) \vee (G0 \wedge \neg I) \vee (G1 \wedge I) \vee (G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11$. The inclusion of signal I in the interface has now resulted in the distinction of two different delays 3 and 4 for $G1 \supset \circ L11$ depending on whether I is present or absent during the reaction. On the other hand, $G0$ split into controls $G0 \wedge I$ and $G0 \wedge \neg I$ produces the same delay of 5 ics in both cases, which is a decrease of WCRT compared to 6 from above. Assuming that input signal I is causally stable, *i. e.*, $I \oplus \neg I \cong$ *true*, the two entries of value 5 can be merged into a single value as in $(5, 3, 4, 3, 1) : (G0 \vee (G1 \wedge I) \vee (G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11$. In the same vein, we could further bundle $G1 \wedge I$ and $G3$ into a single input control $(G1 \wedge I) \oplus G3$ with delay 3. This finally gives $(5, 3, 4, 1) : (G0 \vee ((G1 \wedge I) \oplus G3) \vee (G1 \wedge \neg I) \vee G2) \supset \circ L11$. Still, if we only ever intend to use G as a composite node from $G0$ to $L11$, the typing $(5) : G0 \supset \circ L11$, which takes care of signal dependency on I, might be sufficient.

All operations on interfaces and WCRT analyses are supported by semantically sound transformation rules in WCRT type algebra. The logical manipulation of types often can be done implicitly and hard-coded into the graph-theoretic search strategies that make up the cleverness of a particular WCRT algorithm. Where interface types are not used directly in the calculations they provide for a highly compositional fine-analysis which allows us to validate WCRT algorithms in terms of precise statements about correctness and exactness. Due to their logical-symbolic nature WCRT interfaces can be applied in rather general situations which involve data and higher control-flow constructs as used in synchronous programming.

V. RESULTS

To evaluate our approach, we prototypically implemented some of the key ideas. We identify blocks with threads and compute the through, source, sink and internal WCRT for each thread independently. All outgoing transitions from a thread are abstracted into one. The results for some test-cases, taken from the Estbench test suite [30], can be found in Table I. Since the approach does not consider traps yet, we had to replace traps by local signals and weak abortion. This is trivial for these examples. In general, the transformation can be done analogously to the hardware synthesis from Esterel [23].

TABLE I
EXPERIMENTAL RESULTS FOR THE REACTION TIME IN INSTRUCTION CYCLES. *Graph* AND *Interface* ARE STATICALLY ESTIMATED WCRT, USING THE GRAPH BASED APPROACH [7] AND THE INTERFACE APPROACH PRESENTED IN THIS PAPER, RESPECTIVELY.

	WCRT	
Module name	Graph	Interface
abro	11	11
atds	60	34
mca200	1779	1782
runner	20	16
tcint	191	126
watch	11	12

This limited implementation already leads to improvements over earlier analyses [7] in most of the tested example cases. Still, the analysis is not as exact as it could be. In two cases (mca200 and watch) we are slightly worse than the graph based approach, because the interface approach so far does not distinguish between immediate and delayed aborts. The implementation could also be improved, *e. g.*, by unbundling outgoing thread transitions and other heuristics. The theory could be further strengthened, *e. g.*, by directly integrating abortion in the control flow graph.

VI. Conclusion and Future Work

We introduced an interface algebra for compositional analysis of WCRT in synchronous multi-threading and illustrated this with a small, sequential example. The full report [4] expands on this in several areas, notably on the handling of delay nodes and concurrency, and on how to trade off precision against efficiency by interface bundling.

This algebraic approach is very flexible: from considering all possible data, which gives an exact WCRT for the price of possible exponential computation time, to abstracting from all internal behavior, which is very fast but might lead to a large over-approximation, all levels of exactness can be applied. Beside the handling of control data, the more systematic treatment of parallel execution leads to tighter WCRTs. Since the interfaces are compositional, we should also be able to get a better performance for the WCRT computation on large programs. Furthermore, data-dependencies with arbitrary precision can be easily expressed in the interface algebra, to rule out impossible executions and get an even tighter WCRT. The flexibility for modularization and abstraction together with the tight semantic coupling of numeric and functional information are the main advantages over previous approaches on Boolean timing analysis [19].

Encouraged by our experimental results, we want to fully implement a suite of WCRT algorithms based on the new interfaces. We would like to extend our interfaces to cover exit-traps and non-immediate aborts as well. A further step would be to integrate thread priorities into the interfaces, to reduce the number of considered paths. At the moment, abortions are handled by adding transitions to all pause nodes inside them. It might be more natural to extend the control flow graphs by hierarchy to directly express the hierarchical nature of abortions. This should easily be captured by our interfaces. A central part of future work on the hardware side will consist in developing a formal operational execution model of KEP and verifying our WCRT algorithms with respect to this model.

References

[1] S. Edwards and E. A. Lee, "The case for the Precision Timed (PRET) machine," EECS Dept., Univ. of Calif., Berkeley, Technical Report No. UCB/EECS-2006-149, Nov. 2006.

[2] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone, "The Synchronous Languages Twelve Years Later," in *Proc. IEEE, Special Issue on Embedded Systems*, vol. 91, Jan. 2003, pp. 64–83.

[3] G. Berry and L. Cosserat, "The ESTEREL Synchronous Programming Language and its Mathematical Semantics," in *Seminar on Concurrency, Carnegie-Mellon University*. Springer LNCS 197, 1984, pp. 389–448.

[4] R. von Hanxleden, M. Mendler, and C. Traulsen, "WCRT algebra and scheduling interfaces for Esterel-style synchronous multi-threading," Christian-Albrechts-Univ. Kiel, Dept. of Comp. Sci., Tech. Rep. 0807, Jun. 2008, http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-0807.pdf.

[5] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The determination of worst-case execution times—overview of the methods and survey of tools," *ACM Trans. on Embed. Comp. Syst. (TECS)*, vol. 7, no. 3, 2008.

[6] X. Li, J. Lukoschus, M. Boldt, M. Harder, and R. von Hanxleden, "An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis," in *Proc. CASES 2005*. ACM Press, 2005, pp. 225–236.

[7] M. Boldt, C. Traulsen, and R. von Hanxleden, "Worst case reaction time analysis of concurrent reactive programs," *ENTCS*, vol. 203, no. 4, pp. 65–79, 2008, proc. SLA++P 2007.

[8] G. Logothetis and K. Schneider, "Exact high level WCET analysis of synchronous programs by symbolic state space exploration," in *DATE 2003*. IEEE Computer Society, 2003, pp. 196–203.

[9] C. André, F. Boulanger, M.-A. Péraldi, J. P. Rigault, and G. Vidal-Naquet, "Objects and synchronous programming," *Europ. J. on Automated Syst.*, vol. 31, no. 3, pp. 417–432, 1997.

[10] O. Hainque, L. Pautet, Y. L. Biannic, and E. Nassor, "Cronos: A separate compilation toolset for modular Esterel applications." in *World Congress on Formal Methods*, J. M. Wing, J. Woodcock, and J. Davies, Eds. Springer LNCS 1709, 1999, pp. 1836–1853.

[11] E. A. Lee, H. Zheng, and Y. Zhou, "Causality interfaces and compositional causality analysis," in *Foundations of Interface Technologies (FIT'05)*, ser. ENTCS. Elsevier, 2005.

[12] E. Wandeler and L. Thiele, "Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling," in *Proc. EMSOFT'05*, 2005.

[13] T. Henzinger and S. Matic, "An interface algebra for real-time components," in *Proc. RTAS 2006*. IEEE Computer Society, 2006, pp. 253–266.

[14] F. L. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat, *Synchronisation and Linearity*. John Wiley & Sons, 1992.

[15] J. L. Boudec and P. Thiran, *Network Calculus - A theory of deterministic queuing systems for the internet*. Springer LNCS 2050, 2001.

[16] J. Benkoski and A. J. Strojwas, "Timing verification by formal signal interaction modeling in a multi-level timing simulator," in *Design Automation Conference*, 1989, pp. 668–673.

[17] S. Devadas, K. Keutzer, and S. Malik, "Delay computation in combinational logic circuits: Theory and algorithms," in *International Conference on Computer-Aided Design*, 1991, pp. 176–179.

[18] J. P. M. M. Silva and K. A. Sakallah, "An analysis of path sensitization criteria," in *Proc. ICCD*, 1993, pp. 68–72.

[19] K. C. Lam and R. K. Brayton, *Timed Boolean Functions. A Unified Formalism for Exact Timing Analysis*. Kluwer, 1994.

[20] M. Mendler, "Characterising combinational timing analyses in intuitionistic modal logic," *The Logic Journal of the IGPL*, vol. 8, no. 6, pp. 821–853, 2000.

[21] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. of Comp. Progr.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[22] E. A. Lee, "The problem with threads," *IEEE Computer*, vol. 39, no. 5, pp. 33–42, 2006. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MC.2006.180

[23] D. Potop-Butucaru, S. A. Edwards, and G. Berry, *Compiling Esterel*. Springer, May 2007.

[24] C. Berg, J. Engblom, and R. Wilhelm, "Requirements for and design of a processor with predictable timing," in *Perspectives Workshop: Design of Systems with Predictable Behaviour*, L. Thiele and R. Wilhelm, Eds., Schloss Dagstuhl, Germany, 2004.

[25] R. von Hanxleden, X. Li, P. Roop, Z. Salcic, and L. H. Yoong, "Reactive processing for reactive systems," *ERCIM News*, vol. 66, pp. 28–29, Oct. 2006.

[26] X. Li, M. Boldt, and R. von Hanxleden, "Mapping Esterel onto a multi-threaded embedded processor," in *Proc. ASPLOS'06*, 2006.

[27] S. Yuan, S. Andalam, L. H. Yoong, P. S. Roop, and Z. Salcic, "Starpro—a new multithreaded direct execution platform for Esterel," in *Proc. SLA++P 2008*, Budapest, Hungary, 2008.

[28] O. Tardieu and S. A. Edwards, "Scheduling-independent threads and exceptions in SHIM," in *EMSOFT '06: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*. New York, NY, USA: ACM, 2006, pp. 142–151.

[29] G. Lüttgen and M. Mendler, "Towards a model-theory for Esterel," in *SLAP 2002*, ser. ENTCS, F. Maraninchi, A. Girault, and E. Rutten, Eds., vol. 65,5. Elsevier Science, 2002.

[30] "Estbench Esterel Benchmark Suite," 2007, http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz.