INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

WCRT Algebra and Scheduling Interfaces for Esterel-Style Synchronous Multithreading

Preliminary Report

Reinhard von Hanxleden
Michael Mendler
Claus Traulsen

Bericht Nr. 0807

July 8, 2008

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

WCRT Algebra and Scheduling Interfaces for Esterel-Style Synchronous Multithreading *Preliminary Report*

Reinhard von Hanxleden
Michael Mendler
Claus Traulsen

Bericht Nr. 0807
July 8, 2008

E-mail: rvh@informatik.uni-kiel.de,
michael.mendler@uni-bamberg.de,
ctr@informatik.uni-kiel.de

Abstract. The abstractions used in system design typically limit themselves to encapsulate and guarantee functionality, not timing. Hence, it is very difficult to transfer results on timing behavior across layers, *e. g.*, from the application level through the operating system level to the hardware level. The choice of the model of computation plays a big role in facilitating this transfer. In the realm of reactive systems, the synchronous model of computation has some appeal here, as it inherently limits the number of operations per reaction, and addresses concurrency and preemptive behavior at the language level.
Recently, reactive processing architectures have been proposed as execution platform for synchronous languages, notably Esterel. Initially, these architectures were driven by the desire for high performance with low resource usage, including low power consumption. However, by now they have also demonstrated their benefits in terms of predictability. Preliminary work on worst case reaction time (WCRT) analysis has been promising—fairly simple heuristics already achieve an accuracy typically in the 30–40% range. However, these methods so far lack formal grounding, and do not exploit knowledge about signal consistency etc. To provide a formal basis for WCRT analysis, we here propose a type-theoretic, algebraic approach. This approach not only allows to verify the correctness of WCRT analyses methods, but also opens the door for more exact analyses, as it allows to capture functionality and timing precisely and to trade off precision against analysis effort.
This approach is still under development; this report presents first results on suitable interface types and the proper characterization of instantaneous nodes, delay nodes and concurrency. As a concrete application, it builds on a multi-threaded Esterel processor, the Kiel Esterel Processor (KEP).

Key words: Worst Case Reaction Time analysis, Interface Algebra, Synchronous Languages, Esterel, Multithreading, Reactive Processing, Kiel Esterel Processor

Table of Contents

1 Introduction

Reconciling performance and predictability in embedded systems is a challenge that spans all layers of hard- and software development. The use of abstraction layers, for example distinguishing the operating system (OS) layer from an application software layer above and a hardware layer below, is fundamental to computer science. 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 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]. This model supports deterministic timing in several ways, as further argued in Section 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, and we illustrate this with the case of multi-threaded execution of Esterel-style synchronous programs. The main contribution of this paper then is the introduction if an algebraic framework for precisely capturing WCRT characteristics for this execution approach.

2 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], but 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 classical 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. In other words, 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 [4], which assumes a weaker (non-strict) 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 [5]).

2.1 Reactive Processing

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 compile and run the resulting program on a standard micro processor [6]. 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 should not have unbounded loops, for example. However, depending on the synthesis approach used, the control flow may still be rather complex and difficult to analyze (for example, computed gotos). Furthermore, standard processors typically employ various techniques that improve average execution time, at the expense of worst case execution time and predictability [7].

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 [8]. There have been various proposals on how to support concurrency in reactive processing, including sequentialization [9], parallel execution [10], and, most recently, multi-threading [11, 12]. 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, a combination of static scheduling, hardware-supported context switching, and fixed machine instruction execution times assures timing determinism. This has been exploited in a compiler which translates Esterel programs into multi-threaded assembler code, and as part of the compilation process analyzes the WCRT in terms of instruction cycles [13]. The assembler code is then annotated with this WCRT, which at run time configures a hardware unit, the TickManager, which provides timing stability at the logical tick level [14]. This decouples physical system reaction times from the tick-specific computation requirements, which is for example desirable for control loop stability. Note that, as an alternative, one may still choose not to make use of the TickManager and instead let the processor run freely, *i. e.*, start the next tick as soon as the current tick finishes. This can improve average case performance, at the expense of reaction time jitter and possibly higher power requirements. Furthermore, our experiments indicate that there is no dramatic difference between WCRT and average case reaction time (ACRT)—typically a factor of around 1.5 [13].

The WCRT analysis technique developed so far does already provide fairly promising results. A relatively simple heuristics provides an accuracy typically in the 30–40% range [13]. However, this heuristics still makes conservative and simplifying assumptions, and is not grounded in a formal timing model. We believe that such a formal model will be instrumental in the further development of these techniques to cover different processing platforms and several levels of abstraction between hardware and software layers. A semantically grounded notion of WCRT interface types will be crucial to make our techniques scale up to industrial-sized systems without losing tight control of correctness and exactness of WCRT analysis. In this paper we take first steps towards such an interface model.

3 A Type-Theoretic Approach to WCRT Analysis

Imperative synchronous programming as exemplified in Esterel, Statecharts provide predictability in terms of determinism and bounded reaction in the face of powerful language constructs for concurrency, state hierarchy, priorities, strong and weak preemption. These constructs induce sophisticated control dependencies so that WCRT analysis for such languages and processors that directly support them is non-trivial. The risk of over- and under-approximations jeopardizes the quality of WCRT analysis and thus creates a conflict between performance and predictability.

Combining performance with predictability involves a trade-off between analysis time and execution time. Optimizations in the efficiency of timing analysis are paid for by compromising the quality of timing results. Under-approximations may result in a loss of coverage and over-approximations in a loss of exactness. Thus, the system under analysis (SUA) becomes less predictable or runs at slower (virtual) clock speed. Optimizations in speed and predictability of the SUA, on the other hand, require sophisticated data-dependent analysis which are computationally expensive. This creates a direct conflict with the performance and predictability of the WCRT algorithms themselves. To strike this trade-off a scalable and modular approach towards timing analysis is called for in which precision and efficiency can be adjusted systematically in wide margins.

The existing WCRT algorithms such as [13] are neither compositional nor scalable in terms of precision. They are global analyses on the complete control-flow graph of a monolithic program. Also they 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

– give precise statements about exactness and coverage of timing values, supporting a variety of timing abstractions;
– are dedicated to express the implicit control flow of imperative synchronous programming languages;
– 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. It combines max-plus algebra $(\mathbb{N}, \mathit{max}, +, 0, -\infty)$, see e.g. [15], with Boolean logic[1] to reason about implicit control-flow.

The key element of WCRT analysis is the interchange (distribution) of max and $+$. In its simplest form, the WCRT is the maximum of all sums of paths delays, an expression of the form $\mathit{max}(\sum_{i \in p_1} d_{i1}, \sum_{i \in p_2} d_{i2}, \ldots, \sum_{i \in p_n} d_{in})$ where p_j are execution paths of the system and d_{ij} the delay of path segment i in path p_j. However, the number n of paths is exponential in the number of elementary nodes of a system. Practicable WCRT analyses therefore reduce the *max-of-sums* to the polynomial complexity of *sum-of-maxs* employing various forms of dependency abstraction. Unfortunately, the obvious distribution of $\mathit{max}(d_1 + e_1, d_1 + e_2, d_2 + e_1, d_2 + e_2) = \mathit{max}(d_1, d_2) + \mathit{max}(e_1, e_2)$ is exact only if we have a full set of path combinations. In general, there will be dependencies ruling out certain paths, in which case we only get conservative over-approximations, e.g., $\mathit{max}(d_1, d_2) + \mathit{max}(e_1, e_2) \geq \mathit{max}(d_1 + e_1, d_2 + e_1, d_2 + e_2)$. On the other hand, $\mathit{max}(d_1 + e_1, d_2 + \mathit{max}(e_1, e_2)) = \mathit{max}(d_1 + e_1, d_2 + e_1, d_2 + e_2)$ which eliminates one addition operation, does work in this case. The art of WCRT analysis consists in finding a judicious trade-off between forming the maximum operation early in order to aggregate data and refining dependency paths for the sake of exactness. A practicable WCRT algebra must be able to express and control this trade-off. In Sec. 5.1 we sketch a type theory which achieves this by coupling timing delays d with logic formulas ϕ. A pair $d : \phi$ specifies the semantical meaning of d within the control-flow of a program. Logical operations on the formulas then go hand-in-hand with arithmetic operations on timing. E.g., suppose a schedule activates control points X and Y with a delay of d_1 and d_2 instruction cycles. If they are independent then both control points are jointly active within the maximum of both delays, i.e., $(d_1 : X) \wedge (d_2 : Y) \cong (\mathit{max}(d_1, d_2) : X \wedge Y)$. On the other hand, if reaching Y is causally dependent on having reached X first then we must take addition, i.e., $(d_1 : X) \wedge (X \supset (d_2 : Y)) \cong (d_1 + d_2 : X \wedge Y)$ where $\wedge$, $\supset$ are logical conjunction and implication, respectively. Thus, in general, the computation of paths involves functional analysis of implicit control-flow using logic reasoning rather than just following pointers in a graph (of explicit dependencies). We will illustrate this in Secs. 5.2–5.4 below by way of an extended example.

The idea of modularizing embedded systems programming and specifically synchronous programming using interfaces is not new. Mostly these interface models focus on causality issues, which amounts to dependency analysis without quantitative time. On the other hand, there exist numerous approaches to classical WCET analysis [16] but only few on WCRT analysis [14].

Logothetis, Schneider and Metzler [17, 18] 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. Instead, in reactive system WCRT one considers how long it may take to compute a single logical tick. WCRT is an orthogonal issue to WCET and has been rarely investigated in the literature so far.

André *et al.* [19] employ a causally simple notion of module in the sense that no instantaneous interaction between modules is permitted. Such a model is not suitable for WCRT. Hainque *et*

[1] To be more precise, we use a finite-valued Heyting algebra in which $A \vee \neg A$ is not a tautology. This difference, however, is not essential for this paper.

al. [20] use a topological abstraction of the underlying circuit graphs (or syntactic structure of Boolean equations) to derive a fairly rigid component dependency model. A component is assumed executable iff *all* of its inputs are available; after component execution *all* of its outputs become defined. The former restriction means that single-threaded execution cannot be modeled compositionally. The interface model also does not cover data dependency and thus cannot deal with dynamic schedules and does not support WCRT, either.

The causality interfaces of Lee *et al.* [21] are more flexible. These are functions $\delta : P_i \times P_o \to D$ associating with every pair $(i, o) \in P_i \times P_o$ of input and output ports an element $\delta(i, o) \in D$ of a *dependency domain* D, which expresses if and how an output o depends on input i. The domain D is a linearly ordered dioid structure $(D, 0, 1, \otimes, \oplus, <)$, where $\otimes$ models *sequential composition* and $\oplus$ is the *parallel* composition of dependencies, with neutral elements 0 and 1, respectively. Causality analysis is then performed by multiplication on the global system matrix describing the dependencies between any two signals. Using an appropriate dioid structure D, one can perform the analyses of [20] 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. In other words, 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 Lee's to address these deficiencies.

Similar restrictions apply to recent work [22, 23] combining network calculus [15, 24] with real-time interfaces in which *sequential connection* and *concurrent composition* operators play an analogous role to Lee's sequential and parallel composition, respectively. On top of that, these works are concerned with the compositional modeling of *regular execution patterns* rather than *stabilization processes* that make up the components *inside* each execution cycle of a synchronous program. Existing interface theories [21–23], which aim at the verification of resource constraints for real-time scheduling, handle rather delicate timing properties such as task execution latency, arrival rates, resource utilization, throughput, accumulated cost of context switches, and so on. On the other hand, in those works, the dependency on data and control flow is largely abstracted. For instance, since the task sequences of Henzinger and Matic [23] 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.* [21] in that it permits only one-to-one associations of inputs with outputs. The interfaces of Wandeler and Thiele [22] for modular performance analysis in real-time calculus are like those of Henzinger and Matic [23] but without sequential composition of tasks and thus do not model control flow.

AND- and OR-type synchronization dependencies are important for WCRT in synchronous programming since reachability of control nodes in general depends both conjunctively and disjunctively on the presence of data. Moreover, execution may depend on the absence of data (negative triggering conditions), which makes compositional modeling rather a delicate matter in the presence of logical feedback loops. This severely limits the applicability of existing interface models. The assume-guarantee style specification [22, 23] does not address causality issues arising from feedback and negative triggering conditions. The interface automata of Alfaro, Henzinger, Lee, Xiong [25, 26] model synchronous macro-states and assume that all stabilization processes (sequences of micro-states) can be abstracted into atomic interaction labels. The introduction of *transient states* [27] alleviates this to some extent, but the focus is still on regular (scheduling) behavior. The situation is different, however, for cyclic systems, in which causality information is needed. The interface problem in WCRT for stabilization processes is quite a different game—it is simpler and more complex at the same time. It is simpler since we do not need sophisticated resource and timing models; in a stabilization process the only resource is time—here instruction cycles—which is only consumed once rather than in a regular pattern. It is more complex since we need to model more sophisticated synchronization effects and solve causality and full-abstraction problems due to feedback and negative dependencies. Because of the complications arising from causality issues, there is currently no robust component model for imperative synchronous programming.

Before we move on introducing our approach in the next section let us add a couple of general remarks on our philosophy. Our approach to WCRT is specification-oriented and type-theoretic. The idea we wish to stress is that timing does not have any meaning without at the same time specifying the functional behavior to which the timing information is attached. Without function, timing are just numbers signifying nothing. Working with (matrices of) numbers alone makes is easy to lose track of semantical correctness and exactness and thus the connection between performance and predictability. For illustration consider the following analogy: Imagine we are given a full blueprint of all layers (metal, poly-silicon, etc.) of a VLSI solid-state circuit. We know all its physical parameters and could simulate and fabricate the circuit. Yet, we would not be able to sensibly tell what its timing is without knowing what the inputs and outputs are and what function the circuit is supposed to execute. So just as correct functionality depends on timing, correct timing depends on functionality. By using the type-theoretic structure $f : \phi$ we bring out both the separation and the intimate coupling of scheduling bounds f and scheduling types ϕ. The scheduling shape ϕ acts as a type specification for timing matrix f making clear the semantical meaning of f and the scheduling bound f in turn acts as a quantitative implementation of the schedule ϕ. We believe that type theory seen as an extension of algebra provides a powerful framework for coupling functional and non-functional specifications that has not yet been exploited to its full potential.

4 Example of Esterel-style Multi-threading, KEP Assembler

Esterel [3] programs communicate with the environment and internally via signals, which are either present or absent during one instant. Signals are set present by the `emit` statement and tested with the `present` test. They are reset at the start of each instant. Esterel statements can be either combined in sequence (`;`) or in parallel (`||`). The `loop` statement simply restarts its body when it terminates. All Esterel statements are considered instantaneous, except for the `pause` statement, which pauses for one instant, and derived statements like `halt=loop pause`, which stops forever. Esterel supports multiple forms of preemption, *e. g.*, via the `abort` statement, which simply terminates its body when some trigger signal is present. Abortion can be either weak or strong. Weak abortion permits the execution of its body in the instant the trigger signal becomes active, strong abortion does not. Both kinds of abortions can be either immediate or delayed. The immediate version already senses for the trigger signal in the instant its body is entered, while the delayed version ignores it during the first instant in which the abort body is started.

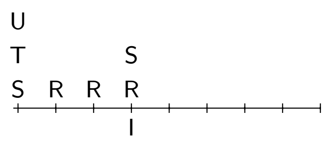
Consider the Esterel fragment in Figure 1a. It consists of two threads. The first thread (G) emits signals `R`, `S`, `T` depending on some input signal `I`. In any case, it emits signal `U` and terminates instantaneously. The thread (H) continuously emits signal `R`, until signal `I` occurs. Thereafter, it either halts, when `E` is present, or emits `S` and terminates otherwise.

The main problems when executing Esterel on standard processors are the handling of preemption and synchronous parallelism. The Kiel Esterel Processor (KEP) [11] handles abortion by watchers, which are executed in parallel with their body and simply set the program-counter when the trigger signal becomes present. Synchronous parallelism is executed by multi-threading. The KEP manages multiple threads, each with their own program counter and a priority. In each instruction cycle, the processors determines the active instruction from the thread with the highest priority and executes it. New (sub-)threads are initialized by the `PAR` instruction. The `PARE` instruction ends the initialization of parallel threads and sets the program counter of the current thread to the corresponding join. By changing the priorities of the threads, arbitrary interleavings can be specified; the compiler has to ensure that the priorities respect all signal dependencies, *i. e.*, all possible emits of a signal are performed before any testings of the signal. The specific interleavings are not relevant for the WCRT analysis under multi-threading. Therefore, priority changing instructions may be treated like the padding statement nothing which has no effect on the system state other than adding a time delay. Additionally, for all parallel threads one join instruction is executed, which checks whether all threads have terminated in the current instant. If this is the case, the whole parallel terminates and the join transforms the control to the next instruction. Otherwise the join blocks. Instructions pause and halt end the execution for the current instant, the execution is resumed from the same point in the next instant.

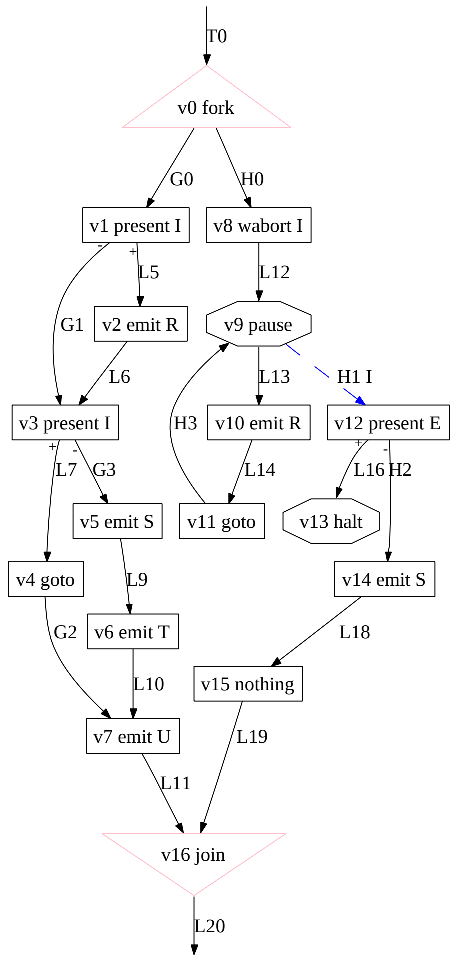
Most instructions, like emit or entering an abort block, are executed in exactly one instruction cycle. The pause instruction is executed both in the instant it is entered, and in the instant it is resumed, to check for weak and strong abortions, respectively. Note that the halt instruction

```
[ % thread G
  present I then
    emit R
  end present;
  present I else
    emit S;
    emit T
  end present;
  emit U;
||
  % thread H
  weak abort
    loop
      pause;
      emit R
    end loop
  when immediate I;
  present E then
    halt
  end present;
  emit S;
  nothing;
]
```

(a) Esterel program T



(b) Execution Trace



(c) CKAG

```
L01: T0: PAR 1,G0,1
L02:     PAR 1,H0,2
L03:     PARE A1

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

L11: H0: WABORT I,H1
L12: H3: PAUSE
L13:     EMIT R
L14:     GOTO H3
L15: H1: PRESENT E,H2
L16:     HALT
L17: H2: EMIT S
L18:     NOTHING

L19: A1: JOIN
```

(d) KEP Assembler

```
Tick 1:
  L01,L02,L03,
    L11,L12,
    L04,L06,L08,L09,L10,
  L19
Tick 2:
  L12,L13,L14,L12
Tick 3:
  L12,L13,L14,L12
Tick 4:
  L12,L13,L14,L12,
  L15,L17,L18,L19
```

(e) KEP sample Trace

Fig. 1: A simple Esterel program T with its corresponding control-flow graph and the resulting KEP Assembler

is executed in one cycle. While halt and loop pause are functionally equivalent, their execution times differ. The latter has a worst case reaction time (WCRT) of 3 instruction cycles (ics) while the former only needs 1 ic.

The concurrent KEP assembler graph (CKAG, see Fig. 1c) captures the control flow, both standard control and abortions, of an Esterel program. We distinguish two kinds of edges, instantaneous and non-instantaneous. Instantaneous edges can be taken immediately when the source node is entered, they reflect control flow starting from instantaneous statements or weak abortions of delayed statements. Non-instantaneous edges can only be taken in an instant where the control started in its source node, like control flow from pause statements or strong abortions. The CKAG can be derived from the Esterel program by structural translation. For a given CKAG, the generation of KEP Assembler is straight-forward (see Fig. 1c). Most nodes are translated into one instruction, only fork nodes are expanded to multiple instructions to initialize the threads. In our example, the fork $v0$ is transformed into three instructions ($L01 - L03$).

5 WCRT Interfaces at Work

The WCRT of an Esterel program is the maximal number of instructions that are executed during one instant. WCRT differs from WCET fundamentally in that it deals with the timing of *stabilization* rather than *iteration* processes. WCRT assumes that all dependencies in the control flow are acyclic and the propagation of control is a monotonic process in which each atomic control point is only ever executed at most once. On the other hand, WCRT for Esterel-style synchronous processing must handle *non-atomic control flow* including features such as hierarchical and concurrent threads, priorities and preemption. In the following we sketch the basic elements of WCRT for synchronous processing and develop an algebra for modular timing analyses. This algebra is an extension and adaptation of the *intuitionistic propositional stabilization theory* presented in [28].

5.1 Introducing Interface Types

An *execution* σ is a finite and monotonically increasing sequence $\sigma = \emptyset \subseteq \sigma(0) \subseteq \sigma(1) \subseteq \sigma(2) \subseteq \cdots \subseteq \sigma(n-1)$ of sets of *control signals* $\sigma(i) \subseteq \mathbb{S}$ $(0 \leq i < |\sigma| = n)$ called *events*. This includes the special case $n = 0$ of the *empty execution* $\sigma = \emptyset$. For all executions, empty or not, we refer to the initial $\emptyset$ as $\sigma(-\infty)$. Signals $\mathbb{S}$ contain the control-flow labels as well as input and output signals of the program. E.g. for the program in Fig. 1c $\{L0 - L20, G0 - G3, H0 - H3, I, E, R, S, T, U\} \subseteq \mathbb{S}$. We will also need 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 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. Accordingly, the number of instructions executed by σ is the number of changes or steps $|\sigma| - 1$ rather than the number of events.

Example 1. One possible execution of the program T in Fig. 1c would be as follows. Initially, control signal $T0$ is set, so $\sigma(0) = \{T0\}$. Then the PAR and PARE instructions making up the fork node v_0 are executed in line numbers L01, L02, L03 each taking one ic. The two PAR instructions set up internal counters for thread control, which does not change the set of events in the signals of Fig. 1c, which are the signals that we are interested in. Hence, $\sigma(1) = \sigma(2) = \{T0\}$. However, after the PARE both control signals $G0$, $H0$ become present bringing threads G and H to life. This means $\sigma(3) = \{T0, G0, H0\}$. The next instruction could be any of the two first instructions of G or H. As it happens, the KEP Assembler Fig. 1d assigns higher priority to H so that our execution continues with wabort (node v_8), i.e., $\sigma(4) = \{T0, G0, H0, L12\}$. This brings up the pause instruction v_9. Now, depending on whether signal I is present or not the execution of pause either moves to v_{12} (weak immediate abort) or terminates. Let us assume the latter, i.e., $\sigma(5) = \{T0, G0, H0, L12\}$ which is the same set as $\sigma(4)$ but now thread H is finished up for the instant and has entered an implicitly wait state. The execution continues with the first instruction of G, the present node v_1 at label $G0$. Since I is assumed absent, its execution effects a jump to label $G1$, i.e., $\sigma(6) = \{T0, G0, H0, L12, G1\}$. Thereafter, we run sequentially through nodes v_3, v_5, v_6, v_7 giving $\sigma(7) = \{T0, G0, H0, L12, G1, G3\}$, $\sigma(8) = \{T0, G0, H0, L12, G1, G3, L9\}$, $\sigma(9) = \{T0, G0, H0, L12, G1, G3, L9, L10\}$. Executing the final emit instruction v_7 hits the join at entry $L11$, so that $\sigma(10) = \{T0, G0, H0, L12, G1, G3, L9, L10, L11\}$. Now both threads G and H are finished. It takes one execution step of the join node v_{16} to detect this and to terminate the synchronous instant of T with the final event $\sigma(11) = \{T0, G0, H0, L12, G1, G3, L9, L10, L11\}$. Overall, we get an execution $\sigma = \sigma(0)\sigma(1)\cdots\sigma(11)$ of the outer-most main thread of T from $T0$ consisting of 11 instruction cycles.

Note that signal $L19$ is not included in the last event $\sigma(11)$ because control remains inside the pause node v_9 of T. Only in the next logical instant when T is resumed in v_9 and thread H eventually comes out at control point $L19$ (if signal I is present and E absent), then executing the join v_{16} will bring us to control point $L19$ and out of T instantaneously.

Note further that the difference $\Delta_i = \sigma(i+1) \setminus \sigma(i)$ may be an arbitrary subset of $\mathbb{S}$. It may be empty as with $i \in \{1, 4, 10\}$, contain exactly one control as for $i \in \{3, 5, 6, 7, 8, 9\}$ or more as in $i = 2$. In general, Δ_i will encompass more than one signal when a 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 set of executions S defines a *schedule*. The possible schedules of a program will be specified by a *scheduling type* ϕ generated by the language

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

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 statement that "*A is active (is present, has been traversed, is scheduled) in all executions of the schedule*". The constant *true* is validated by all schedules and *false* only by the empty schedule or the schedule which contains only the empty execution. The type operators $\neg$, $\supset$ are (intuitionistic) negation and implication. The two 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.

We will keep matters brief and present only some essential constructions in this theory. In fact, a fragment of the language will suffice to define interface types for KEP programs as far as they are treated in this paper. To begin with, define a

– *basic control type* to be 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 σ*) and $\neg A$ (*A absent throughout σ*). 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, $A \in \sigma(0)$ or never becomes active, $A \notin \sigma(|\sigma| - 1)$. 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 reveals the intuitionistic nature of negation which is crucial to handle the semantics of synchronous languages in a compositional and fully abstract way [29, 30]. For special signals like activation of nodes $\mathit{active}(v)$ it is safe to assume $\mathit{active}(v) \oplus \neg \mathit{active}(v) \cong \mathit{true}$ since these are state signals and 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 block. When used as an output we need to express that ζ occurs delayed after some maximal number of ics, d say. We write $\sigma, d \models \zeta$ or $\sigma \models d : \zeta$ for this as an abbreviation 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. We will see that this may be exploited for optimizations in WCRT analysis. The limit cases are $+\infty : \mathit{false} \cong \mathit{true}$ and $-\infty : \mathit{false} \cong \mathit{false}$. The specification $\mathit{wait} =_{\mathit{df}} 1 : \mathit{false}$ is of particular interest. It says that an execution has at most one event, i.e., $\sigma \models \mathit{wait}$ 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 $\sigma(0) \subseteq \mathbb{S}$. We permit *wait* to be used as a third constant besides *true* and *false* inside basic controls. The reaction time of an execution σ may then either be specified as $\sigma \models d : \mathit{wait}$ or $\sigma \models d + 1 : \mathit{false}$ 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 ics. The selection of which ζ_j is activated is expressed by $\oplus$ since it 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 one bound for all exits ζ_j.
– An *input control* is an expressions $\phi = \zeta_1 \vee \zeta_2 \vee \cdots \vee \zeta_m$ where the disjunction $\vee$ refers to the external non-determinism resolved by the environment which determines how a program block 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 regarding executions: $S \models \zeta_1 \vee \zeta_2$ requires $\exists i \in \{1, 2\}. \forall \sigma \in S.\ \sigma \models \zeta_i$ and 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.

Finally, we build (input-output) *interface types* for KEP 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. In other words, the ζ_i and ξ_j represent logical input and output lines of the program. 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 an interface type ϕ together with a timing matrix D. The former describes the qualitative aspect of scheduling, the latter captures the quantitative 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.

Let us look at how interface types are used. Figure 2 depicts a program fragment T abstracted into a reactive box with input and output controls. The paths inside T seen in Figure 2 illustrate the four ways in which a reactive block T may participate in the execution of a logical tick: Threads may (a) pass straight through the block 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 block and eventually (instantaneously) leave through some exit control ξ, or (d) start inside the block 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.

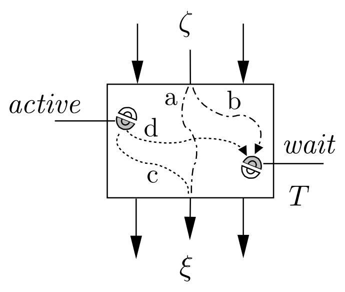


Fig. 2: The four types of thread paths: through path (a), sink path (b), source path (c), internal path (d).

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

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

If one of the paths does not exist its associated delay is set to $-\infty$. A block T can be classified according to the paths that are executable in it, i.e., that have $d_{type} \geq 0$ (rather than $d_{type} = -\infty$) for $\mathit{type} \in \{\mathit{thr}, \mathit{src}, \mathit{snk}, \mathit{int}\}$. Specifically, we call T a

- *through* node, if $d_{thr} \geq 0$, and N_{thr} the set of all through nodes
- *source* node, if $d_{src} \geq 0$, and N_{src} the set of all source nodes
- *sink* node, if $d_{snk} \geq 0$, and N_{snk} the set of all sink nodes
- *internal* node, if $d_{int} \geq 0$, and N_{int} the set of all internal nodes.

A *delay node* is a node with at least one non-instantaneous path, *i. e.*, $N_{del} = N_{src} \cup N_{snk} \cup N_{int}$. A *strong delay node* is a delay node without any through path, hence $N_{sdel} = N_{del} \setminus N_{thr}$. A *transient node* is a through node that contains only through paths, i.e., $d_{src} = d_{snk} = d_{int} = -\infty$. Thus $N_{trans} = N_{thr} \setminus N_{del}$. We assume that each cyclic dependency loop in the program is broken by at least one strong delay node.

It is useful to classify the exits of a node T according to the type of path on which they can appear. We call an exit *instantaneous* if it can *only* be activated on through paths (type a) and *non-instantaneous* if it can be reached *only* by source paths (type c). The successor nodes reached by them are referred to accordingly as *instantaneous successors* and *non-instantaneous successors* of T. In the KEP assembler graphs of [13] two other types of successor nodes are distinguished, the *control successors* and *exit successors*. Since control successors are activated either by through paths (type a) or source paths (type c) they constitute the general case from the WCRT scheduling point of view. The exit successors, which are introduced by exit-trap blocks, are activated exclusively as part of through paths. Thus, they are instantaneous successors in our terminology.

Often we split the interface T into its two parts $T = [T_{srf}, T_{dpt}]$, the *surface interface* and the *depth interface*

$$T_{srf} = \begin{pmatrix} d_{thr} \\ d_{snk} \end{pmatrix} : \zeta \supset (\circ\xi \oplus \circ wait) \qquad T_{dpt} = \begin{pmatrix} d_{src} \\ d_{int} \end{pmatrix} : active \supset (\circ\xi \oplus \circ wait).$$

By logical transformations of interfaces various optimizations can be achieved. For instance, for transient nodes we reduce T_{srf} as follows:

$$\begin{aligned} T_{srf} &\cong (d_{thr}, -\infty)^T : \zeta \supset (\circ\xi \oplus \circ wait) &&\cong \zeta \supset ((d_{thr} : \xi) \oplus (-\infty : wait)) \\ &\cong \zeta \supset (d_{thr}{:}\xi \oplus ((-\infty + 1) : false)) &&\cong \zeta \supset (d_{thr}{:}\xi \oplus (-\infty : false)) \\ &\cong \zeta \supset (d_{thr}{:}\xi \oplus false) &&\cong \zeta \supset (d_{thr} : \xi) \\ &\cong (d_{thr}) : \zeta \supset \circ\xi, \end{aligned}$$

while $T_{dpt} = (-\infty, -\infty)^T : active \supset (\circ\xi \oplus \circ wait) \cong true$ is simply dropped. Moreover, without loss of generality we may suppose that T_{srf} is *normalized* so it satisfies $d_{thr} \leq d_{snk}$ for all sink nodes T. Otherwise, if $d_{snk} < d_{thr}$ we would have

$$\begin{aligned} (d_{thr} : \xi) \oplus (d_{snk} : wait) &\cong (d_{thr} : \xi) \oplus (d_{snk} + 1 : false) \\ &\preceq (d_{thr} : \xi) \oplus (d_{snk} + 1 : \xi) \\ &\cong max(d_{thr}, d_{snk} + 1) : \xi \\ &\cong d_{thr} : \xi, \end{aligned}$$

where $\phi \preceq \psi$ means that all executions satisfying ϕ also satisfy ψ. In the other direction, it trivially holds that $d_{thr} : \xi \preceq (d_{thr} : \xi) \oplus (d_{snk} : wait)$. Hence, whenever $d_{snk} < d_{thr}$ the two types $(d_{thr} : \xi) \oplus (d_{snk} : wait)$ and $(d_{thr} : \xi)$ are equivalent which means essentially that the sink paths are redundant and thus could be pruned. Operationally, if $d_{snk} < d_{thr}$ then the through path $(d_{thr}) : \zeta \supset \circ\xi$ of T dominates the WCRT while the sink path $(d_{snk}) : \zeta \supset \circ wait$ in T cannot contribute to the longest execution. In this case we might as well assume $d_{snk} = -\infty$, i.e., that T is not a sink node at all. The same arguments apply to the depth interfaces T_{dpt}, for which we may thus assume $d_{src} \leq d_{int}$ or otherwise $d_{int} = -\infty$.

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

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

with a WCRT matrix D of shape $n \times m$. The terminology above can be applied, *mutatis mutandis*, to such general controls. 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 theory. The basic controls appearing in ϕ_i, ψ_i describe the causal dependencies between the blocks T_i. In its general form, WCRT analysis amounts to a transformation

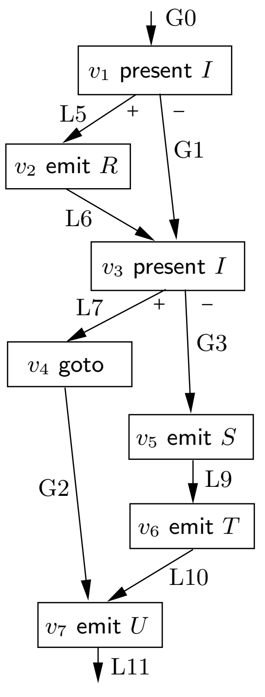
$$\bigwedge_i D_i : \phi_i \supset \psi_i \quad \preceq \quad D : \phi \supset \psi \tag{2}$$

in which the individual timing interfaces 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 the 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 $\zeta =_{df} \bigoplus_{i \in I} \zeta_i$

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). 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).

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. In general, the right-hand side of (2) will include more executions than the left-hand side. E.g., this occurs naturally whenever the composite type $\phi \supset \psi$ does not include all internal signals mentioned in the types $\phi_i \supset \psi_i$. Then the right-hand side of (2) does not constrain the executions on those internal signals while the left-hand side still restricts them. Many other types of abstractions are possible as we shall see.

How is the composition (2) performed? In general, we can use any sound and complete logical calculus for WCRT type theory as described, e.g., in [28]. For our special interface types this calculus reduces to matrix multiplication in max-plus algebra $(\mathbb{N}, +, max, 0, -\infty)$ [15] combined with logical reasoning on basic controls ζ, which is a slight generalization of Boolean algebra. We will explain this in the following sections by way of our running example from Fig. 1. More details on the semantic theory of WCRT algebra can be found in the appendix, Sec. A.



(a)

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

(b)

```
// KEP 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
```

(c)

Fig. 3: Control Flow G (a), Esterel (b), KEP Assembler (c).

5.2 Instantaneous Behavior: Transient Nodes

Let us begin by illustrating the role of type specifications for WCRT in the single-threaded execution of transient nodes. Although in this case WCRT analysis is equivalent to computing longest paths and straight-forward even without timing interfaces, it will give a first idea of the power of types for modularization and abstraction. This will lay out the playground in which we can later deal with Esterel-style pausing, preemption, multi-threading, etc. General synchronous flow graphs live at a higher level of abstraction in which control dependencies are more implicit and thus WCRT analysis no longer identical to the longest path problem.

Our example is the sequential program G of Fig. 3 which is the fragment of Fig. 1c consisting of nodes v_1–v_7. Each of them is a KEP assembler instruction 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.

Let us assume that each of the basic instructions present, emit, goto take 1 processor cycle regardless how it is entered or exited. Their timing interfaces are then specified as follows:

$$v_1 = \begin{pmatrix}1\\1\end{pmatrix} : G0 \supset \circ L5 \oplus \circ G1 \qquad v_2 = (1) : L5 \supset \circ L6 \quad v_3 = \begin{pmatrix}1 & 1\\1 & 1\end{pmatrix} : (L6 \vee G1) \supset \circ L7 \oplus \circ G3$$
$$v_4 = (1) : L7 \supset \circ G2 \qquad v_5 = (1) : G3 \supset \circ L9 \quad v_6 = (1) : L9 \supset \circ L10$$
$$v_7 = \begin{pmatrix}1 & 1\end{pmatrix} : (G2 \vee L10) \supset \circ L11.$$

Logically speaking, the problem of WCRT for G amounts to obtaining the tightest bound d such that $d : G0 \supset \circ L11$ is semantically entailed by this theory, i.e., by $\bigwedge_{i=1}^{7} v_i$.

Weaving Paths The naive strategy would be to enumerate all paths from $G0$ to $L11$, sum up the delays on each path and then take the maximum. There are four paths $p_1 = G0\,L5\,L6\,L7\,G2\,L11$, $p_2 = G0\,G1\,L7\,G2\,L11$, $p_3 = G0\,L5\,L6\,G3\,L9\,L10\,L11$ and $p_4 = G0\,G1\,G3\,L9\,L10\,L11$. Each of these paths defines a sub-graph of G with specific side-inputs and side-outputs. For instance, p_1 as indicated in Fig. 4 has the side-outputs G1, G3 and side-inputs G1, L10 so that its full scheduling type is $G0 \vee L10 \vee G1 \supset \circ G1 \oplus \circ G3 \oplus \circ L11$. This type says that if p_1 gets activated through control edges $G0$, $L10$ or $G1$ then it must be terminated through one of the exits $G1$, $G3$ or $L11$. The timing matrix associated with this type is

$$D_1 : G0 \vee L10 \vee G1 \supset \circ G1 \oplus \circ G3 \oplus \circ L11$$

where

$$D_1 = \begin{pmatrix}1 & -\infty & -\infty\\ 3 & -\infty & 1\\ 5 & 1 & 3\end{pmatrix}.$$

The entries $-\infty$ in D_1 mean that there is no causal control flow from the corresponding input to the corresponding output line. D_1 can be obtained by successively multiplying (in max-plus algebra) the timing matrices of the individual nodes traversed by p_1. We explain below in Sec. 5.2 how this is done. At this point note that for WCRT of G we are not actually interested in the exact timing of the side-inputs, i.e., the fact that p_1 can also be executed from $L10$ and $G1$. One way to suppress this information is by pre-composing D_1 with the timing map $(0, -\infty, -\infty)^T : G0 \supset \circ G0 \oplus \circ L10 \oplus \circ G1$ giving

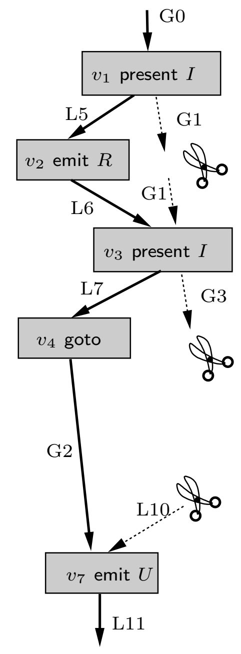


Fig. 4: The path p_1 in G.

$$p_1' = D_1' : G0 \supset \circ G1 \oplus \circ G3 \oplus \circ L11 \tag{3}$$

where

$$D_1' = D_1 \cdot \begin{pmatrix}0\\-\infty\\-\infty\end{pmatrix} = \begin{pmatrix}1 & -\infty & -\infty\\ 3 & -\infty & 1\\ 5 & 1 & 3\end{pmatrix} \cdot \begin{pmatrix}0\\-\infty\\-\infty\end{pmatrix} = \begin{pmatrix}1\\3\\5\end{pmatrix}.$$

The type (3) stipulates that every execution which enters path p_1' through input $G0$ either leaves p_1 through exit $G1$ in at most 1 ic, through exit $G3$ in at most 3 ic or through $L11$ within no more than 5 ics. D_1' in contrast to D_1 no longer records the exact delay between particular combinations of inputs and outputs of p_1. For instance, the fact that $L11$ is reached in p_1 from $L10$ in only 1 ic of time and from $G1$ in at most 3 ics instead of 5, is lost. Similarly, the information that from input $L10$ we cannot reach output $G3$ at all, indicated by the entry $-\infty$ in D_1 is not present in D_1' any more.

In talking about executions along p_1 we also assume that the path is not exited through $G1$ or $G3$. Can we suppress the references to side-outputs $G1$ and $G3$ on the right of $\supset$ of the scheduling type (3)? Well, not directly, because if we simply drop them the resulting scheduling type $G0 \supset \circ L11$ would guarantee that *all* executions entering p_1 through $G0$ eventually come out at $L11$. This is obviously false. However, we can fix this using negations and conjunctions. The right type is $(G0 \wedge \neg G1 \wedge \neg G3) \supset \circ L11$ which strengthens the assumption $G0$ by $\neg G1 \wedge \neg G3$ to the effect that only executions starting in $G0$ not involving $G1$ or $G3$ are considered. Indeed we can construct a timing for this type and justify its semantic correctness as follows: First, by purely logical reasoning on types (not involving any matrix calculations) we argue that D_1' which has type $G0 \supset \circ G1 \oplus \circ G3 \oplus \circ L11$ also must have type $(G0 \wedge \neg G1 \wedge \neg G3) \supset \circ \mathit{false} \oplus \circ \mathit{false} \oplus \circ L11$ and then compose with the sound schedule $(-\infty, -\infty, 0) : (\mathit{false} \vee \mathit{false} \vee L11) \supset \circ L11$ to obtain

$$p_1 = D_1'' : (G0 \wedge \neg G1 \wedge \neg G3) \supset \circ L11 \quad \text{where} \quad D_1'' = \begin{pmatrix} -\infty & -\infty & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 3 \\ 5 \end{pmatrix} = (5). \tag{4}$$

which is the proper type specification of path p_1 without side-inputs and side-outputs. For the other paths we get in a similar fashion

$$p_2 = D_2'' : (G0 \wedge \neg L5 \wedge \neg G3) \supset \circ L11 \qquad D_2'' = (4) \tag{5}$$
$$p_3 = D_3'' : (G0 \wedge \neg G1 \wedge \neg L7) \supset \circ L11 \qquad D_3'' = (6) \tag{6}$$
$$p_4 = D_4'' : (G0 \wedge \neg L5 \wedge \neg L7) \supset \circ L11 \qquad D_4'' = (5). \tag{7}$$

The path schedules (4)–(7) can now be woven together to obtain the final result $G = (6) : G0 \supset \circ L11$. First, recall that $[D_1, D_2] : (\phi_1 \vee \phi_2) \supset \circ L11$ is the sum of $D_1 : \phi_1 \supset \circ L11$ and $D_2 : \phi_2 \supset \circ L11$. So we can combine (3)–(7) as $D'' =_{\mathit{df}} [(5), (4), (6), (5)] = (5, 4, 6, 5)$ with the type

$$D'' : ((G0 \wedge \phi_1) \vee (G0 \wedge \phi_2) \vee (G0 \wedge \phi_3) \vee (G0 \wedge \phi_4)) \supset \circ L11 \tag{8}$$

in which $\phi_1 =_{\mathit{df}} \neg G1 \wedge \neg G3$, $\phi_2 =_{\mathit{df}} \neg L5 \wedge \neg G3$, $\phi_3 =_{\mathit{df}} \neg G1 \wedge \neg L7$ and $\phi_4 =_{\mathit{df}} \neg L5 \wedge \neg L7$. We pre-compose (8) with the timing $(0, 0, 0, 0)^T : \oplus_{i=1}^4 (G0 \wedge \phi_i) \supset \oplus_{i=1}^4 \circ (G0 \wedge \phi_i)$:

$$(5, 4, 6, 5) \cdot (0, 0, 0, 0)^T = (6) : ((G0 \wedge \phi_1) \oplus (G0 \wedge \phi_2) \oplus (G0 \wedge \phi_3) \oplus (G0 \wedge \phi_4)) \supset \circ L11. \tag{9}$$

Then consider that by distributivity $\oplus_{i=1}^4 (G0 \wedge \phi_i)$ is logically equivalent to $G0 \wedge \oplus_{i=1}^4 \phi_i$. Moreover, under single-threaded execution (and only then[2]) the type equivalence

$$\oplus_{i=1}^4 \phi_i = (\neg G1 \wedge \neg G3) \oplus (\neg L5 \wedge \neg G3) \oplus (\neg G1 \wedge \neg L7) \oplus (\neg L5 \wedge \neg L7) \equiv \mathit{true}$$

holds in G since every execution thread must make a split decision for either exit L5 or G1 at node v_1 and for either L7 or G3 at node v_3. Hence, every thread satisfies one of the summands $\neg G1 \wedge \neg G3$, $\neg L5 \wedge \neg G3$, $\neg G1 \wedge \neg L7$ or $\neg L5 \wedge \neg L7$. Taking all together gives $\oplus_{i=1}^4 (G0 \wedge \phi_i) \equiv G0 \wedge \oplus_{i=1}^4 \phi_i \equiv G0 \wedge \mathit{true} \equiv G0$ and thus (9) turns into $(6) : G0 \supset \circ L11$ which finally is the WCRT of graph G.

Weaving Nets WCRT analysis by enumeration of paths, though sound, is of worst-case exponential complexity. A more efficient way of going about is to exploit dynamic programming techniques. In the following we illustrate this process in terms of the decomposition of G seen in Fig. 5. The strategy is to propagate WCRT information forward through G describing and composing (in this case) sub-nets $N1$, $N2$, $N3$ rather than paths. In doing so we extend the scheduling interfaces appropriately in order to make the matrices match up:

[2] For multi-threaded execution several exits may be taken in one and the same run by different threads. Also observe that, in contrast to $\oplus$, the disjunction $\phi_1 \vee \phi_2 \vee \phi_3 \vee \phi_4$ is not equivalent to *true*! This would hold under *static* and *deterministic* scheduling in which *all* executions are in one of the schedules ϕ_i. Since the exits from statements v_1 and v_3 are data-dependent, different executions may choose different paths.

1. We obtain the scheduling type of $N1$ by combining v_1 of type $G0 \supset \circ L5 \oplus \circ G1$ with v_2 of type $L5 \supset \circ L6$. To compose them we first lift v_2 to type $L5 \vee G1 \supset \circ L6 \oplus \circ G1$ by pre-multiplying it with the embedding $(0, -\infty)^T : L6 \supset \circ L6 \oplus \circ G1$ giving

$$v_2' = \begin{pmatrix} 0 \\ -\infty \end{pmatrix} \cdot v_2 = \begin{pmatrix} 0 \\ -\infty \end{pmatrix} \cdot (1) = \begin{pmatrix} 1 \\ -\infty \end{pmatrix} : L5 \supset \circ L6 \oplus \circ G1.$$

Next we combine v_2' with $(-\infty, 0)^T : G1 \supset \circ L6 \oplus \circ G1$ to give

$$v_2'' = \left[\begin{pmatrix} 1 \\ -\infty \end{pmatrix}, \begin{pmatrix} -\infty \\ 0 \end{pmatrix} \right] = \begin{pmatrix} 1 & -\infty \\ -\infty & 0 \end{pmatrix} : L5 \vee G1 \supset \circ L6 \oplus \circ G1,$$

where as before $[D_1, D_2] : (\phi_1 \vee \phi_2) \supset \psi$ is the sum of $D_1 : \phi_1 \supset \psi$ and $D_2 : \phi_2 \supset \psi$. The shapes of matrices v_1 and v_2'' now connect up and we get $N1 = v_2'' \cdot v_1$:

$$N1 = \begin{pmatrix} 1 & -\infty \\ -\infty & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} : G0 \supset \circ L6 \oplus \circ G1.$$

2. Next we construct the WCRT of block $N2 : (L6 \vee G1) \supset \circ G2 \oplus \circ G3$ by composing $v_3 : (L6 \vee G1) \supset \circ L7 \oplus \circ G3$ with $v4 : L7 \supset \circ G2$. With the help of $(0, -\infty)^T : G2 \supset \circ G2 \oplus \circ G3$ and $(-\infty, 0)^T : G3 \supset \circ G2 \oplus \circ G3$ we extend the schedule of v_4 to become of type $(L7 \vee G3) \supset \circ G2 \oplus \circ G3$ which can then be composed with v_3 as follows:

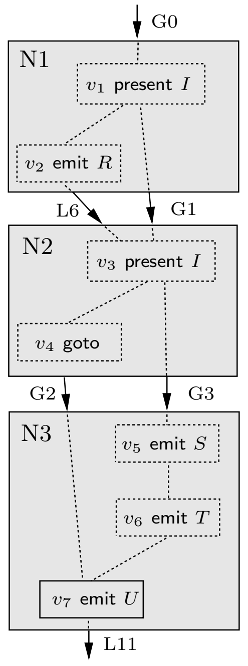


Fig. 5: Decomposition of G.

$$\begin{aligned} N2 &= \left[\begin{pmatrix} 0 \\ -\infty \end{pmatrix} \cdot v_4, \begin{pmatrix} -\infty \\ 0 \end{pmatrix} \right] \cdot v_3 = \left[\begin{pmatrix} 0 \\ -\infty \end{pmatrix} \cdot (1), \begin{pmatrix} -\infty \\ 0 \end{pmatrix} \right] \cdot \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & -\infty \\ -\infty & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 2 \\ 1 & 1 \end{pmatrix} : (L6 \vee G1) \supset \circ G2 \oplus \circ G3. \end{aligned}$$

3. The third step is to build $N3 : (G2 \vee G3) \supset \circ L11$ from $v_5 : G3 \supset \circ L9$, $v_6 : L9 \supset \circ L10$ and $v_7 : (G2 \vee L10) \supset \circ L11$. v_5 and v_6 can be directly multiplied: $v_6 \cdot v_5 = (1) \cdot (1) = (2) : G3 \supset \circ L10$. Then we lift $v_6 \cdot v_5$ to type $(G2 \vee G3) \supset \circ G2 \oplus \circ L10$ and pre-multiply with v_7:

$$\begin{aligned} N3 &= v_7 \cdot \left[\begin{pmatrix} 0 \\ -\infty \end{pmatrix}, \begin{pmatrix} -\infty \\ 0 \end{pmatrix} \cdot v_6 \cdot v_5 \right] = v_7 \cdot \begin{pmatrix} 0 & -\infty \\ -\infty & 2 \end{pmatrix} \\ &= (1\ 1) \cdot \begin{pmatrix} 0 & -\infty \\ -\infty & 2 \end{pmatrix} = (1\ 3) : (G2 \vee G3) \supset \circ L11. \end{aligned}$$

4. If we compose the three sub-nets $N1$, $N2$, $N3$ in sequence, our schedule of G all the way from entry point $G0$ to exit $L11$ is complete:

$$G = N3 \cdot N2 \cdot N1 = (1\ 3) \cdot \begin{pmatrix} 2 & 2 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 1 \end{pmatrix} = (1\ 3) \cdot \begin{pmatrix} 4 \\ 3 \end{pmatrix} = (6) : G0 \supset \circ L11.$$

This is indeed the weight of the longest path p_3.

Bundling Abstractions There are of course other ways of arriving at (an approximation of) the WCRT, corresponding to different network decompositions of G. It is also possible to condense the timing information by *bundling* the inputs and outputs of $N1$, $N2$, $N3$ *before* they are composed. For instance, one might decide to compress the scheduling type of $N1$ into a single entry-exit delay $N1' : G0 \supset \circ(L6 \oplus G1)$ which specifies the worst-case delay for an execution entering through $G0$ to come out at $L6$ or $G1$, without distinguishing between threads exiting on $L6$ and those exiting on $G1$. This is applied also to N2 and N3 as indicated in Fig.6.

This compression is done algebraically by pre-composing $N1$ with $(0,0) : (L7 \vee G1) \supset \circ(L7 \oplus G1)$ which yields $N1' = (0,0) \cdot N1 = (0,0) \cdot (2,1)^T = (2) : G0 \supset \circ(L6 \oplus G1)$. In the same way, we could compress $N2$ and $N3$ to

$$N2' = \begin{pmatrix}0 & 0\end{pmatrix} \cdot \begin{pmatrix}2 & 2\\ 1 & 1\end{pmatrix} \cdot \begin{pmatrix}0\\ 0\end{pmatrix} = (2) : (L6 \oplus G1) \supset \circ(G2 \oplus G3)$$

$$N3' = \begin{pmatrix}1 & 3\end{pmatrix} \cdot \begin{pmatrix}0\\ 0\end{pmatrix} = (3) : (G2 \oplus G3) \supset \circ L11.$$

Each of the scheduling types N_i' is a max-abstraction of the original interface N_i. This is seen from the fact that semantically $N_i \subseteq N_i'$, i.e., each execution satisfying N_i is also an execution under schedule N_i'. Thus, the timing value associated with N_i' is an upper bound of all schedules in N_i. Yet, it is not exact because N_i' is properly larger than N_i. For instance, N_1' contains an execution of duration 2 which exits on $G1$ while N_1 (exactly) states that all threads leaving through $G1$ consume no more than 1 ic. The same is true of the other sub-nets N_2' and N_3'. As a result of this imprecision the composition $N3' \cdot N2' \cdot N1' = (2) \cdot (2) \cdot (3) = (7)$ yields an over-approximation of G's WCRT.

Why would we want to abstract the sub-nets in this way and lose exactness? The answer is that composing $N1'$, $N2'$, $N3'$ is more efficient since it involves only scalars rather than matrices.

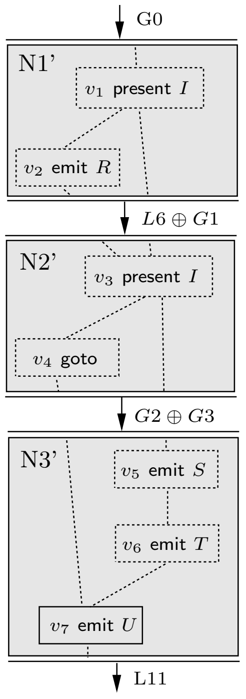


Fig. 6: Bundle Abstraction of G.

Data Dependency and Degrees of Precision A full and exact 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 for single-threaded scheduling of G would be

$$G = (6, 4, 3, 1) : (G0 \vee G1 \vee G3 \vee G2) \supset \circ L11.$$

This is still not the exact description of G since it does not express the dependency of the WCRT on signal I. If I is present then all threads must take control edges L5 and L7 rather than G1 or G3 which are blocked. If I is absent then both G1 and G3 must be taken instead. As a result the longest path $p_3 = G0L5L6G3L9L10L11$ with delay 6 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:

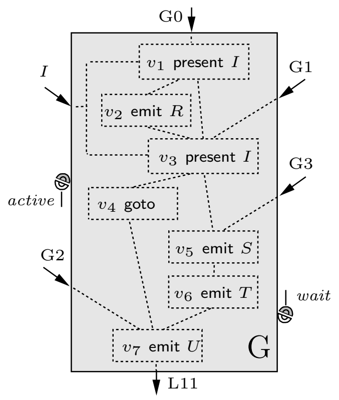
$$G = (5, 5, 3, 4, 3, 1) : ((G0 \wedge I) \vee (G0 \wedge \neg I) \vee (G1 \wedge I) \vee (G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11. \quad (10)$$

The inclusion of signal I in the interface has now resulted in the distinction of two different delay values 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 $6 : G0 \supset \circ L11$ from above. Assuming that input signal I is causally stable, i.e., $I \oplus \neg I \cong \mathit{true}$ it is possible to optimize the interface without losing precision: since $(G0 \wedge I) \oplus (G0 \wedge \neg I) \cong G0 \wedge (I \oplus \neg I) \cong G0 \wedge \mathit{true} \cong G0$ the map $(0,0)^T : G0 \supset \circ(G0 \wedge I) \oplus \circ(G0 \wedge \neg I)$ is sound and can be used to compress the two entries of value 5 in (10) into a single value $5 = \mathit{max}(5,5)$ giving

$$G = (5, 3, 4, 3, 1) : (G0 \vee (G1 \wedge I) \vee (G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11.$$

In the same vein, but this time without referring to stability, we could further bundle $G1 \wedge I$ and $G3$ into a single control with the single delay $(3) : ((G1 \wedge I) \oplus G3) \supset \circ L11$ at the same level precision.

Still, if we only ever intend to use G as a composite transient node $G0 \supset \circ L11$ the typing $G = (5) : G0 \supset \circ L11$ might be sufficient. The different WCRT type specifications of G are summarized in Fig. 7. Considering that G is a transient block, i.e., it does not have any delay nodes, the depth interface of G is trivial $G_{dpt} = (-\infty, -\infty)^T : \mathit{active} \supset \circ L11 \oplus \circ \mathit{wait}$. Though trivial it will be useful when we integrate G in a fork-join block uniformly as described in the next section.



$$
\begin{aligned}
(6) &: G0 \supset \circ L11 \quad \text{(worst-case abstraction from } I\text{)} \\
(6,4,3,1)^T &: (G0 \vee G1 \vee G3 \vee G2) \supset \circ L11 \\
(5,3,4,3,1)^T &: (G0 \vee (G1 \wedge I) \vee (G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11 \\
(5,3,4,1)^T &: (G0 \vee ((G1 \wedge I) \oplus G3) \vee (G1 \wedge \neg I) \vee G2) \supset \circ L11 \\
(5) &: G0 \supset \circ L11 \quad \text{(accounting for dependency on } I\text{)}
\end{aligned}
$$

Fig. 7: Component G and some of its WCRT Types with varying precision and conciseness.

All the operations performed above are supported by semantically sound proof rules in WCRT type theory. The logical manipulation of types often can be also 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 our 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. Some aspects of the latter will be expounded in the following sections.

5.3 Sequential Behavior: Delay Nodes

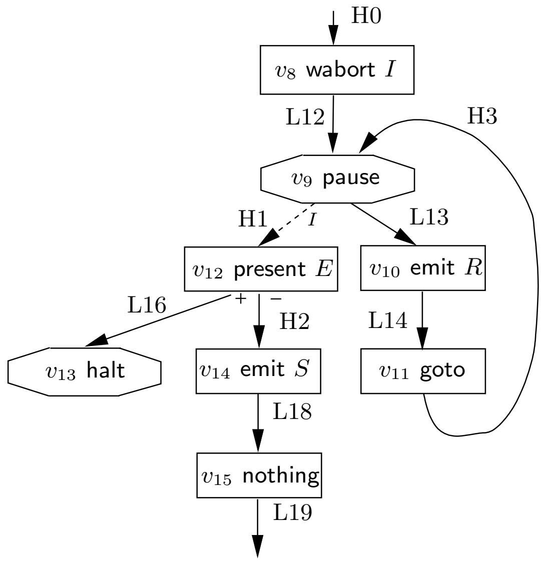
Now we take a look at sequential control flow which initiates and terminates in pause and halt nodes. We illustrate how these are related to the scheduling types *active* and *wait*. We use the example seen in Fig. 8 which is the fragment of nodes v_8–v_{15} from our running example in Fig. 1a. Nodes wabort, emit, goto, present, nothing are transient and specified as before in Sec. 5.2. But now the instantaneous paths are broken by the delay nodes v_9 and v_{13}.

Consider the pause node v_9. It can be entered by two controls, the line number L12 and program label H3 and left via two kinds of exits, a non-instantaneous edge L13 and a instantaneous exit H1 (weak abortion). When a control thread enters v_9 then either it terminates the current instant inside the node or leaves through the weak abort H1 (data-dependent, if signal I is present) continuing the current instant, instantaneously. A thread entering v_9 never exits through L13 in the same instant. On the other hand, if a thread is started (resumed) from inside the pause node v_9 then control can only exit through L13. This suggests to specify the pause node by the following pair of scheduling types:

$$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} : (H3 \vee L12) \supset \circ H1 \oplus \circ \mathit{wait} \tag{11}$$

$$(1) : \mathit{active}(v_9) \supset \circ L13 \tag{12}$$

The specification (11) says that if pause is entered through H3 or L12 it can be left through H1 or terminate (*wait*) inside the pause. In both cases execution takes 1 instruction cycle, either



(a)

```
Esterel Program H:
weak abort
    loop
        pause;
        emit R
    end loop
when immediate I;
present E then
    halt
end present;
emit S;
nothing;
```

(b)

```
KEP FRAGMENT H
L11: H0: WABORT I,H1
L12: H3: PAUSE
L13:     EMIT R
L14:     GOTO H3
L15: H1: PRESENT E,H2
L16:     HALT
L17: H2: EMIT S
L18:     NOTHING
```

(c)

Fig. 8: Sequential Control Flow H (a), Esterel program (b) and KEP Assembler (c).

to move the program counter forward to H1 or to reach an internal wait state. Since there are no differences in the delays we could bundle the inputs $H3$, $L12$ and compress the matrix as $(1,1)^T : (H3 \oplus L12) \supset \circ H1 \oplus \circ wait$ or even $(1) : (H3 \oplus L12) \supset \circ(H1 \oplus wait)$ without losing information over (11). Still, we could do even better and record the dependency of control on signal I, with the more precise type

$$\begin{pmatrix} 1 & -\infty \\ -\infty & 1 \end{pmatrix} : ((H3 \oplus L12) \wedge I) \vee ((H3 \oplus L12) \wedge \neg I) \supset \circ H1 \oplus \circ wait.$$

This separates the threads which must stop inside the pause from those which must leave via H1 due to a weak immediate abort on signal I.

The specification (12) accounts for threads starting in the pause which must necessarily pass control to L13 within one instruction cycle. This is why (12) does not include a source path of type $active(v_9) \supset \circ H1$. In a similar way we can model strong or non-immediate aborts. For delayed (non-immediate) weak abort, there is no path from L12 to H1. For strong abort, the abortion can not be taken in the instant when v_9 is entered, thus there is no path from H3 or L12 to H1.

The halt node v_{13} in Fig. 8 (equivalent to an infinitely pausing loop pause, but faster) is not only a sink for control threads entering through L16 but it also has an internal path of length 1 (which is repeated at every instant). It is a strong delay node and specified by

$$(1,1) : (active(v_{13}) \vee L16) \supset \circ wait. \tag{13}$$

Now let us determine the WCRT of G, essentially refining the strategy informally described in [13]. First, generalizing the notion of surface interfaces we define the *surface reaction time* of a control edge A of graph H as the (component-wise) smallest vector $A.srf$ such that $A.srf : A \supset \circ L19 \oplus \circ wait$ is derivable from the H's WCRT theory.[3] Obviously, the value $H0.srf$ is the surface interface H_{srf} of program H in Fig. 8 seen as a reactive box with (sole) input H0. Following the strategy of the WCRT algorithm of [13] the computation of $H0.srf$ proceeds by depth-first search forward from H0: We compose node v_8 of type $(1) : H0 \supset \circ L12$ with the

[3] In [13] this number is denoted $A.inst$ and computed for *complete* programs in which all paths are terminated by pause or halt nodes. Dangling exits like L19 of Fig. 8 are not considered. Also, the WCRT is defined for nodes rather than edges as we do here. These are minor differences, however.

through part $(1,1)^T : L12 \supset \circ H1 \oplus \circ wait$ of v_9's interface given in (11) to get $(1,1)^T \cdot (1) = (2,2)^T : H0 \supset \circ H1 \oplus \circ wait$. This reduces the computation of $H0.srf$ to that of computing $H1.srf : H1 \supset \circ L19 \oplus \circ wait$ via $H0.srf = [H1.srf, (-\infty, 0)^T] \cdot (2,2)^T$, where $[H1.srf, (-\infty, 0)^T]$ is the lifting of $H1.srf : H1 \supset \circ L19 \oplus \circ wait$ to type $(H1 \vee wait) \supset \circ L19 \oplus \circ wait$ so it can be composed with $(2,2)^T$. As shown below we eventually get $H1.srf = (3,2)^T$ and thus

$$\begin{aligned} H_{srf} = H0.srf &= [H1.srf, (-\infty, 0)^T] \cdot (3,2)^T \\ &= \begin{pmatrix} 2 & -\infty \\ 2 & 0 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 2 \end{pmatrix} = \begin{pmatrix} 5 \\ 4 \end{pmatrix} : H0 \supset \circ L19 \oplus \circ wait \end{aligned} \tag{14}$$

which tells us that the longest through path (H0 L12 H1 H2 L18 L19) has 5 ics and the longest sink path (H0 L12 H1 L16) consumes 4 ics.

Like the surface interfaces, the depth interfaces, too, may be generalized: Each node v has a *depth reaction time*[4] in H, $v.dpt$, which is the smallest vector m derivable such that $m : active(v) \supset \circ L19 \oplus \circ wait$. It characterizes the maximal duration of an instant which starts inside v either leaving at exit L19 or terminating in the pause or halt nodes of H. It only needs to be computed for source nodes and internal nodes. For all others we have $v.d_{src} = v.d_{int} = -\infty$ which implies $v.dpt = -\infty$, too. In graph H of Fig. 8 only the pause node v_9 and the halt node v_{13} have a source or internal WCRT.

For instance, the depth interface of (13) gives delay $1 : active(v_{13}) \supset \circ wait$ for the halt. Since this internal path is the only way to start in v_{13} and eventually terminate or exit H, the depth reaction time of v_{13} is

$$v_{13}.dpt = (-\infty, 1) : active(v_{13}) \supset \circ L19 \oplus \circ wait.$$

In contrast, the depth interface (12) of the pause node v_9 only contains a source path $(1) : active(v_9) \supset \circ L13$. All source paths $active(v_9) \supset \circ L19 \oplus \circ wait$ of H must have this source path as their prefix. So, we must first determine the worst case delay of type $L13 \supset \circ L19 \oplus \circ wait$, which is nothing but $L13.srf$. Again by depth-first search we get $L13.srf$ from functional composition

$$L13.srf = H3.srf \cdot (1) \cdot (1) = H3.srf \cdot (2) \ : \ L13 \supset \circ L19 \oplus \circ wait \tag{15}$$

from nodes v_{10} of type $(1) : L13 \supset \circ L14$ and v_{11} of type $(1) : L14 \supset \circ H3$ together with the surface delay $H3.srf : H3 \supset \circ L19 \oplus \circ wait$. From (11) we extract $(1,1)^T : H3 \supset \circ H1 \oplus \circ wait$. This type specifies two ways for termination. Either directly inside the pause or continuing via $H1.srf : H1 \supset \circ L19 \oplus \circ wait$. We can compute the latter using v_{12}'s type $(1,1)^T : H1 \supset \circ H2 \oplus \circ L16$ as follows: We first combine v_{13}'s surface type $(1) : L16 \supset \circ wait$ from (13) and both v_{14}'s interface $(1) : H2 \supset \circ L18$ and v_{15}'s interface $(1) : L18 \supset \circ L19$ to give

$$[\begin{pmatrix} 0 \\ -\infty \end{pmatrix} \cdot (1) \cdot (1), \begin{pmatrix} -\infty \\ 0 \end{pmatrix} \cdot (1)] = \begin{pmatrix} 2 & -\infty \\ -\infty & 1 \end{pmatrix} : (H2 \vee L16) \supset \circ L19 \oplus \circ wait$$

and from there get

$$H1.srf = \begin{pmatrix} 2 & -\infty \\ -\infty & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \end{pmatrix} : H1 \supset \circ L19 \oplus \circ wait.$$

We embed this information canonically as $[H1.srf, (-\infty, 0)^T] : (H1 \vee wait) \supset \circ L19 \oplus \circ wait$ and compose with $(1,1)^T : H3 \supset \circ H1 \oplus \circ wait$ for the final result

$$H3.srf = [H1.srf, (-\infty, 0)^T] \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 & -\infty \\ 2 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 3 \end{pmatrix} : H3 \supset \circ L19 \oplus \circ wait.$$

Now we plug this into (15) and get

$$L13.srf = H3.srf \cdot (2) = \begin{pmatrix} 4 \\ 3 \end{pmatrix} \cdot (2) = \begin{pmatrix} 6 \\ 5 \end{pmatrix} \ : \ L13 \supset \circ L19 \oplus \circ wait.$$

[4] This number is denoted $v.next$ in [13].

Finally, the depth reaction time of v_9 is obtained by composing $L13.srf$ with $(1) : active(v_2) \supset \circ L13$ obtaining

$$v_9.dpt = L13.srf \cdot (1) = (7,6)^T : active(v_9) \supset \circ L19 \oplus \circ wait.$$

Having $v_9.dpt$ and $v_{13}.dpt$ available we can compute the depth interface of block H: Since the instant can start in the pause node v_9 or the halt node v_{13} the *active* control of H is $active = active(v_9) \oplus active(v_{13})$. So, we are looking for the smallest bound for type $(active(v_9) \oplus active(v_{13})) \supset \circ L19 \oplus \circ wait$. Formally, we get this by combining $v_9.dpt$ and $v_{13}.dpt$ into the matrix $[v_9.dpt, v_{13}.dpt] : (active(v_9) \vee active(v_{13})) \supset \circ L19 \oplus \circ wait$ and then pre-composing with $(0,0)^T : (active(v_9) \oplus active(v_{13})) \supset \circ active(v_9) \oplus \circ active(v_{13})$. This gives

$$H_{dpt} = [v_9.dpt, v_{13}.dpt] \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix} = max(v_9.dpt, v_{13}.dpt) = \begin{pmatrix} 7 \\ 6 \end{pmatrix} \ : \ active \supset \circ L19 \oplus \circ wait.$$

In general, H_{dpt} is computed as the (component-wise) maximum over all depth reaction times for all delay nodes in H. This is perfectly uniform in the interfaces of H's nodes since we could well include all nodes into the maximum, $max\{v_i.dpt \mid 8 \leq i \leq 14\}$, postulating $active(H) = \bigoplus_{i=8}^{14} active(v_{14})$. However, since $v_i.dpt = -\infty$ for nodes different from v_9 and v_{13}, all nodes except the delay nodes will happily drop out of the global maximum.

Note on Optimization The WCRT algorithm presented in [13] is somewhat more generous about surface and depth interfaces than what we have discussed above. Here we take these interfaces to have output control $\circ L19 \oplus \circ wait$ distinguishing the WCRT of through and source paths ending in $L19$ from those of sink and internal paths ending in *wait*. The algorithm [13] (implicitly by hard-coding maximum operations) merges both types of path into the single control $\circ L19$. This is sound since the types $\circ L19 \oplus \circ wait$ and $\circ L19$ are equivalent in the sense that every schedule well-timed for one is also well-typed for the other. For instance, we have $d_1 : L19 \oplus d_2 : wait \preceq d_1 : L19 \oplus d_2 + 1 : false \preceq d_1 : L19 \oplus d_2 + 1 : L19 \preceq max(d_1, d_2 + 1) : L19$ and $d : L19 \preceq d : L19 \oplus -\infty : wait$. Provided[5] $d_1 > d_2$, the optimization $\circ L19 \oplus \circ wait$ to $\circ L19$ does not involve any loss of precision for WCRT due to the maximum operation.

In our example program H this optimization can be exploited as follows: The surface type of v_{13} is reworked $(1) : L16 \supset \circ wait \cong (2) : L16 \supset \circ false \preceq (2) : L16 \supset \circ L19$ which is then combined with $(2) : H2 \supset \circ L19$ (obtained from the surface types of v_{14} and v_{15}) to give $[(2),(2)] = (2,2) : (H2 \vee L16) \supset \circ L19$. This is multiplied with node v_{12}'s interface $(1,1)^T : H1 \supset \circ H2 \oplus \circ L16$ to produce the surface type $H1.srf = (2,2) \cdot (1,1)^T = (3) : H1 \supset \circ L19$. From there we get $H3.srf = (4) : H3 \supset \circ L19$ and finally $v_9.dpt = (7) : active(v_9) \supset \circ L19$ without ever having to include possible paths ending in *wait*. The reason is that these paths may be safely pruned in the search.

On the other hand, suppose the halt node v_{13} encapsulated an entire sub-system which took 100 ics to reach termination on the sink path inside (rather than just 1). Applying the same approximating optimization which replaces $(100) : L16 \supset \circ wait$ with $(100) : L16 \supset \circ L19$ we would get $H1.srf = (2,100) \cdot (1,1)^T = (101) : H1 \supset \circ L19$ and further $v_9.dpt = (105) : active(v_9) \supset \circ L19$ rather than the more exact $v_9.dpt = (7,105)^T : active(v_9) \supset \circ L19 \oplus \circ wait$. The latter is more faithful with respect to instantaneous paths that may be added from control point L19 onwards in the external context of G (specifically, parallel compositions of G with other programs) which should not be weighted with delay 105 but 7. This is actually a source of over-approximation in the existing algorithm [13]. Keeping track of the scheduling type (the difference between $\circ L19 \oplus \circ wait$ and $\circ L19$) can help to control the trade-off between efficiency and precision.

5.4 Concurrent Behavior: Fork and Join

[5] We used this special condition in Sec. 5.1 to normalize interfaces

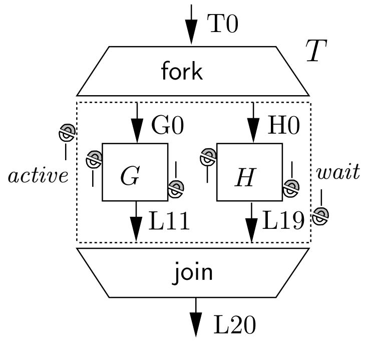


Fig. 9: Program T from Fig. 1a in which threads G and H are executed concurrently.

Consider Fig. 9 in which the two sub-programs G and H with input controls $G0$, $H0$ and exits $L11$, $L19$ have now been combined inside the concurrent fork-join block of our original example program T from Fig. 1a. As discussed in the previous sections G and H are specified by their scheduling types

$$G = \begin{pmatrix} 5 & -\infty \\ -\infty & -\infty \end{pmatrix} : (G0 \vee \mathit{active}(G)) \supset \circ L11 \oplus \circ \mathit{wait} \tag{16}$$

$$H = \begin{pmatrix} 5 & 7 \\ 4 & 6 \end{pmatrix} : (H0 \vee \mathit{active}(H)) \supset \circ L19 \oplus \circ \mathit{wait}. \tag{17}$$

How is multi-threading of G and H expressed? The logical conjunction $G \wedge H$ is not appropriate because it would say that both threads run in parallel while instead G and H are supposed to be scheduled in an interleaved fashion. For instance, the conjunction $(d_1, d_2) : \circ L11 \wedge \circ L19$ of two threads producing an exit signal $L11$ and $L19$ in d_1 and d_2 number of ics, respectively, would imply that $\mathit{max}(d_1, d_2) : \circ(L11 \wedge L19)$ which is correct for concurrent multi-processing while for multi-threading it should be the sum $d_1 + d_2 : \circ(L11 \wedge L19)$ instead. To capture the fact that G and H share the same processor and thus are interleaved, a new operator $G \parallel H$ on scheduling types is introduced to represent the concurrent configuration in Fig. 9. The definition of $T = G \parallel H$ is developed in Sec. B. In the following, let us sum up its basic properties in application to G and H.

Firstly, we need to extend the interfaces G and H somewhat to instrument the synchronization between them. E.g., G is strengthened to $G' =_{df} G \wedge ((L11 \wedge \neg L19) \supset \mathit{wait})$ to express that thread G runs into a wait state whenever it reaches its exit control $L11$[6] and H is not already at its exit $L19$ ($\neg L19$). Symmetrically, H is adapted to $H' =_{df} H \wedge ((L19 \wedge \neg L11) \supset \mathit{wait})$. These conjunctive additions do not change the WCRT interfaces (16) and (17). They merely strengthen the behavior with regard to the exit controls $L11$, $L19$, wait rather than the causality between entry controls $G0$, $H0$, $\mathit{active}(G)$, $\mathit{active}(H)$ and these exits.

Secondly, for the source behavior of composite T we introduce a new activation control $\mathit{active}(T)$ which starts the threads G and H according to $\mathit{active}(T) \supset (\mathit{active}(G) \wedge \mathit{active}(H)) \oplus (\mathit{active}(G) \wedge \neg \mathit{active}(H)) \oplus (\neg \mathit{active}(G) \wedge \mathit{active}(H))$. This is because source paths of T can arise from any non-empty subset of source paths from the sub-threads G and H. In terms of scheduling types this is presented systematically as

$$(0, 0, 0, -\infty)^T : \mathit{active}(T) \supset \mathit{active}(\circ\{G, H\}) \tag{18}$$

with the abbreviation $\mathit{active}(\circ\{G, H\}) =_{df} \circ(\mathit{active}(G) \wedge \mathit{active}(H)) \oplus \circ(\mathit{active}(G) \wedge \neg \mathit{active}(H)) \oplus \circ(\neg \mathit{active}(G) \wedge \mathit{active}(H)) \oplus \circ(\neg \mathit{active}(G) \wedge \neg \mathit{active}(H))$. The entry $-\infty$ in (18) says that the case is excluded where T is active but $\neg \mathit{active}(G) \oplus \neg \mathit{active}(H)$ holds, i.e., no thread is active.

Thirdly, when running the depth executions of T from delay nodes inside G and H we always assume that those threads which are not part of this depth execution have already terminated instantaneously in the previous instant and thus wait at their exits. Thus, under the global control $\mathit{active}(T)$ we strengthen the depth interfaces of G and H to wait at their exits whenever they are inactive. We put $G'' =_{df} G' \wedge (\neg \mathit{active}(G) \supset (L11 \wedge \mathit{wait}))$ and $H'' =_{df} H' \wedge (\neg \mathit{active}(H) \supset (L19 \wedge \mathit{wait}))$. This has an effect on the depth interfaces of G and H. We now get the immediate reaction $L11$ in case of $\neg \mathit{active}(G)$ and $L19$ if $\neg \mathit{active}(H)$. This can be accounted for in new

[6] Note that label L11 and H0 point to the same program instruction, viz. the first instruction of H. Since they encode *different* ways of reaching the same state of the program counter they cannot be identified. In particular, we can have H0 active in an execution in which H has been started without L11 being true, viz. as long as G is not yet finished.

interfaces

$$G'' = \begin{pmatrix} 5 & -\infty & 0 \\ -\infty & -\infty & -\infty \end{pmatrix} : (G0 \vee \mathit{active}(G) \vee \neg\mathit{active}(G)) \supset \circ L11 \oplus \circ \mathit{wait}$$

$$H'' = \begin{pmatrix} 5 & 7 & 0 \\ 4 & 6 & -\infty \end{pmatrix} \qquad : (H0 \vee \mathit{active}(H) \vee \neg\mathit{active}(H)) \supset \circ L19 \oplus \circ \mathit{wait}$$

with the added third column vector $(0, -\infty)^T$ of type $\neg\mathit{active}(G) \supset \circ L11 \oplus \circ \mathit{wait}$ and $\neg\mathit{active}(H) \supset \circ L19 \oplus \circ \mathit{wait}$, respectively.

The composition of $G'' \parallel H''$ now generates all possible interleavings of executions specified by type G'' with those from type H''. On the input side there are 9 possible ways of combining the three input controls $G0 \vee \mathit{active}(G) \vee \neg\mathit{active}(G)$ of G with those of $H0 \vee \mathit{active}(H) \vee \neg\mathit{active}(H)$ from H. However, we are only interested in the surface paths activated by $G0 \wedge H0$ and the depth paths activated by the combinations contained in $\mathit{active}(\circ\{G, H\})$. Instead of computing $G'' \parallel H''$, which gives too many paths, we calculate the surface interface $T_{srf} = G''_{srf} \parallel H''_{srf}$ and the depth interface $T_{dpt} = G''_{dpt} \parallel H''_{dpt}$ separately where

$$G''_{srf} = \begin{pmatrix} 5 \\ -\infty \end{pmatrix} : G0 \supset \circ L11 \oplus \circ \mathit{wait} \qquad\qquad H''_{srf} = \begin{pmatrix} 5 \\ 4 \end{pmatrix} : H0 \supset \circ L19 \oplus \circ \mathit{wait}. \tag{19}$$

Notice how the entry $-\infty$ in G''_{srf} expresses that program G does not have a sink path and thus must pass through L11 instantaneously.

$$G''_{dpt} = \begin{pmatrix} -\infty & 0 \\ -\infty & -\infty \end{pmatrix} : (\mathit{active}(G) \vee \neg\mathit{active}(G)) \supset \circ L11 \oplus \circ \mathit{wait} \tag{20}$$

$$H''_{dpt} = \begin{pmatrix} 7 & 0 \\ 6 & -\infty \end{pmatrix} : (\mathit{active}(H) \vee \neg\mathit{active}(H)) \supset \circ L19 \oplus \circ \mathit{wait}. \tag{21}$$

Again the first column $(-\infty, -\infty)^T$ in G''_{dpt} records that G possesses neither source nor internal paths. It turns out that the interleaving $G''_{srf} \parallel H''_{srf}$ is the *Kronecker product* $\otimes$ of the interfaces G''_{srf} and H''_{srf}:

$$\begin{pmatrix} 5 \\ -\infty \end{pmatrix} \otimes \begin{pmatrix} 5 \\ 4 \end{pmatrix} = \begin{pmatrix} (5)\,(5\ 4)^T \\ (-\infty)\,(5\ 4)^T \end{pmatrix} = \begin{pmatrix} 10 & 9 & -\infty & -\infty \end{pmatrix}^T \tag{22}$$

of type

$$(G0 \wedge H0) \supset (\circ(L11 \wedge L19) \oplus \circ(L11 \wedge \mathit{wait}) \oplus \circ(\mathit{wait} \wedge L19) \oplus \circ(\mathit{wait} \wedge \mathit{wait})).$$

This gives separate WCRT for all paths started in $G0 \wedge H0$ in which both sub-systems have through paths ($L11 \wedge L19$), one has a through path but the other pauses ($L11 \wedge \mathit{wait}$, $\mathit{wait} \wedge L19$) and those in which both threads end up in a pause ($\mathit{wait} \wedge \mathit{wait}$). In the last three cases the composite system pauses. They can be merged by the inclusion

$$\begin{pmatrix} 0 & -\infty & -\infty & -\infty \\ -\infty & 0 & 0 & 0 \end{pmatrix} : ((L11 \wedge L19) \vee (L11 \wedge \mathit{wait}) \vee (\mathit{wait} \wedge L19) \vee (\mathit{wait} \wedge \mathit{wait})) \\ \supset \circ(L11 \wedge L19) \oplus \circ \mathit{wait}. \tag{23}$$

Composing this exit-abstraction (23) with (22) gives the surface interface of T:

$$T_{srf} = \begin{pmatrix} 0 & -\infty & -\infty & -\infty \\ -\infty & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 10 & 9 & -\infty & -\infty \end{pmatrix}^T = \begin{pmatrix} 10 \\ 9 \end{pmatrix} : (G0 \wedge H0) \supset \circ(L11 \wedge L19) \oplus \circ \mathit{wait}.$$

For the depth interface of T we apply the same strategy: First we compose the two depth interfaces (20) and (21), $G''_{dpt} \parallel H''_{dpt}$ forming the Kronecker product of their timing matrices

$$\begin{pmatrix} -\infty & 0 \\ -\infty & -\infty \end{pmatrix} \otimes \begin{pmatrix} 7 & 0 \\ 6 & -\infty \end{pmatrix} = \begin{pmatrix} (-\infty) \cdot \begin{pmatrix} 7 & 0 \\ 6 & -\infty \end{pmatrix} & (0) \cdot \begin{pmatrix} 7 & 0 \\ 6 & -\infty \end{pmatrix} \\ (-\infty) \cdot \begin{pmatrix} 7 & 0 \\ 6 & -\infty \end{pmatrix} & (-\infty) \cdot \begin{pmatrix} 7 & 0 \\ 6 & -\infty \end{pmatrix} \end{pmatrix} = \begin{pmatrix} -\infty & -\infty & 7 & 0 \\ -\infty & -\infty & 6 & -\infty \\ -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty \end{pmatrix}$$

of type

$$\mathit{active}(\{G, H\}) \supset (\circ(L11 \wedge L19) \oplus \circ(L11 \wedge \mathit{wait}) \oplus \circ(\mathit{wait} \wedge L19) \oplus \circ(\mathit{wait} \wedge \mathit{wait})).$$

subject to the abbreviation $\mathit{active}(\{G, H\}) =_{df} (\mathit{active}(G) \wedge \mathit{active}(H)) \vee (\mathit{active}(G) \wedge \neg\mathit{active}(H)) \vee (\neg\mathit{active}(G) \wedge \mathit{active}(H)) \vee (\neg\mathit{active}(G) \wedge \neg\mathit{active}(H))$. Finally, the input and output controls are adjusted again: on the input side we connect with $\mathit{active}(T)$ using (18) which drops the irrelevant case $\neg\mathit{active}(G) \wedge \neg\mathit{active}(H)$ and on the output side we compress according to (23) which finally gives

$$\begin{aligned} T_{dpt} &= \begin{pmatrix} 0 & -\infty & -\infty & -\infty \\ -\infty & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} -\infty & -\infty & 7 & 0 \\ -\infty & -\infty & 6 & -\infty \\ -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ -\infty \end{pmatrix} \\ &= \begin{pmatrix} 0 & -\infty & -\infty & -\infty \\ -\infty & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 7 \\ 6 \\ -\infty \\ -\infty \end{pmatrix} = \begin{pmatrix} 7 \\ 6 \end{pmatrix} \ : \ \mathit{active}(T) \supset \circ(L11 \wedge L19) \oplus \circ\mathit{wait}. \end{aligned}$$

This is precisely what we should expect: Since block G is transient the depth interface of $T = G \parallel H$ is entirely determined by that of H.

To sum up, we have established the WCRT interface of $T = G \parallel H$ from Fig. 9 as

$$T = [T_{srf}, T_{dpt}] \ = \ \begin{pmatrix} 10 & 7 \\ 9 & 6 \end{pmatrix} \ : \ ((G0 \wedge H0) \vee \mathit{active}(T)) \supset \circ(L11 \wedge L19) \oplus \circ\mathit{wait}.$$

This models the concurrent composition of G and H but not yet the interface of the composite box T with fork and join. These are specified as

$$\begin{array}{lll} \mathsf{join} & \begin{pmatrix} 0 & -\infty \\ -\infty & 0 \end{pmatrix} & : ((L11 \wedge L19) \vee \mathit{wait}) \supset (\circ L20 \oplus \circ\mathit{wait}) \\ \mathsf{fork} & \begin{pmatrix} 4 & -\infty \\ -\infty & 1 \end{pmatrix} & : (T0 \vee \mathit{active}) \supset (\circ(G0 \wedge H0) \oplus \circ\mathit{active}). \end{array}$$

The entry $(4) : T0 \supset \circ(G0 \wedge H0)$ of fork includes the ics for two PAR, one PARE and one JOIN. Since the JOIN is always executed when at least one thread is active, its execution time is added to the fork, not the join. join itself is the identity matrix. Adding fork and join on the input and output side as depicted in Fig. 9 obtains

$$T = \begin{pmatrix} 0 & -\infty \\ -\infty & 0 \end{pmatrix} \cdot \begin{pmatrix} 10 & 7 \\ 9 & 6 \end{pmatrix} \cdot \begin{pmatrix} 4 & -\infty \\ -\infty & 1 \end{pmatrix} = \begin{pmatrix} 14 & 8 \\ 13 & 7 \end{pmatrix} \ : \ (T0 \vee \mathit{active}) \supset (\circ L20 \oplus \circ\mathit{wait})$$

as the WCRT for the composite program T. The longest through path is exemplified by the sequence of nodes $v_0(3)+\{v_1+v_2+v_3+v_4+v_7\}_G(5)+\{v_8+v_9+v_{12}+v_{14}+v_{15}\}_H(5)+v_{16}(1) = 14$. A longest sink path is $v_0(3)+\{v_1+v_2+v_3+v_4+v_7\}_G(5)+\{v_8+v_9+v_{12}+v_{13}\}_H(4)+v_{16}(1) = 13$. As a maximal source path we could take $\{\}_G(0)+\{v_9+v_{10}+v_{11}+v_9+v_{12}+v_{14}+v_{15}\}_H(7)+v_{16}(1) = 8$ and finally a possible longest internal path $\{\}_G(0)+\{v_9+v_{10}+v_{11}+v_9+v_{12}+v_{13}\}_H(6)+v_{16}(1) = 7$.

In a practical WCRT algorithm such as the one of [13] many of the matrix multiplications shown above are executed implicitly and more efficiently in the combinatorics of traversing the program's control flow graph forming maximum and additions as we go along. This is possible only so far as control flow dependencies are represented explicitly in the graph. In general, with data-dependencies, this may be an exponential problem so that symbolic techniques are needed. The point of the above is to highlight the essential algebraic nature of WCRT analysis and the role of timing interfaces in keeping track of the semantic meaning of WCRT data for modular analysis.

6 Conclusion

We introduced an interface algebra for compositional analysis of WCRT in synchronous multi-threading. We have applied these interfaces to compute the WCRT for programs running on the Kiel Esterel Processor. The 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. Even though it is still under development, the interface algebra results in tighter WCRT analysis for the KEP compared to existing algorithms. For the small example in Section 4 we computed a WCRT of 14, while the algorithm described in [13] computes an over-approximation of 16. This is due to the better handling of parallel composition. Since the interfaces are compositional, we should also be able to get a better performance for the WCRT computation itself. 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.

Encouraged by the manual computations, we want to implement a WCRT algorithm based on the new interfaces. There are still some constructs of Esterel that cannot be directly expressed in the interface algebra so far. These are in particular exit-traps, which are a third form of abortion to express exceptions. Furthermore, in this paper we consider all abortions as immediate, which might lead to more and longer paths than actually executable. 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. So far, 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 could also be captured by the interfaces.

So far, we use the interfaces to compute WCRTs for Esterel programs on the KEP. But this approach could as well be used for other execution platforms, which implement a similar form of synchronous multi-threading or multi-processing.

References

1. Edwards, S., Lee, E.A.: The case for the Precision Timed (PRET) machine. Technical report no. ucb/eecs-2006-149, EECS Department, University of California, Berkeley (November 2006) `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-149.pdf`.
2. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Guernic, P.L., de Simone, R.: The Synchronous Languages Twelve Years Later. In: Proceedings of the IEEE, Special Issue on Embedded Systems. Volume 91. (January 2003) 64–83
3. Berry, G., Cosserat, L.: The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In: Seminar on Concurrency, Carnegie-Mellon University. Volume 197 of Lecture Notes in Computer Science (LNCS)., Springer-Verlag (1984) 389–448
4. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8**(3) (June 1987) 231–274
5. Lee, E.A.: The problem with threads. IEEE Computer **39**(5) (2006) 33–42
6. Potop-Butucaru, D., Edwards, S.A., Berry, G.: Compiling Esterel. Springer (May 2007)
7. Berg, C., Engblom, J., Wilhelm, R.: Requirements for and design of a processor with predictable timing. In Thiele, L., Wilhelm, R., eds.: Perspectives Workshop: Design of Systems with Predictable Behaviour. Number 03471 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2004) `http://drops.dagstuhl.de/opus/volltexte/2004/5`.
8. von Hanxleden, R., Li, X., Roop, P., Salcic, Z., Yoong, L.H.: Reactive processing for reactive systems. ERCIM News **66** (October 2006) 28–29 `http://ercim-news.ercim.org/content/view/51/82/`.
9. Li, X., von Hanxleden, R.: The Kiel Esterel Processor - a semi-custom, configurable reactive processor. In Edwards, S.A., Halbwachs, N., v. Hanxleden, R., Stauner, T., eds.: Synchronous Programming - SYNCHRON'04. Number 04491 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany (2005) `http://drops.dagstuhl.de/opus/volltexte/2005/159`.
10. Yoong, L.H., Roop, P., Salcic, Z., Gruian, F.: Compiling Esterel for distributed execution. In: International Workshop on Synchronous Languages, Applications, and Programming (SLAP'06), Vienna, Austria (March 2006)

11. Li, X., Boldt, M., von Hanxleden, R.: Mapping Esterel onto a multi-threaded embedded processor. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06), San Jose, CA (October 21–25 2006)
12. Yuan, S., Andalam, S., Yoong, L.H., Roop, P.S., Salcic, Z.: Starpro—a new multithreaded direct execution platform for Esterel. In: Proceedings of Model Driven high-Level Programming of Embedded Systems (SLA++P), Workshop at ETAPS '08, Budapest, Hungary (April 2008)
13. Boldt, M., Traulsen, C., von Hanxleden, R.: Worst case reaction time analysis of concurrent reactive programs. Electronic Notes in Theoretical Computer Science **203**(4) (June 2008) 65–79 Proceedings of the International Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P 2007), March 2007, Braga, Portugal.
14. Li, X., Lukoschus, J., Boldt, M., Harder, M., von Hanxleden, R.: An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis. In: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), New York, NY, USA, ACM Press (September 2005) 225–236
15. Baccelli, F.L., Cohen, G., Olsder, G.J., Quadrat, J.P.: Synchronisation and Linearity. John Wiley & Sons (1992)
16. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The determination of worst-case execution times—overview of the methods and survey of tools. ACM Transactions on Embedded Computing Systems (TECS) **7**(3) (2008)
17. Logothetis, G., Schneider, K.: Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In: Design, Automation and Test in Europe (DATE), Munich, Germany, IEEE Computer Society (March 2003) 196–203
18. Logothetis, G., Schneider, K., Metzler, C.: Exact low-level runtime analysis of synchronous programs for formal verification of real-time systems. In: Forum on Design Languages (FDL), Frankfurt, Germany, Kluwer (2003)
19. André, C., Boulanger, F., Péraldi, M.A., Rigault, J.P., Vidal-Naquet, G.: Objects and synchronous programming. European Journal on Automated Systems **31**(3) (1997) 417–432
20. Hainque, O., Pautet, L., Biannic, Y.L., Nassor, E.: Cronos: A separate compilation toolset for modular Esterel applications. In Wing, J.M., Woodcock, J., Davies, J., eds.: World Congress on Formal Methods. Volume 1709 of Lecture Notes in Computer Science., Springer (September 1999) 1836–1853
21. Lee, E.A., Zheng, H., Zhou, Y.: Causality interfaces and compositional causality analysis. In: Foundations of Interface Technologies (FIT'05). ENTCS, Elsevier (2005)
22. Wandeler, E., Thiele, L.: Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In: Proceedings of the ACM International Conference on Embedded Software (EMSOFT'05). (September 2005)
23. Henzinger, T., Matic, S.: An interface algebra for real-time components. In: Proceedings of the 12th Annual Real-Time and Embedded Technology and Applications Symposium (RTAS), Los Alamitos, CA, USA, IEEE Computer Society (2006) 253–266
24. Boudec, J.L., Thiran, P.: Network Calculus - A theory of deterministic queuing systems for the internet. Volume 2050 of Lecture Notes in Computer Science. Springer (2001)
25. de Alfaro, L., Henzinger, T.: Interface automata. In: Proc. Foundations of Software Engineering, ACM Press (2001) 109–120
26. Lee, E.A., Xiong, Y.: System-level types for component-based design. In: Workshop on Embedded Software EMSOFT 2001, Lake Tahoe, CA, USA (October 2001)
27. Lee, E.A., Xiong, Y.: A behavioral type system and its application in Ptolemy II. Formal Aspects of Computing **13**(3) (August 2004) 210–237
28. Mendler, M.: Characterising combinational timing analyses in intuitionistic modal logic. The Logic Journal of the IGPL **8**(6) (November 2000) 821–853
29. Lüttgen, G., Mendler, M.: Axiomatizing an algebra of step reactions for synchronous languages. In Brim, L., Jančar, P., Ketínský, M., Kučera, A., eds.: International Conference on Concurrency Theory (CONCUR'02). Number 2421 in Lecture Notes in Computer Science, Brno, Springer (August 2002) 386–401
30. Lüttgen, G., Mendler, M.: Towards a model-theory for Esterel. In Maraninchi, F., Girault, A., Rutten, E., eds.: Synchronous Languages, Application, and Programming (SLAP '02). Volume 65,5 of ENTCS., Elsevier Science (2002)

A WCRT Interface Types

Executions and schedules are typed using logical expressions of the form $f{:}\phi$ consisting of an interface formula ϕ (scheduling type) together with a timing function f (scheduling bound). The canonical interfaces used in this paper (introduced in Sec. 5.1) are of the form

$$(\zeta_1 \vee \zeta_2 \cdots \vee \zeta_m) \supset (\circ\xi_1 \oplus \circ\xi_2 \oplus \cdots \oplus \circ\xi_n) \tag{24}$$

with basic controls ζ_i and ξ_j. For these types the timing functions are essentially delay matrices of shape $n \times m$. These interfaces are particular structures in a more general theory that we shall briefly outline now. First, recall our language of types

$$\phi ::= A \mid \mathit{true} \mid \mathit{false} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \oplus \phi \mid \phi \supset \phi \mid \phi \parallel \phi \mid \circ\phi,$$

where $A \in \mathbb{S}$ ranges over a set of control signals. General WCRT theory associates with every scheduling type ϕ a set of *scheduling bounds* or *reaction bounds* $\mathsf{Bnd}(\phi)$ uniformly as follows:

$$\begin{aligned}
\mathsf{Bnd}(\mathit{false}) &= \underline{1} & \mathsf{Bnd}(\mathit{true}) &= \underline{1}\\
\mathsf{Bnd}(A) &= \underline{1} & \mathsf{Bnd}(\neg\phi) &= \underline{1}\\
\mathsf{Bnd}(\phi \wedge \psi) &= \mathsf{Bnd}(\phi) \times \mathsf{Bnd}(\psi) & \mathsf{Bnd}(\phi \vee \psi) &= \mathsf{Bnd}(\phi) + \mathsf{Bnd}(\psi)\\
\mathsf{Bnd}(\phi \oplus \psi) &= \mathsf{Bnd}(\phi) \times \mathsf{Bnd}(\psi) & \mathsf{Bnd}(\phi \supset \psi) &= \mathsf{Bnd}(\phi) \rightarrow \mathsf{Bnd}(\psi)\\
\mathsf{Bnd}(\circ\phi) &= (\mathbb{N} \cup \{-\infty\}) \times \mathsf{Bnd}(\phi) & \mathsf{Bnd}(\phi \parallel \psi) &= \mathsf{Bnd}(\phi) \times \mathsf{Bnd}(\psi),
\end{aligned}$$

where $\underline{1} = \{0\}$ is a distinguished singleton set. More generally, we use the notation $\underline{n}$ for $n \in \mathbb{N}$ to denote the set $\{0, 1, \ldots, n-1\}$, discretely ordered. Elements of the disjoint sum $\mathsf{Bnd}(\phi) + \mathsf{Bnd}(\psi)$ are presented as pairs $(0, f)$ where $f \in \mathsf{Bnd}(\phi)$ or $(1, g)$ where $g \in \mathsf{Bnd}(\psi)$. An element $f \in \mathsf{Bnd}(\phi)$ is a form of generalized higher-order timing matrix for schedules of shape ϕ.

Let $\sigma = \emptyset \subseteq \sigma(0) \subseteq \sigma(1) \subseteq \sigma(2) \subseteq \cdots \subseteq \sigma(n-1) \subseteq \mathbb{S}$ be an execution. We define a *sub-execution* $\sigma' \subseteq \sigma$ to be a sub-sequence $\sigma' = \emptyset \subseteq \sigma'(0) \subseteq \sigma'(1) \subseteq \sigma'(2) \subseteq \cdots \subseteq \sigma'(m-1) \subseteq \mathbb{S}$ consisting of an arbitrary number of events $\sigma'(i) = \sigma(f_i)$ of σ with a monotonic function $f{:}\underline{m} \rightarrow \underline{n}$. Such a sub-sequence models a sub-thread of σ which observes only a certain subset of events from σ according to when it is scheduled. We write $\sigma = \sigma_1 + \sigma_2$ to express that execution σ can be partitioned into sub-executions $\sigma_1, \sigma_2 \subseteq \sigma$ such that each computation step $(\sigma(i), \sigma(i+1))$ in σ is contained in σ_1 or in σ_2. As a degenerated case the empty execution can always be split $\emptyset = \emptyset + \emptyset$. Finally, for every $d \in \mathbb{N} \cup \{-\infty\}$ we define the *delayed* execution $\sigma[d,:]$ to be the sequence $\sigma[d,:] = \sigma(d) \subseteq \sigma(d+1) \subseteq \sigma(d+2) \subseteq \cdots \subseteq \sigma(n-1)$. If $d \geq |\sigma|$ then $\sigma[d,:] = \emptyset$ is the empty execution. If $d = -\infty$ then $\sigma[d,:] = \emptyset\sigma$, i.e., an initial empty event is added to σ.

We say that an execution σ *validates* a scheduling type ϕ with bound $f \in \mathsf{Bnd}(\phi)$ written $\sigma \models f : \phi$, according to the rules

$$\begin{array}{ll}
\sigma \models 0 : \mathit{false} & \mathit{iff}\ |\sigma| = 0\\
\sigma \models 0 : \mathit{true} & \mathit{iff}\ \text{always}\\
\sigma \models 0 : A & \mathit{iff}\ \forall 0 \leq j < |\sigma| \Rightarrow A \in \sigma(j)\\
\sigma \models (f, g) : \phi \wedge \psi & \mathit{iff}\ \sigma \models f : \phi \text{ and } \sigma \models g : \psi\\
\sigma \models (0, f) : \phi \vee \psi & \mathit{iff}\ \sigma \models f : \phi\\
\sigma \models (1, g) : \phi \vee \psi & \mathit{iff}\ \sigma \models g : \psi\\
\sigma \models (f, g) : \phi \oplus \psi & \mathit{iff}\ \sigma \models f : \phi \text{ or } \sigma \models g : \psi\\
\sigma \models f : \phi \supset \psi & \mathit{iff}\ \forall \sigma' \subseteq \sigma.\ \forall g \in \mathsf{Bnd}(\phi).\ (\sigma' \models g : \phi \Rightarrow \sigma' \models f\,g : \psi)\\
\sigma \models (t, f) : \circ\phi & \mathit{iff}\ \sigma[t,:] \models f : \phi\\
\sigma \models (f, g) : \phi_1 \parallel \phi_2 & \mathit{iff}\ \exists \sigma_1, \sigma_2 \subseteq \sigma.\ \sigma = \sigma_1 + \sigma_2 \text{ and } \sigma_1 \models f : \phi_1 \text{ and } \sigma_2 \models g : \phi_2.
\end{array}$$

One shows by induction on type ϕ that the empty execution validates all types. Similarly, for all delay times d after the end of an execution, $d \geq |\sigma|$, we have $\sigma[d,:] \models f : \phi$ for *all* types $f : \phi$, including $0 : \mathit{false}$. Validity is monotonic in the sense that as the delay time increases more and more types may become valid. Formally, if $\sigma[d,:] \models f : \phi$ then also $\sigma[d',:] \models f : \phi$ for all $d' \geq d$. More generally, truth is inherited by sub-schedules, i.e., if $\sigma \models f : \phi$ and $\sigma' \subseteq \sigma$ then $\sigma' \models f : \phi$, too. The inclusion $\sigma[d',:] \subseteq \sigma[d,:]$ for $d' \geq d$ is a special case of this. We will sometimes write $\sigma, d \models f : \phi$ for $\sigma[d,:] \models f : \phi$.

Notice how $\|$ captures multi-threading. An execution σ validates $\phi_1 \parallel \phi_2$ if it can be divided into two sub-executions σ_1, σ_2 each of which validates its part of the composition. In general σ_i overlaps with parts of the other execution σ_j $(i \neq j)$. All interleaved activities inside σ_2 overlapping with σ_1 are happening concurrently from σ_1's local point of view and can involve the simultaneous occurrence of control signals. So, although at the outermost level of a master-thread σ each step $(\sigma(i), \sigma(i+1))$ involves only a single control signal, a sub-thread $\sigma_1 \subseteq \sigma$ may well experience the occurrence of several signals in the same local instruction cycle. This is why in multi-threading (and in multi-processing for that matter) we need to consider arbitrary executions sequences σ.

A set of executions S defines a *schedule*. Thus, our semantics associates with every pair $f : \phi$ a schedule $[\![f : \phi]\!] = \{\, \sigma \mid \sigma \models f : \phi \,\}$. It is useful to view $f : \phi$ as a specification of executions which combines a qualitative aspect ϕ with the quantitative aspect $f \in \mathsf{Bnd}(\phi)$. The scheduling type ϕ captures the abstract causal relationships between the control points and the scheduling bound f refines this chronometrically by giving concrete numeric distances and durations. The colon as a binary connective separates these concerns. Specifications may be compared naturally in terms of their executions. We write $f : \phi \cong g : \psi$ if $[\![f : \phi]\!] = [\![g : \psi]\!]$ and $f : \phi \preceq g : \psi$ if $[\![f : \phi]\!] \subseteq [\![g : \psi]\!]$.

Example 2. For instance, the pair $(5, 0) : \circ \mathit{false}$ specifies all executions which have at most 5 events and thus consume at most 4 cycles. The type *false* represents abstract termination and $(5, 0)$ measures the duration until it occurs. Note that the second component $0 \in \mathsf{Bnd}(\mathit{false}) = \underline{1}$ in the pair $(5, 0)$ is needed for systematic reasons to deal with the general case, uniformly. It does not have any semantic meaning, however, and can be dropped systematically if needed. So, we would write $5 : \circ\mathit{false}$ instead.

The pair $\lambda x.\,(12, 0) : A \supset \circ B$ comprises all executions in which control point B is necessarily passed after at most 11 scheduling steps after any occurrence of A. Here, the type $A \supset \circ B$ expresses a causal control flow from A to B and the bound $\lambda x.\,(12, 0)$ measures the distance. Again, the arguments $x \in \mathsf{Bnd}(A) = \underline{1}$ and value $0 \in \mathsf{Bnd}(B) = \underline{1}$ are irrelevant in this function, whence we will write $(12) : A \supset \circ B$ more compactly.

In a concrete WCRT analysis problem we are given some schedule S and a scheduling type ϕ (as its specification) and ask for a stabilization bound f such that $S \subseteq [\![f : \phi]\!]$. If such a bound exists we say that S is *well timed* for ϕ with bound f, and write $S \models f : \phi$. If we are not interested in the bound itself, we write $S \models \phi$ to express that S is well-timed for ϕ, i.e., there exists $f \in \mathsf{Bnd}(\phi)$ such that $S \subseteq [\![f : \phi]\!]$. In this way each scheduling type ϕ defines a class of well-timed schedules $[\![\phi]\!] =_{\mathit{df}} \{S \mid S \models \phi\}$ while a pair $f : \phi$ specifies individual executions. Because of this it is possible to use expressions $f : \phi$ themselves as generalized "control signals" in types such as $(f : \phi) \wedge \psi$ or $\circ(f : \phi)$. We simply define $\mathsf{Bnd}(f : \phi) =_{\mathit{df}} \underline{1}$ and $\sigma \models 0 : (f : \phi)$ iff $\sigma \models f : \phi$.

For any given schedule S there may be infinitely many bounds under which S is well-timed for a type. We will be interested in optimal bounds. To make this formal we introduce a partial ordering $\sqsubseteq$ on bounds, so that $f \sqsubseteq g$ means f is *tighter* than g. The ordering on $\mathsf{Bnd}(\phi)$ is generated by induction on type ϕ from the natural ordering $\leq$ on $\mathbb{N}$, taking point-wise ordering on products $\mathsf{Bnd}(\phi) \times \mathsf{Bnd}(\psi)$ and function spaces $\mathsf{Bnd}(\phi) \to \mathsf{Bnd}(\psi)$. For disjoint unions $\mathsf{Bnd}(\phi) + \mathsf{Bnd}(\psi)$ we take the discrete ordering, so that $(i, f) \sqsubseteq (j, g)$ *iff* $i = j$ and $f \sqsubseteq g$. Then, a scheduling bound $f \in \mathsf{Bnd}(\phi)$ is *exact* or *worst-case* for S and ϕ, if for all $g \in \mathsf{Bnd}(\phi)$ such that $g \sqsubseteq f$ we have $f = g$ *iff* $S \models g : \phi$.

Example 3. Bounded termination can be specified by the scheduling type $\circ\mathit{false}$ because for any execution σ, the statement $\sigma \models n : \circ\mathit{false}$ says that σ is finite and has length at most n, i.e., $|\sigma| \leq n$. Now suppose schedule S_G is the set of all possible executions of a program G. The WCRT of G then is the maximal number of steps in all executions $\mathsf{WCRT}(G) = \mathit{max}\{|\sigma| - 1 \mid \sigma \in S_G\}$. This is one less than the minimal upper bound on the length of all executions, i.e.,

$\mathsf{WCRT}(G) = \mathit{min}\{n-1 \mid \forall\sigma \in S_G.\ |\sigma| \leq n\}$ and thus the same as the optimal scheduling bound m such that $S_G \models m+1 : \circ \mathit{false}$. Specifically if $S_G = \emptyset$, then $\mathsf{WCRT}(G) = -\infty$.

If program G is not stand-alone but a program fragment then S_G is the set of all possible executions of G in arbitrary program contexts into which G is embedded. Let us suppose that G is started through control signal *active* and every thread entering G must leave G via a unique control point *end*. In that case $\mathsf{WCRT}(G)$ is the least upper bound on the number of cycles of any execution $\sigma \in S_G$ between the occurrence of *active* and *end*. In other words, we want the optimal bound f such that $S_G \models (f+1) : \mathit{active} \supset \circ \mathit{end}$. In this paper the unique termination point used is the global constant *false*. Abbreviating $\mathit{wait} =_{\mathit{df}} 1 : \mathit{false}$ we may write $S_G \models (f) : \mathit{active} \supset \circ \mathit{wait}$.

We have the following natural monotonicity property which says that if we relax the scheduling bound or reduce the set of schedules we are not losing well-timedness:

Proposition 1. *Let R, S be schedules and $f, g \in \mathit{Bnd}(\phi)$ scheduling bounds for type ϕ such that $R \subseteq S$ and $f \sqsubseteq g$. Then, $S \models f : \phi$ implies $R \models g : \phi$.*

On the other hand if we enlarge the schedule (i.e., include more executions) then the worst-case scheduling bound becomes larger in general and similarly if we tighten the bounds we get fewer executions satisfying the bound. A bound $f \in \mathsf{Bnd}(\phi)$ is a *uniform* bound for ϕ if $S \models f : \phi$ for any schedule S, written $\models f : \phi$.

The partial ordering $(\mathsf{Bnd}(\phi), \sqsubseteq)$ depends on the structure of a scheduling type ϕ and measures the amount of WCRT information that is associated with ϕ. In this respect the most simple class of types is that for which $\mathsf{Bnd}(\phi)$ is (order) isomorphic to $\underline{1}$. This happens precisely if $|\mathsf{Bnd}(\phi)| = 1$. Such types are called *units* since they only carry trivial timing information. An example of unit propositions are the double negated formulas. Such statements may have uniform bounds, but they do not contain any information. More precisely, it can be shown that if $f \in \mathsf{Bnd}(\neg\neg\phi)$ and $\sigma \models f : \neg\neg\phi$ then for all $g \in \mathsf{Bnd}(\neg\neg\phi)$, $\sigma \models g : \neg\neg\phi$. Thus, $\neg\neg\phi$ either has a uniform scheduling bound or it does not; and if it does all bounds are uniform bounds. Moreover, they cannot be distinguished from each other by the relation $\models$. Hence they might just as well all be identified. The unit types, which we denote by meta-variable ζ, are characterized syntactically as follows:

$$\zeta ::= \mathit{true} \mid \mathit{false} \mid A \mid \zeta \wedge \zeta \mid \zeta \oplus \zeta \mid \phi \supset \zeta,$$

where ϕ is an arbitrary type. The *basic controls* introduced in Sec. 5.1 are a particular class of unit types. It is natural to exploit the isomorphisms $\mathsf{Bnd}(\zeta) \cong \underline{1}$ and identify all bounds $f \in \mathsf{Bnd}(\zeta)$ canonically with the unique $0 \in \underline{1}$. In fact we may simply identify ζ with $0 : \zeta$ and write $d : \zeta$ instead of $(d, 0) : \circ\zeta$. Note that no non-empty execution satisfies $\sigma \models -\infty : A$ for any control signal A which means that timing delay $-\infty$ can be used for non-existing control dependencies.

Proposition 2. *The type operators $\wedge$, $\oplus$ and $\|$ correspond to the arithmetic operations min, max and $+$, respectively. More precisely, the values $\mathit{min}(d_1, d_2)$, $\mathit{max}(d_1, d_2)$ and $d_1 + d_2$ are the optimal scheduling bounds such that*

$$\begin{aligned}
& d_1 : X \wedge d_2 : Y \ \preceq\ \mathit{min}(d_1, d_2) : (X \oplus Y) \ \cong\ \mathit{min}(d_1, d_2) : (X \vee Y) \\
& d_1 : X \wedge d_2 : Y \ \preceq\ \mathit{max}(d_1, d_2) : (X \wedge Y) \\
& ((d_1 : X) \wedge (X \supset \mathit{wait})) \parallel ((d_2 : Y) \wedge (Y \supset \mathit{wait})) \ \preceq\ d_1 + d_2 : X \wedge Y \\
& (d_1 : X) \wedge (X \supset (d_2 : Y)) \ \preceq\ d_1 + d_2 : X \wedge Y \\
& d_1 : X \oplus d_2 : Y \ \preceq\ \mathit{max}(d_1, d_2) : X \oplus Y.
\end{aligned}$$

A somewhat larger but still rather special class of types are the formulas for which $\mathsf{Bnd}(\phi)$ is canonically order-isomorphic to a Cartesian product of numbers, i.e., to $\mathbb{N}^{\underline{n}}$ for some $n \geq 0$. Here, by canonically order-isomorphic we mean that $\mathsf{Bnd}(\phi) \cong \mathbb{N}^{\underline{n}}$ can be derived by the natural isomorphisms of partial orderings $\mathbb{N}^{\underline{0}} \cong \underline{1}$, $\mathbb{N}^{\underline{1}} \cong \mathbb{N}$, $\underline{1}^{\tau} \cong \underline{1}$, $\mathbb{N}^{\underline{k}} \times \mathbb{N}^{\underline{n}} \cong \mathbb{N}^{\underline{k+n}}$, $\underline{n} \times \underline{k} \cong \underline{nk}$, $\underline{n} + \underline{k} \cong \underline{n+k}$, $\underline{n} \to \underline{k} \cong \underline{k^n}$, $\underline{n} \to \mathbb{N}^{\underline{k}} \cong \mathbb{N}^{\underline{nk}}$ alone. This implies that, for instance, the well-known (dove-tailing) bijection between $\mathbb{N}$ and $\mathbb{N}^{\underline{2}}$ does not count as canonical. The scheduling types ϕ with $\mathsf{Bnd}(\phi) \cong \mathbb{N}^{\underline{n}}$ are called *elementary*. Elementary formulas are referred to by θ and generated by the grammar

$$\theta ::= \theta \wedge \theta \ \mid \ \circ\zeta \ \mid \ \phi \supset \theta,$$

where ζ is a unit and ϕ is $\circ$-free. Note that every unit type is elementary. Elementary scheduling types are of special interest since $\mathsf{Bnd}(\phi)$ is a lattice and its elements first-order objects, *i. e.* vectors of natural numbers. The following Proposition 3 provides the basis for WCRT analysis:

Proposition 3. *Let θ be an elementary type and S a schedule such that S is well-timed for θ. Then, the set $\{ f \mid S \models f : \theta \}$ ordered by $\sqsubseteq$ is (nonempty and) a complete lower semi-lattice.*

Prop. 3 implies the existence of unique worst-case stabilization bounds for elementary scheduling types. More precisely, it says that for every schedule S we have a worst case reaction bound relative to any scheduling type ϕ by putting $\mathsf{WCRT}(S, \phi) = \sqcap\{f \mid S \models f : \phi\}$. As a degenerate case we have $\mathsf{WCRT}(S, \phi) = -\infty$ if $S = \emptyset$.

The WCRT analyses suggested in this paper are implementations of scheduling bounds for elementary types θ of the form

$$\theta \equiv (\bigvee_{i \in I} \zeta_i) \supset \bigoplus_{k \in K} \circ\xi_k \qquad \zeta_i \equiv \bigoplus_{j \in J_i} x_{ij} \qquad \xi_k \equiv \bigoplus_{l \in L_k} y_{kl},$$

where x_{ij} and y_{kl} are min-terms of literals over control signals $\mathbb{S}$. The set of scheduling bounds for such θ are canonically order-isomorphic to $\mathbb{N}^{|K| \times |I|}$ which can be understood as $|K| \times |I|$ timing matrices relative to the base controls ζ_i and ξ_k. The logical conjunction of these interfaces in a fixed set of base controls corresponds to matrix multiplications in max-plus algebra $(\mathbb{N}, +, \mathit{max}, 0, -\infty)$. Furthermore, using logical reasoning on base controls ζ_i, ξ_j we can massage the semantics of timing matrices very much like we do with base transformations in ordinary linear algebra. This equally tight as uniform combination of timing algebra and logical reasoning is the key to expressing timing abstractions and distinguishes our WCRT algebra from previous work on component interfaces such as [21].

B Multi-threading Composition

This section explains how the operator $\|$ is used to express multi-threading and also fills in some technical details supporting the developments in Sec. 5.4.

In multi-threading we need to enforce termination of threads. There must be some means of expressing that a schedule does not lock up inside a thread which has reached a certain exit point but rather hands over to some other thread. In `fork-join` blocks this may be either a concurrent sibling thread or the parent thread. Stopping can in fact be specified by the special type $\mathit{wait} =_{\mathit{df}} 1 : \mathit{false}$ which is satisfied on singleton intervals, i.e., $\sigma \models \mathit{wait}$ iff $|\sigma| \leq 1$.[7]

Take the example in Fig. 10 (b): If T_1 reaches its exit control Y_1 then T_1 is finished and yields back to the scheduler, i.e. either to thread T_2 or the parent thread which means leaving the join through Y. This yielding of T_1 to T_2 is specified by the scheduling type $(Y_1 \wedge \neg Y_2) \supset \mathit{wait}$. Any execution σ_1 of T_1 satisfying $\sigma_1 \models (Y_1 \wedge \neg Y_2) \supset \mathit{wait}$ which reaches control point Y_1 in some cycle i must either include Y_2 in the very next cycle $i + 1$ or terminate, i.e., Y_1 occurs in the last control event of σ_1. Formally, $\sigma \models (Y_1 \wedge \neg Y_2) \supset \mathit{wait}$ iff $\forall i < |\sigma| - 1.\ Y_1 \in \sigma(i) \Rightarrow Y_2 \in \sigma(i + 1)$.

Now, consider the type

$$\phi_1 \parallel \phi_2 =_{\mathit{df}} \circ Y_1 \wedge ((Y_1 \wedge \neg Y_2) \supset \mathit{wait}) \ \parallel \ \circ Y_2 \wedge ((Y_2 \wedge \neg Y_1) \supset \mathit{wait}). \tag{25}$$

[7] Note the expression $(\neg X) \supset \mathit{wait}$ means that X has a down-time of at most 1 ic. The combination $(\neg X \vee X) \supset \mathit{wait}$ means that X oscillates with 1 ic period.

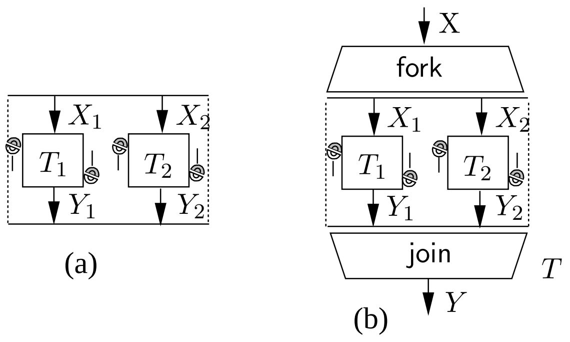


Fig. 10: Interleaved concurrent threads T_1, T_2 (a) and composite system T with both join and fork (b).

and assume $\sigma \models (d_1, d_2) : \phi_1 \parallel \phi_2$ and $d_1, d_2 \geq 1$. Then, by definition, there are sub-executions $\sigma_i \sqsubseteq \sigma$ with $\sigma = \sigma_1 + \sigma_2$ such that $\sigma_i \models d_i : \phi_i$ for $i, j \in \{1, 2\}$ and $i \neq j$. Let us look at σ_1 which is the sub-execution in which thread T_1 is executed, satisfying $\sigma_1 \models d_1 : \phi_1$. The other case $\sigma_2 \models d_2 : \phi_2$ is perfectly symmetric. First, $\sigma_1 \models d_1 : \circ Y_1$ says that after maximal delay d_1 (taken relative to σ_1, not σ) σ_1 will activate the exit control Y_1 of T_1. Second, $\sigma_1 \models 0 : ((Y_1 \wedge \neg Y_2) \supset \mathit{wait})$ makes sure that when Y_1 appears then either σ_1 terminates or Y_2 appears in the very next cycle. Thus, the execution σ passes through Y_2 or continues *outside* of σ_1 (in σ_2) no later than immediately in the next cycle after T_1 has reached Y_1. The same applies to the sub-execution $\sigma_2 \models d_2 : \phi_2$ with delay d_2. Since the total execution $\sigma = \sigma_1 + \sigma_2$ is fully covered by σ_1 and σ_2 we conclude that σ must reach the conjunction of Y_1 and Y_2 within $d_1 + d_2$ ics[8], i.e.,

$$\sigma \models d_1 + d_2 : \circ(Y_1 \wedge Y_2). \tag{26}$$

Let us see in more detail why this must be true. First, if σ is not longer than $d_1 + d_2$ then (26) holds trivially. Hence, suppose σ comprises at least $d_1 + d_2 + 1$ ics, i.e., $|\sigma| \geq d_1 + d_2 + 1$. It is easy to see that then at least one of the executions σ_i must contain at least $d_i + 1$ cycles. For otherwise, if both $|\sigma_1| \leq d_1$ and $|\sigma_2| \leq d_2$ then $|\sigma| \leq |\sigma_1| + |\sigma_2| - 1 \leq d_1 + d_2 - 1$ which is a contradiction. The subtraction of 1 here is due to the fact that σ_1 and σ_2 form a transition cover of σ, and the number of transitions of an execution is one less than its length.

Consider the case that $|\sigma_1| \geq d_1 + 1$. Since $\sigma_1 \models d_1 : \circ Y_1$ we know that after a maximal delay d_1 the schedule σ_1 will activate the exit control Y_1, i.e., $\sigma_1, i_1 \models 0 : Y_1$ for some $i_1 \leq d_1 \leq |\sigma_1| - 1$. Further, because of $\sigma_1 \models ((Y_1 \wedge \neg Y_2) \supset \mathit{wait})$, we either have $i_1 = |\sigma_1| - 1$ or $Y_2 \in \sigma_1(i_1 + 1)$. The latter implies that $\sigma_1, i_1 + 1 \models Y_1 \wedge Y_2$ and thus (26) because $i_1 + 1 \leq d_1 + 1 \leq d_1 + d_2$.

Thus it remains to look at the former case in which we conclude $|\sigma_1| = d_1 + 1$. Thus, schedule σ_2 must have length $|\sigma_2| \geq |\sigma| - |\sigma_1| + 1 = |\sigma| - d_1 \geq d_2 + 1$. By an argument analogous to above, $\sigma_2 \models d_2 : \circ Y_2$ and $\sigma_2 \models ((Y_2 \wedge \neg Y_1) \supset \mathit{wait})$ give us $Y_2 \in \sigma_2(j_2)$ for some $j_2 \leq d_2 \leq |\sigma_2| - 1$ such that either (i) $j_2 = d_2 = |\sigma_2| - 1$, or (ii) $Y_1 \in \sigma_2(j_2 + 1)$. Again, (ii) yields (26) immediately because then $\sigma_2, j_2 + 1 \models Y_1 \wedge Y_2$ and $j_2 + 1 \leq d_2 + 1 \leq d_1 + d_2$.

Now what if (i) is true? Consider the absolute cycle times i and j in the global schedule σ which correspond to the relative indices i_1 and j_2 in the sub-executions σ_1 and σ_2, respectively. Recall that for these indices we have $\sigma, i \models 0 : Y_1$ and $\sigma, j \models 0 : Y_2$. By transition cover $\sigma = \sigma_1 + \sigma_2$ and we must have $j - j_2 \leq |\sigma_1| - 1$ and $i - i_1 \leq |\sigma_2| - 1$. From this is follows that $\mathit{max}(i, j) \leq \mathit{max}(|\sigma_2| + i_1 - 1, |\sigma_1| + j_2 - 1) = \mathit{max}(d_2 + 1 + d_1 - 1, d_1 + 1 + d_2 - 1) = d_1 + d_2$. This proves (26). Since our argument is symmetric in σ_1 and σ_2 we have thus proven (26) whenever $|\sigma_i| \geq d_i + 1$ for some $i = 1, 2$.

One can show in fact that the sum $d_1 + d_2$ is the best *uniform delay* d such that $\sigma \models (d_1, d_2) : \phi_1 \parallel \phi_2$ implies $\sigma \models d_1 + d_2 : \circ(Y_1 \wedge Y_2)$ for *arbitrary* executions σ. More precisely, one can show that for any number $d < d_1 + d_2$ we can find an execution such that $\sigma \models (d_1, d_2) : \phi_1 \parallel \phi_2$ but

[8] Recall that *ic* is a time unit short for '*instruction cycle*'

$\sigma \not\models d : \circ(Y_1 \wedge Y_2)$ (see also Prop. 2). Of course, this is the WCRT for the joint instantaneous execution of T_1 and T_2 under multi-threading ignoring signal cross-dependencies.

To keep matters concise we introduce the derived operator

$$\phi_1 \parallel_{X,Y} \phi_2 =_{df} (\phi_1 \wedge ((X \wedge \neg Y) \supset \mathit{wait})) \parallel (\phi_2 \wedge ((Y \wedge \neg X) \supset \mathit{wait}))$$

with bounds $\mathsf{Bnd}(\phi_1 \parallel_{X,Y} \phi_2) =_{df} \mathsf{Bnd}(\phi_1) \times \mathsf{Bnd}(\phi_2)$ such that $\sigma \models (f, g) : \phi_1 \parallel_{X,Y} \phi_2$ iff $\sigma \models (f, g) : (\phi_1 \wedge ((X \wedge \neg Y) \supset \mathit{wait})) \parallel (\phi_2 \wedge ((Y \wedge \neg X) \supset \mathit{wait}))$.

Multi-threading of Surface Interfaces Let us further assume each thread inside a fork-join has (at most) one instantaneous entry X_i and one instantaneous exit control Y_i and is specified by its surface WCRT interface:

$$\begin{aligned} T_{1,\mathit{srf}} = (d_1, e_1)^T : X_1 \supset (\circ Y_1 \oplus \circ \mathit{wait}) &= X_1 \supset (d_1{:}Y_1 \oplus e_1{:}\mathit{wait}) \\ T_{2,\mathit{srf}} = (d_2, e_2)^T : X_2 \supset (\circ Y_2 \oplus \circ \mathit{wait}) &= X_1 \supset (d_2{:}Y_2 \oplus e_2{:}\mathit{wait}). \end{aligned}$$

The single entry X_i is justified since there are no inter-level transitions which would enter a thread from outside the concurrent block. There may be inter-thread dependencies between concurrent threads, though, but we ignore these for the moment. As explained above, if there are several instantaneous exits Y_{ik} in a thread T_i then these are assumed to have been bundled into a single exit $Y_i = \oplus_k Y_{ik}$, which simply amounts to forming the maximum as we have seen. Moreover, without loss of generality we may suppose that the T_i are normalized so that we have $d_i \leq e_i$ for all sink nodes.

The surface interface of $T_{1,\mathit{srf}} \parallel_{Y_1,Y_2} T_{2,\mathit{srf}}$ (the system seen in Fig. 10 (a) without fork or join) is given by the smallest d, e such that

$$T_{1,\mathit{srf}} \parallel_{Y_1,Y_2} T_{2,\mathit{srf}} \preceq X_1 \wedge X_2 \supset d : (Y_1 \wedge Y_2) \oplus e : \mathit{wait}.$$

To obtain these the timing interface $T_{1,\mathit{srf}} \parallel_{Y_1,Y_2} T_{2,\mathit{srf}}$ is transformed in context[9] $X_1 \wedge X_2$ as follows:

$$\begin{aligned} X_1, X_2 \vdash T_{1,\mathit{srf}} \parallel_{Y_1,Y_2} T_{2,\mathit{srf}} &= (X_1 \supset (d_1{:}Y_1 \oplus e_1{:}\mathit{wait})) \parallel_{Y_1,Y_2} (X_2 \supset (d_2{:}Y_2 \oplus e_2{:}\mathit{wait})) \\ &= (\mathit{true} \supset (d_1{:}Y_1 \oplus e_1{:}\mathit{wait})) \parallel_{Y_1,Y_2} (\mathit{true} \supset (d_2{:}Y_2 \oplus e_2{:}\mathit{wait})) \\ &= (d_1{:}Y_1 \oplus e_1{:}\mathit{wait}) \parallel_{Y_1,Y_2} (d_2{:}Y_2 \oplus e_2{:}\mathit{wait}) \\ &\preceq ((d_1 + d_2){:}(Y_1 \wedge Y_2) \oplus \\ &\qquad (d_1 + e_2){:}(Y_1 \wedge \mathit{wait}) \oplus \\ &\qquad\quad (e_1 + d_2){:}(\mathit{wait} \wedge Y_2) \oplus \\ &\qquad\qquad (e_1 + e_2){:}(\mathit{wait} \wedge \mathit{wait})) \end{aligned}$$

where we exploit Prop. 2 and the equivalence $(\mathit{true} \supset \phi) \cong \phi$. This type adds up all possible ways in which the through paths and sink paths of T_1 and T_2 may interleave. The underlying arithmetic operation is the *Kronecker product* of the surface interfaces

$$\begin{pmatrix} d_1 \\ e_1 \end{pmatrix} \otimes \begin{pmatrix} d_2 \\ e_2 \end{pmatrix} = \begin{pmatrix} (d_1) \begin{pmatrix} d_2 & e_2 \end{pmatrix}^T \\ (e_1) \begin{pmatrix} d_2 & e_2 \end{pmatrix}^T \end{pmatrix} = \begin{pmatrix} d_1 + d_2 & d_1 + e_2 & e_1 + d_2 & e_1 + e_2 \end{pmatrix}^T$$

The type $Y_1 \wedge Y_2$ specifies the cases where each block T_i executes a through path from X_i to Y_i which amounts to a through path of $T_{1,\mathit{srf}} \parallel_{Y_1,Y_2} T_{2,\mathit{srf}}$. The types $Y_1 \wedge \mathit{wait}$ and $\mathit{wait} \wedge Y_2$ cover the interleaving of a through path in one block with a sink path in the other and finally $\mathit{wait} \wedge \mathit{wait}$ types the executions in which both blocks eventually pause through sink paths. Since a single pausing thread forces the whole fork-join block to pause we get that in all cases except the first one, the composition $T_{1,\mathit{srf}} \parallel_{Y_1,Y_2} T_{2,\mathit{srf}}$ pauses, i.e., it executes a sink path. This is reflected

[9] Proving an (in-)equation $\phi \preceq \psi$ in the context $X_1 \wedge X_2$, i.e., $X_1 \wedge X_2 \vdash \phi \preceq \psi$ amounts to proving $\phi \preceq \psi$ using the equations $X_1 \cong \mathit{true}$ and $X_2 \cong \mathit{true}$ (which are equivalent to equation $X_1 \wedge X_2 \cong \mathit{true}$.

by the (in)equations $Y_1 \wedge \mathit{wait} \preceq \mathit{wait}$, $\mathit{wait} \wedge Y_2 \preceq \mathit{wait}$, $\mathit{wait} \wedge \mathit{wait} \cong \mathit{wait}$ which we can use to condense the type of $T_{1,\mathit{srf}} \parallel_{Y_1,Y_2} T_{2,\mathit{srf}}$ into

$$T_{1,\mathit{srf}} \parallel_{Y_1,Y_2} T_{2,\mathit{srf}} \preceq \begin{pmatrix} d_1 + d_2 \\ \mathit{max}(d_1 + e_2, e_1 + d_2, e_1 + e_2) \end{pmatrix} : X_1 \wedge X_2 \supset (\circ(Y_1 \wedge Y_2) \oplus \circ\mathit{wait}) \quad (27)$$

considering that $x{:}\mathit{wait} \oplus y{:}\mathit{wait} \cong \mathit{max}(x, y){:}\mathit{wait}$ in general. This means that the desired through WCRT of T is $d = d_1 + d_2$ and the sink WCRT $e = \mathit{max}(d_1 + e_2, e_1 + d_2, e_1 + e_2)$.

In the computation of the sink WCRT e it is useful to distinguish between sink and non-sink nodes. Provided at least one system T_1 or T_2 is a (normalized) sink node satisfying $d_i \leq e_i$, one can simplify $e = \mathit{max}(d_1{+}e_2, e_1{+}d_2, e_1{+}e_2) = \mathit{max}(d_1, e_1){+}\mathit{max}(d_2, e_2)$. Thus, in this case, we simply take the maximum of instantaneous and sink paths in each block separately and add them up. If none of T_1 or T_2 is a sink node ($e_1 = e_2 = -\infty$) we get $e = \mathit{max}(d_1 + e_2, e_1 + d_2, e_1 + e_2) = -\infty$, i.e., $T1 \parallel_{Y_1,Y_2} T2$ is not a sink node either. This suggest the following simple WCRT strategy on normalized surface interfaces: We always compute the sink WCRT as

$$e = \mathit{max}(d_1, e_1) + \mathit{max}(d_2, e_2).$$

Then, if we find $d = d_1 + d_2 \geq e$, we normalize and put $e = -\infty$. This is very useful since $e = \mathit{max}(d_1, e_1) + \mathit{max}(d_2, e_2)$ is compositional and thus more efficient than the general $e = \mathit{max}(d_1 + e_2, e_1 + d_2, e_1 + e_2)$ from (27).

Another potential optimization arises if both T_1 and T_2 are sink nodes with $d_i \leq e_i$. Then, $\mathit{max}(d_1 + e_2, e_1 + d_2, e_1 + e_2) = e_1 + e_2$. Thus, for sink interfaces, the instantaneous WCRT for concurrent blocks can be calculated separately for the through paths and sink paths. The Kronecker product reduces to coordinate-wise addition

$$\begin{pmatrix} d_1 \\ e_1 \end{pmatrix} \otimes \begin{pmatrix} d_2 \\ e_2 \end{pmatrix} = \begin{pmatrix} (d_1) \cdot (d_2) \\ (e_1) \cdot (e_2) \end{pmatrix} = \begin{pmatrix} d_1 + d_2 \\ e_1 + e_2 \end{pmatrix}$$

which might be exploited for parallelization of WCRT analyses.

Multi-threading of Depth Interfaces The specification for $T_{1,\mathit{srf}} \parallel_{Y_1,Y_2} T_{2,\mathit{srf}}$ was built from the surface interfaces of $T_{1,\mathit{srf}}$ and $T_{2,\mathit{srf}}$ which only cover through paths and sink paths. To get the full picture the concurrent composition still needs to include the source paths and internal paths that make up the depth interface $\mathit{active} \supset (\circ(Y_1 \wedge Y_2) \oplus \circ\mathit{wait})$. Let us look at them now. The interface types of $T_{1,\mathit{dpt}}$, $T_{2,\mathit{dpt}}$ for source and internal paths are, respectively,

$$\begin{aligned} T_{1,\mathit{dpt}} = (s_1, t_1)^T : \mathit{active}_1 \supset (\circ Y_1 \oplus \circ\mathit{wait}) &= \mathit{active}_1 \supset (s_1{:}Y_1 \oplus t_1{:}\mathit{wait}) \\ T_{2,\mathit{dpt}} = (s_2, t_2)^T : \mathit{active}_2 \supset (\circ Y_2 \oplus \circ\mathit{wait}) &= \mathit{active}_2 \supset (s_2{:}Y_2 \oplus t_2{:}\mathit{wait}). \end{aligned}$$

As before we may assume that internal nodes (i.e., those for which $t_i \neq -\infty$) satisfy $s_i \leq t_i$ since otherwise the internal paths would not contribute to the critical path starting in block T_i and we could replace t_i by $-\infty$ without changing the WCRT.

Obviously, to obtain a source path $\mathit{active} \supset \circ(Y_1 \wedge Y_2)$ which leaves the system instantaneously we need to interleave the source paths from T_1 and T_2 and take into account that only one of the threads has paused while the other has terminated instantaneously at exit control Y_i. In general, for every nonempty subset of pausing threads we calculate the interleaving of their source paths, while assuming that all other threads are already terminated instantaneously, contributing nothing.

Unfortunately, we cannot simply put the depth types $T_{1,\mathit{dpt}}$ and $T_{2,\mathit{dpt}}$ in parallel as before, say

$$\begin{aligned} &T_{1,\mathit{dpt}} \parallel_{Y_1,Y_2} T_{2,\mathit{dpt}} \\ &\quad = \mathit{active}_1 \supset (s_1{:}Y_1 \oplus t_1{:}\mathit{wait}) \parallel_{Y_1,Y_2} \mathit{active}_2 \supset (s_2{:}Y_2 \oplus t_2{:}\mathit{wait}) \\ &\quad \preceq \mathit{active}_1 \wedge \mathit{active}_2 \supset (s_1 + s_2) : (Y_1 \wedge Y_2) \oplus \mathit{max}(t_1 + s_2, s_1 + t_2, t_1 + t_2) : \mathit{wait}. \end{aligned}$$

The reason is that if a source path, say from T_1 has WCRT $s_1 = -\infty$, i.e., if T_1 is not a source node, then the overall source WCRT of T would come out as $s_1 + s_2 = -\infty + s_2 = -\infty$ even if T_2 is a source node with $s_2 \geq 0$. In other words, in T_1 would block the execution of the source node T_2. Instead, the right source WCRT would be s_2. This is the delay we get if T_2 starts from its pause and block T_1 has been instantaneously terminated in the previous clock cycle and is waiting at the exit control Y_1 rather than an internal pause node (from which it would never reach exit Y_1 because $s_1 = -\infty$). The key observation is that source-paths in a concurrent composition are not composed conjunctively like the through-paths but disjunctively as seen in the following table:

$\mathit{active}_1 \supset \circ Y_1$	$\mathit{active}_2 \supset \circ Y_2$	$\mathit{active}_1 \wedge \mathit{active}_2 \supset \circ(Y_1 \wedge Y_2)$
$s_1 \geq 0$	$s_2 \geq 0$	$s_1 + s_2$
$s_1 = -\infty$	$s_2 \geq 0$	s_2
$s_1 \geq 0$	$s_2 = -\infty$	s_1
$s_1 = -\infty$	$s_2 = -\infty$	$-\infty$

To implement this disjunctive behavior we might try to coerce $-\infty$ to 0 before we add the paths as in $max(0, s_1) + max(0, s_2)$. However, this would give us a source WCRT of 0 in case $s_1 = s_2 = -\infty$ which is not correct. For if none of the blocks T_1 and T_2 is a source node their composition T should not be a source node either.

The right way to combine the source paths of concurrent blocks T_1, T_2, ..., T_n is to take the maximum over all possible ways in which some subset of blocks are started (active_i) from an internal pause while the other blocks already wait at their exit controls Y_i. The degenerated case where all blocks are at their exits Y_i must be excluded since no source path of the composition T can start in this way. Any path which reaches $Y_1 \wedge Y_2 \wedge \cdots \wedge Y_n$ would have left the block T instantaneously in the previous logical tick.

To achieve this composition we activate the source paths of T not with $\mathit{active} \supset (\mathit{active}_1 \wedge \mathit{active}_2)$ but with

$$\mathit{active} \supset (\mathit{active}_1 \oplus \mathit{active}_2) \wedge (\mathit{active}_1 \oplus \neg \mathit{active}_1) \wedge (\mathit{active}_2 \oplus \neg \mathit{active}_2). \tag{28}$$

This says that once the fork-join block is *active* at least one block T_i executes a source path ($\mathit{active}_1 \oplus \mathit{active}_2$) and also that in those executions the activation controls behave as constants ($\mathit{active}_i \oplus \neg \mathit{active}_i$), i.e., they are either switched on or switched off, throughout. The right-hand side of type (28) is equivalent to listing all the possible cases

$$\mathit{active} \supset (\mathit{active}_1 \wedge \mathit{active}_2) \oplus (\mathit{active}_1 \wedge \neg \mathit{active}_2) \oplus (\neg \mathit{active}_1 \wedge \mathit{active}_2) \tag{29}$$

in which one block is activated and all others are switched off. Note however that this expansion (29) is exponential in the number of concurrent blocks, in contrast to (28) which is linear. In general, for blocks $T_1, T_2, \ldots, T_n$ (28) would be

$$\mathit{active} \supset ((\oplus_i^n \mathit{active}_i) \wedge \bigwedge_{i=1}^{n} (\mathit{active}_{i=1} \oplus \neg \mathit{active}_i)).$$

The other measure to take is to instrument the parallel composition $T_1 \parallel_{Y_1, Y_2} T_2$ so that the source paths of an activated block T_i do not get preempted by the other block T_j if it is not active, too. To make inactive blocks wait at their join we strengthen their specification as follows:

$$T_i^* =_{df} T_i \wedge ((\mathit{active} \wedge \neg \mathit{active}_i) \supset (Y_i \wedge \mathit{wait})). \tag{30}$$

The type $(\mathit{active} \wedge \neg \mathit{active}_i) \supset (Y_i \wedge \mathit{wait})$ adds to T_i waiting configurations specified by $Y_i \wedge \mathit{wait}$ whenever the parallel composition in which T_i is part of is activated (*active*) but T_i itself is not ($\neg \mathit{active}_i$). The type $Y_i \wedge \mathit{wait}$ consists of singleton events in which control Y_i is present. The conjunction $Y_i \wedge \mathit{wait}$ simulates T_i waiting at its exit Y_i.

Putting (30) together with (28) we obtain:

$$\begin{aligned}
\mathit{active}, \mathit{active}_i \ \vdash \ T_i^* &= T_i \wedge ((\mathit{active} \wedge \neg \mathit{active}_i) \supset (Y_i \wedge \mathit{wait})) \\
&= T_i \wedge ((\mathit{true} \wedge \mathit{false}) \supset (Y_i \wedge \mathit{wait})) \\
&= T_i \wedge (\mathit{false} \supset (Y_i \wedge \mathit{wait})) \\
&= T_i \wedge \mathit{true} \\
&= T_i \\
&= \mathit{active}_i \supset (s_i{:}Y_i \oplus t_i{:}\mathit{wait}) \\
&= \mathit{true} \supset (s_i{:}Y_i \oplus t_i{:}\mathit{wait}) \\
&= s_i{:}Y_i \oplus t_i{:}\mathit{wait}.
\end{aligned}$$

$$\begin{aligned}
\mathit{active}, \neg\mathit{active}_i \ \vdash \ T_i^* &= T_i \wedge ((\mathit{active} \wedge \neg \mathit{active}_i) \supset (Y_i \wedge \mathit{wait})) \\
&= T_i \wedge ((\mathit{true} \wedge \mathit{true}) \supset (Y_i \wedge \mathit{wait})) \\
&= T_i \wedge (Y_i \wedge \mathit{wait}) \\
&= (\mathit{active}_i \supset (s_i{:}Y_i \oplus t_i{:}\mathit{wait})) \wedge (Y_i \wedge \mathit{wait}) \\
&= (\mathit{false} \supset (s_i{:}Y_i \oplus t_i{:}\mathit{wait})) \wedge (Y_i \wedge \mathit{wait}) \\
&= \mathit{true} \wedge (Y_i \wedge \mathit{wait}) \\
&= Y_i \wedge \mathit{wait}.
\end{aligned}$$

Now we put the two blocks in parallel in all three activation contexts (29) using the approximation $\phi \wedge \mathit{wait} \preceq \mathit{wait}$ and $x{:}\phi \oplus y{:}\phi \preceq \mathit{max}(x, y){:}\phi$:

$$\begin{aligned}
&\mathit{active}, \mathit{active}_1, \mathit{active}_2 \ \vdash \\
&\qquad T_1^* \parallel_{Y_1,Y_2} T_2^* = (s_1{:}Y_1 \oplus t_1{:}\mathit{wait}) \parallel_{Y_1,Y_2} (s_2{:}Y_2 \oplus t_2{:}\mathit{wait}) \\
&\qquad\qquad \preceq (s_1 + s_2) : (Y_1 \wedge Y_2) \oplus (s_1 + t_2) : (Y_1 \wedge \mathit{wait}) \oplus \\
&\qquad\qquad\qquad (t_1 + s_2) : (\mathit{wait} \wedge Y_2) \oplus (t_1 + t_2) : (\mathit{wait} \wedge \mathit{wait}) \\
&\qquad\qquad \preceq (s_1 + s_2) : (Y_1 \wedge Y_2) \oplus \mathit{max}(t_1 + s_2, s_1 + t_2, t_1 + t_2) : \mathit{wait} \qquad (31) \\
&\mathit{active}, \mathit{active}_1, \neg\mathit{active}_2 \ \vdash \\
&\qquad (T_1^* \parallel_{Y_1,Y_2} T_2^*) = (s_1{:}Y_1 \oplus t_1{:}\mathit{wait}) \parallel_{Y_1,Y_2} (Y_2 \wedge \mathit{wait}) \\
&\qquad\qquad \preceq s_1{:}(Y_1 \wedge Y_2) \oplus t_1{:}(Y_2 \wedge \mathit{wait}). \qquad (32) \\
&\mathit{active}, \neg\mathit{active}_1, \mathit{active}_2 \ \vdash \\
&\qquad (T_1^* \parallel_{Y_1,Y_2} T_2^*) = (Y_1 \wedge \mathit{wait}) \parallel_{Y_1,Y_2} (s_2{:}Y_2 \oplus t_2{:}\mathit{wait}) \\
&\qquad\qquad \preceq t_2{:}(Y_1 \wedge \mathit{wait}) \oplus s_2{:}(Y_1 \wedge Y_2). \qquad (33)
\end{aligned}$$

Finally, we sum up (31)–(33) under *active* as specified in (28), or equivalently (29):

$$\begin{aligned}
\mathit{active} \ \vdash \ T_1^* \parallel_{Y_1,Y_2} T_2^* &\preceq (s_1 + s_2) : (Y_1 \wedge Y_2) \oplus \mathit{max}(t_1 + s_2, s_1 + t_2, t_1 + t_2) : \mathit{wait} \oplus \\
&\qquad s_1{:}(Y_1 \wedge Y_2) \oplus t_1{:}(Y_2 \wedge \mathit{wait}) \oplus \\
&\qquad\quad t_2{:}(Y_1 \wedge \mathit{wait}) \oplus s_2{:}(Y_1 \wedge Y_2) \oplus \\
&\preceq \mathit{max}(s_1, s_2, s_1 + s_2) : (Y_1 \wedge Y_2) \oplus \\
&\qquad \mathit{max}(t_1, t_2, t_1 + s_2, s_1 + t_2, t_1 + t_2) : \mathit{wait},
\end{aligned}$$

again using $\phi \wedge \mathit{wait} \preceq \mathit{wait}$, $x{:}\phi \oplus y{:}\phi \preceq \mathit{max}(x, y){:}\phi$, as well as associativity, commutativity of $\oplus$. This shows that $s = \mathit{max}(s_1, s_2, s_1 + s_2)$ is the source WCRT of $T_1^* \parallel_{Y_1,Y_2} T_2^*$ and $t =_{df} \mathit{max}(t_1, t_2, t_1 + s_2, s_1 + t_2, t_1 + t_2)$ the internal WCRT.

Regarding the source WCRT s, this is what we wanted: The WCRT $s = \mathit{max}(s_1, s_2, s_1 + s_2)$ for source paths of T is the sum of all non-∞ source paths of the sub-blocks T_1 and T_2. If all source paths are $-\infty$ then the source WCRT of T is $-\infty$, too, and T is a not a source node. A simple way top compute s this is by iterative monotonic update: initialize $x_0 \leftarrow -\infty$ and then for each source WCRT s_i of block T_i take $x_{i+1} \leftarrow s_i + \mathit{max}(0, x_i)$.

Next, consider the internal WCRT $t = \mathit{max}(t_1, t_2, t_1 + s_2, s_1 + t_2, t_1 + t_2)$. Let us see how this could be transformed into a sum-of-maxs. Obviously, $t = -\infty$ iff both $t_1 = -\infty$ and $t_2 = -\infty$, i.e., if the composition is an internal node iff one of the blocks is internal. Suppose one of the blocks, say T_1, is internal, whence $t_1 \geq 0$. By normalization we may assume $s_1 \leq t_1$. We claim that the internal WCRT t can also be determined by the sum-of-max expression

$$t' =_{df} \mathit{max}(0, s_1, t_1) + \mathit{max}(0, s_2, t_2)$$

if at least one of T_1, T_2 is an internal node and otherwise $t = -\infty$. First observe that because of $t_1 \geq 0$ and $s_1 \leq t_1$ the first summand of t' reduces to $\mathit{max}(0, s_1, t_1) = t_1$, so that $t' = t_1 + \mathit{max}(0, s_2, t_2) = \mathit{max}(t_1, t_1 + s_2, t_1 + t_2)$ invoking the distributive law $x + \mathit{max}(y_1, y_2) = \mathit{max}(x + y_1, x + y_2)$ of max-plus algebra. Now if $t_2 = -\infty$ then $t' = t$ is easily shown since $t' = \mathit{max}(t_1, t_1 + s_2, t_1 + t_2) = \mathit{max}(t_1, t_1 + s_2)$ and also $t = \mathit{max}(t_1, t_2, t_1 + s_2, s_1 + t_2, t_1 + t_2) = \mathit{max}(t_1, t_1 + s_2)$ due to $x + (-\infty) = -\infty$ and $\mathit{max}(x, y, -\infty) = \mathit{max}(x, y)$. In the other case, both $t_1 \geq 0$ and $t_2 \geq 0$ so we have $t_2 \leq t_1 + t_2$ and $s_1 + t_2 \leq t_1 + t_2$. Then both t_2 and $s_1 + t_2$ can be added to the sums of t', whence $t' = \mathit{max}(t_1, t_1 + s_2, t_1 + t_2) = \mathit{max}(t_1, t_2, t_1 + s_2, s_1 + t_2, t_1 + t_2) = t$ as desired. The calculations are symmetric in t_i which means that if one of the blocks is internal then the internal WCRT can indeed be computed by $t = \mathit{max}(0, s_1, t_1) + \mathit{max}(0, s_2, t_2)$ as claimed.

Putting it Together: Adding Fork and Join Observe that the surface interface $T1 \parallel_{Y_1,Y_2} T2$ does not change if we replace T_i by $T_i^* = T_i \wedge \theta_i$ where $\theta_i =_{df} (\mathit{active} \wedge \neg \mathit{active}_i) \supset (Y_i \wedge \mathit{wait})$. We have

$$T_1^* \parallel_{Y_1,Y_2} T_2^* \preceq \begin{pmatrix} d \\ e \end{pmatrix} : X_1 \wedge X_2 \supset (\circ(Y_1 \wedge Y_2) \oplus \circ \mathit{wait}) \tag{34}$$

with $d = d_1 + d_2$ and $e = \mathit{max}(d_1, e_1) + \mathit{max}(d_2, e_2)$ (subject to normalization). This is because the conjunctive additions θ_i in the latter over the former do not affect the behavior under input controls X_i. In general, $(T_1 \parallel T_2) \preceq \psi$ implies $(T_1^* \parallel T_2^*) \preceq \psi$ since $(T_1 \wedge \theta_1) \parallel (T_2 \wedge \theta_2)$ is a subset of executions of $T_1 \parallel T_2$. Also, we do not lose exactness since every longest path execution of $T_1 \parallel T_2$ which violates $(d', e')^T : \psi$ for $d' < d$ or $e' < e$ do not (need to) involve the *active* control and thus are also executions of $T_1^* \parallel T_2^*$.

On the other hand, we have the depth interface

$$T_1^* \parallel_{Y_1,Y_2} T_2^* \preceq \begin{pmatrix} s \\ t \end{pmatrix} : \mathit{active} \supset \circ(Y_1 \wedge Y_2) \oplus \circ \mathit{wait},$$

with $s = \mathit{max}(s_1, s_2, s_1 + s_2)$ and $t = \mathit{max}(0, s_1, t_1) + \mathit{max}(0, s_2, t_2)$ (subject to normalization as described above). Putting both interfaces together yields

$$T_1^* \parallel_{Y_1,Y_2} T_2^* \preceq \begin{pmatrix} d & s \\ e & t \end{pmatrix} : ((X_1 \wedge X_2) \vee \mathit{active}) \supset \circ(Y_1 \wedge Y_2) \oplus \circ \mathit{wait}.$$

Up to this point we have captured the interleaved execution of threads T_1 and T_2 without the fork and join nodes. This is visualized in Fig. 10 (a) where the horizontal bars denote the concurrent synchronization of T_1 and T_2 at entry and exit controls. In particular it indicates that the combined entry and exit are $X_1 \wedge Y_2$ and $Y_1 \wedge Y_2$, respectively, and that thread T_i yields when reaching Y_i. As depicted in Fig. 10 (b) we add the fork and join nodes which have the types

$$\mathsf{join} \quad \begin{pmatrix} 0 & -\infty \\ -\infty & 0 \end{pmatrix} : ((Y_1 \wedge Y_2) \vee \mathit{wait}) \supset (\circ Y \oplus \circ \mathit{wait})$$

$$\mathsf{fork} \quad \begin{pmatrix} 4 & -\infty \\ -\infty & 1 \end{pmatrix} : (X \vee \mathit{active}(T)) \supset (\circ(X_1 \wedge X_2) \oplus \circ \mathit{active})$$

The specification $(4) : X \supset \circ(X_1 \wedge X_2)$ of fork includes the ics for two PAR, one PARE and one JOIN. The entry $(1) : \mathit{active}(T) \supset \circ \mathit{active}$ takes care of the execution of join on all source and internal paths of T. Since the join is always executed when at least on thread is active, we add the execution time to the fork and not to the join node. Finally we arrive at

$$\begin{pmatrix} 0 & -\infty \\ -\infty & 0 \end{pmatrix} \cdot \begin{pmatrix} d & s \\ e & t \end{pmatrix} \cdot \begin{pmatrix} 4 & -\infty \\ -\infty & 1 \end{pmatrix} = \begin{pmatrix} d & s \\ e & t \end{pmatrix} \cdot \begin{pmatrix} 4 & -\infty \\ -\infty & 1 \end{pmatrix} = \begin{pmatrix} d+4 & s+1 \\ e+4 & t+1 \end{pmatrix}$$

as the WCRT for the composite fork-join block T with canonical interface : $(X \vee \mathit{active}) \supset (\circ Y \oplus \circ \mathit{wait})$. Note that we do not need to re-normalize. If $d \leq e$ then $d+4 \leq e+4$ and if $s \leq t$ then $s+1 \leq t+1$.