Logic Meets Algebra: Compositional Timing Analysis for Synchronous Reactive Multithreading

Michael Mendler[1], Joaquín Aguado[1], Bruno Bodin[2], Partha Roop[3], and Reinhard von Hanxleden[4]

[1] Faculty of Inform. Syst. and Appl. Comp. Sciences, Bamberg University, Germany
{michael.mendler,joaquin.aguado}@uni-bamberg.de
[2] Dept. of Computer Science, Edinburgh University
bbodin@inf.ed.ac.uk
[3] Dept. of Elect. and Comp. Engineering, Auckland University, New Zealand
p.roop@auckland.ac.nz
[4] Dept. of Computer Science, Christian-Albrechts-Universität zu Kiel, Germany
rvh@informatik.uni-kiel.de

Abstract. The intuitionistic theory of the real interval $[0, 1]$, known as Skolem-Gödel-Dummet logic (SGD), generates a well-known Heyting algebra intermediate between intuitionistic and classical logic. Originally of purely mathematical interest, it has recently received attention in Computer Science, notably for its potential applications in concurrency theory. In this paper we show how the logical operators of SGD over the discrete frame $\mathbb{Z}_\infty$, extended by the additive group structure $(\mathbb{Z}, 0, +)$, provides an expressive and yet surprisingly economic calculus to specify the quantitative stabilisation behaviour of synchronous programs. This is both a new application of SGD and a new way of looking at the constructive semantics of synchronous programming languages. We provide the first purely algebraic semantics of timed synchronous reactions which adapts the semantics of Esterel to work on general concurrent/sequential control-flow graphs. We illustrate the power of the algebra for the modular analysis of worst-case reaction time (WCRT) characteristics for time-predictable reactive processors with hardware-supported multi-threading.

1 Introduction

Synchronous control-flow programming (SCP) extends standard imperative programming by deterministic concurrency. This is achieved by forcing threads to execute under the control of a logical clock in lock-step synchronisation, thereby generating a sequence of global *macro steps*, also called *logical instants* or *clock ticks*. During each tick, threads use *signals* to communicate with each other. In contrast to shared variables, signals are accessed using a synchronisation protocol which makes all writes to a signal happen before any read and the value read to be a value uniquely *combined* from the values written. Programs that cannot

be scheduled in this way, tick by tick, are detected at compile-time and rejected as *non-constructive*. Synchronous programs can be compiled into sequential C code, hardware circuits for parallel execution or multi-threaded assembly code.

The physical time spent by the running threads to compute the tick reaction is functionally immaterial, because of the clock synchronisation. The functional semantics of SCP is fully captured by the synchronous composition of Mealy machines. The physical timing of a module can be ignored until it is compiled and mapped to an execution architecture. Then it becomes crucial, however, since the *worst-case reaction time* (WCRT) determines the correct physical synchronisation of the compiled modules and the environment. This WCRT value gives the maximal frequency of the clock and the minimal length of a reaction cycle. Assuming an implementation on clocked instruction set processors, the purpose of the WCRT analysis is to determine, at compile time, the maximal number of instruction cycles in any tick.

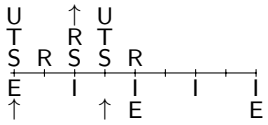
This paper extends previous work by the authors [23, 22, 24, 29, 1] on the WCRT analysis of imperative multi-threaded SCP code running on *Precision-timed (PRET)* architectures. It discusses the Skolem-Gödel-Dummett intuitionistic logic SGD[X] of formal power series for the cycle-accurate modelling of sequential and concurrent program behaviour. Formal power series arise from adjoining an abstract variable X to build polynomials. This is the first time that SGD[X] is presented as a component model, exploring its applications for modular analysis and timing abstractions to trade efficiency and precision. The power of SGD[X] is shown using the Esterel program in Fig. 1 as case study.

We believe the algebraic approach for WCRT analysis of SCP can be an elegant and powerful alternative to other more combinatorial techniques, such as those based on graph traversal [4, 23], state exploration [17, 31], implicit path enumeration with integer linear programming (ILP) solving and iterative narrowing [15, 16, 30, 25] or timed automata [28]. The advantage of SGD[X] algebra over combinatorial definitions of WCRT is that it combines timing and functional specifications in a simple equational calculus. It permits us to study the timed behaviour of SCP employing standard mathematical tools familiar from linear and non-linear system theory. The logical interpretation of SGD[X] supports modular reasoning and timing abstractions at a fine-grained level. Existing WCRT algorithms may be studied as decision procedures for specialised fragments of SGD[X] algebra.

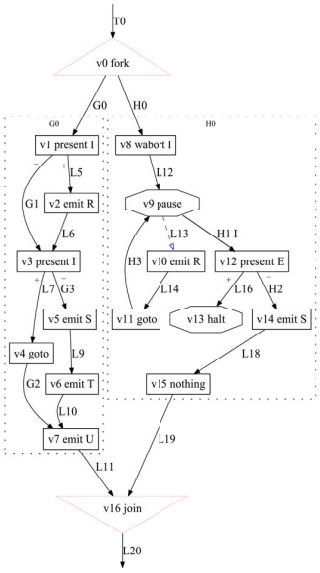
This paper ties up previous work of the authors spread over different publications which have not all used the same mathematical setting. By presenting a single case study, covering many aspects studied separately before, this paper lays out clearly the theoretical background of our approach in a new and uniform way. Regarding the practical usefulness of the approach we refer to our other publications, as cited herein. The relationship with the authors' previous work is discussed both as we go along and in Sec. 6.

```
1  [ % thread G
2    present I then
3      emit R
4    end present;
5    present I else
6      emit S;
7      emit T
8    end present;
9    emit U;
10 ||
11   % thread H
12   weak abort
13     loop
14       pause;
15       emit R
16     end loop
17   when immediate I;
18   present E then
19     halt
20   end present;
21   emit S;
22   nothing;
23 ]
```

(a) Module M



(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
L20:
```

(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,
  L15,L17,L18,L19
```

(e) KEP sample trace

Fig. 1: A simple Esterel module M with its corresponding control-flow graph and the resulting KEP Assembler (example from [13]).

2 Esterel-style Multi-threading and WCRT Analysis

A representative example of a high-level SCP language is Esterel [3]. Esterel signals are either *present* or *absent* during one tick. Signals are set to present by the emit statement and signal state is tested with the present test statement. They are reset to absent at the start of each tick. Esterel statements can be either combined in sequence (;) or in parallel (‖). The loop statement restarts its body when it terminates. All Esterel statements complete within a single tick, called *(instantaneous) termination*, except for the pause statement, which *pauses* for one tick, and derived statements like halt (= loop pause end), which pauses 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 tick 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 tick its body is entered, while the *delayed* version ignores the trigger signal during the first tick in which the abort body is entered.

Consider the Esterel fragment in Fig. 1a, which 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 time line seen in Fig. 1b illustrates a sequence of ticks in which the Esterel program module M in Fig. 1a is activated twice by its execution context, first in tick 1 and then again in tick 4. Below the horizontal line we list the input stimulus at each tick and above the line the reaction output. The arrows indicate when the module is activated (below the time line) and terminated (above the line).

PRET processing architectures have been proposed as a new class of general purpose processors for real-time, embedded applications [8, 7, 19, 27]. PRETs are designed not only to make worst-case execution times predictable, but also to simplify the derivation of this worst case through careful architectural choices. There have also been a number of reactive processor designs dedicated to SCP with instruction set architectures that can express concurrency and preemption and preserve functional determinism [12]. Here we use the Kiel Esterel Processor (KEP) [18], which allows a direct mapping from the control-oriented language Esterel.

The KEP assembly code for our example module M is seen in Fig. 1d. KEP 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 processor determines the active instruction from the thread with the highest priority and executes it. New child threads are initialised by the PAR instruction. The PARE instruction ends the initialisation of parallel threads and sets the program counter of the current thread to the corresponding JOIN. By

changing the priorities of the threads, using PRIO instructions, 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 testing of the signal. For all parallel threads one join instruction is executed, which checks whether all threads have terminated in the current tick. If this is the case, the whole parallel terminates and the join passes the control to the next instruction. Otherwise the join blocks. On KEP, most instructions, like emit or entering an abort block, are executed in exactly one instruction cycle (ic). The pause instruction is executed both in the tick it is entered, and in the tick it is resumed, to check for weak and strong abortions, respectively. Note that the halt instruction is executed in one ic. Priority changing instructions may be treated like the padding statement nothing, which has no effect other than adding a time delay.

The KEP assembler's control flow is represented in the *concurrent KEP assembler graph (CKAG)* depicted in Fig. 1c. The CKAG is an intermediate representation compiled from the Esterel source Fig. 1a which, due to the nature of the KEP architecture, retains much of the original Esterel program structure. It is important to observe that the CKAG and the KEP assembler have a very close and timing-predictable relationship. Hence, the timing of the KEP can be back-annotated in the CKAG by associating WCRT weights to nodes and edges. 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 pre-empted 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 (see Fig. 1c) is straightforward [4]. Most nodes are translated into one instruction, only fork nodes are expanded to multiple instructions to initialise the threads. In our example, the fork v_0 is transformed into three instructions ($L01$–$L03$).

3 Max-Plus Algebra and Skolem-Gödel-Dummet Logic

A standard setting for timing analysis is the discrete max-plus structure over integers $(\mathbb{Z}_\infty, \oplus, \odot, \mathbb{0}, \mathbb{1})$ where $\mathbb{Z}_\infty =_{df} \mathbb{Z} \cup \{-\infty, +\infty\}$ and $\oplus$ is the maximum and $\odot$ stands for addition. Both binary operators are commutative, associative with the neutral elements $\mathbb{0} =_{df} -\infty$ and $\mathbb{1} =_{df} 0$, respectively, i.e., $x \oplus \mathbb{0} = x$ and $x \odot \mathbb{1} = x$. The constant $\mathbb{0}$ is absorbing for $\odot$, i.e., $x \odot \mathbb{0} = \mathbb{0} \odot x = \mathbb{0}$. In particular, $-\infty \odot +\infty = -\infty$. Addition $\odot$ distributes over $\oplus$, i.e., $x \odot (y \oplus z) = x + max(y, z) = max(x + y, x + z) = (x \odot y) \oplus (x \odot z)$. This induces on $\mathbb{Z}_\infty$ a (commutative, idempotent) semi-ring structure. Multiplicative expressions $x \odot y$ are often written $x\,y$ and $\odot$ is assumed to bind more strongly than $\oplus$. Extending $\mathbb{Z}$ to $\mathbb{Z}_\infty$ weakens the ring structure, because the limit values $+\infty$ and $-\infty$ cannot be subtracted. E.g., there is no x such that $x \odot +\infty = \mathbb{0}$.

There is, however, a weak form of negation, the *adjugate* $x^* = -x$ which is an involution $(x^*)^* = x$ and antitonic, i.e., $x \leq y$ iff $x^* \geq y^*$. The adjugate satisfies $x \odot x^* \in \{\mathbb{0}, \mathbb{1}\}$ and $x \odot x^* = \mathbb{1}$ iff x is *finite*, i.e., $x \in \mathbb{Z}$. The set $\mathbb{Z}_\infty$ is not only an adjugated semi-ring but also a lattice with the natural ordering $\leq$. Meet and join are $x \wedge y = \mathit{min}(x, y)$ and $x \vee y = \mathit{max}(x, y)$, respectively. In fact, with its two limits $-\infty$ and $+\infty$ the order structure $(\mathbb{Z}_\infty, \leq, -\infty, +\infty)$ is a complete lattice. The operators $\oplus$, $\odot$ are monotonic and upper continuous. Note that $\odot$ is upper continuous, $x \odot \bigvee_i y_i = \bigvee_i (x \odot y_i)$, but not lower continuous. Indeed, $+\infty \odot \bigwedge_{i \in \mathbb{Z}} -i = +\infty \odot -\infty = -\infty \neq +\infty = \bigwedge_{i \in \mathbb{Z}} +\infty = \bigwedge_{i \in \mathbb{Z}} (+\infty \odot -i)$.

Max-plus algebra is well-known and widely exploited for discrete event system analysis (see, e.g., [2, 10]). What we are going to exploit here, however, is that $\mathbb{Z}_\infty$ also supports logical reasoning, built around the meet (min) operation and the top element of the lattice $(\mathbb{Z}_\infty, \leq)$. The logical view is natural for our application where the values in $\mathbb{Z}_\infty$ represent activation conditions for control flow points, or measure the presence or absence of a signal during a tick. Logical truth, $\top = +\infty$ indicates a signal being *statically present* without giving a definite bound. All other stabilisation values $d \in \mathbb{Z}$ codify *timed presence* which are forms of truth stronger than $\top$. On these multi-valued forms of truth (aka "presence") the meet $\wedge$ acts like logical conjunction while the join $\vee$ is logical disjunction. The bottom element $\bot = -\infty$ corresponding to falsity indicates that a signal is *absent*.

The behaviour of $\bot$ and $\top$, as the truth values for static signals follows the classical Boolean truth tables with respect to $\wedge$ and $\vee$. However, like $\odot$ has no inverse for the limit elements $+\infty$ and $-\infty$, there is no classical complementation for the finite truth values, i.e., those different from $+\infty$ and $-\infty$. For SCP, however, negation is important to model data-dependent branching, priorities and preemption. As it happens, there is a natural *pseudo-complement*, or implication $\supset$, turning the lattice $\mathbb{Z}_\infty$ into an *intuitionistic* logic, or which is the same, a Heyting algebra [5]. The *implication* $\supset$ is the residual with respect to conjunction $\wedge$, i.e, $x \supset y$ is the largest element z such that $x \wedge z \leq y$. It can be directly computed as follows: $x \supset y = y$ if $y < x$ and $x \supset y = +\infty$ if $x \leq y$. Implication internalises the ordering relation in the sense that $x \supset y = \top$ iff $x \leq y$. Taking $x \equiv y$ as an abbreviation of $(x \supset y) \wedge (y \supset x)$, then two values are logically equivalent $x \equiv y = \top$ iff they are identical $x = y$. Implication generates a *pseudo-complement* as $\neg x =_{df} x \supset \bot$ with the property that $\neg x = \top$ if $x = \bot$ and $\neg x = \bot$ if $x > -\infty$. There is also a residual operation $\oslash$ of $\odot$ so that $z \odot x \leq y$ iff $z \leq y \oslash x$. This is a weak form of subtraction so that $y \oslash x = y - x$ if both y and x are finite, $y \oslash x = +\infty$ if $y = +\infty$ or $x = -\infty$ and $y \oslash x = -\infty$ if $-\infty = y < x$ or $y < x = +\infty$. One shows that for all x with $x \odot x^* = \mathbb{1}$ we have $y \oslash x = y \odot x^*$.

The logic $(\mathbb{Z}_\infty, \top, \bot, \wedge, \vee, \supset)$ is isomorphic to the Skolem-Gödel-Dummet logic [6] of the interval $[0, 1] \subset \mathbb{R}$, which is decidable and completely axiomatised by the laws of intuitionistic logic plus the linearity axiom $(x \supset y) \vee (y \supset x)$. This logic, which we name[5] SGD, has played an important role in the study of logics intermediate between intuitionistic and classical logic. It has recently received

[5] Dummett (1959) calls it LC, yet Skolem (1931) and Gödel (1932) studied LC earlier.

attention for its applications in Computer Science, notably as a semantics of fuzzy logic [11], dialogue games [9] and concurrent λ-calculus [14].

For our application of SGD, both its semi-ring $(\mathbb{Z}_\infty, \oplus, \odot, \mathbb{0}, \mathbb{1})$ and intuitionistic truth algebra $(\mathbb{Z}_\infty, \bot, \top, \wedge, \vee, \supset)$ structure are equally important. The former to calculate WCRT timing and the latter to express signals and reaction behaviour. To state that a signal a is present with a worst-case delay of 5 ic we can write the equation $a \oplus 5 = 5$ or the formula $a \supset 5$. That c becomes active within 5 ticks of both signals a and b being present is stated by the formula $c \supset (5 \odot (a \vee b))$. Every SGD expression is at the same time the computation of a WCRT and a logical activation condition.

4 Max-plus Formal Power Series

To capture the behaviour of a program along sequences of macro ticks, we extend the adjuagated semi-ring $\mathbb{Z}_\infty$ to formal power series. A *(max-plus) formal power series*, *fps*, is an ω-sequence

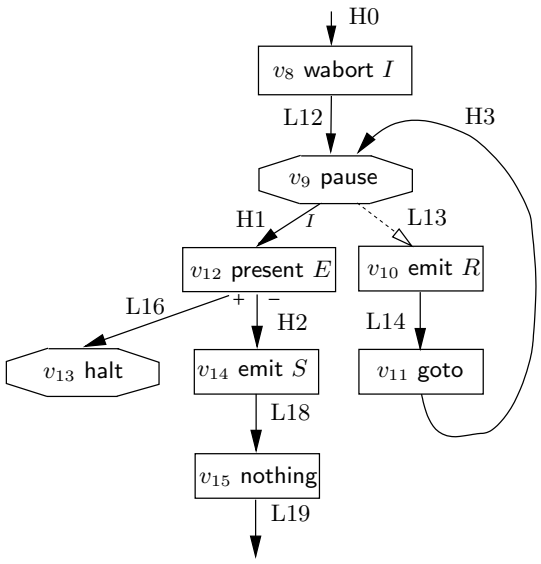
$$A \;=\; \bigoplus_{i\geq 0} a_i X^i \;=\; a_0 \oplus a_1\, X \oplus a_2\, X^2 \oplus a_3\, X^3 \cdots \tag{1}$$

with $a_i \in \mathbb{Z}_\infty$ and where exponentiation is repeated multiplication, i.e., $X^0 = \mathbb{1}$ and $X^{k+1} = X\, X^k = X \odot X^k$. An fps stores an infinite sequence of numbers $a_0, a_1, a_2, a_3, \ldots$ as the scalar coefficients of the base polynomials X^i. An fps A may model the time cost a_i for a thread A to complete each tick i, to reach a given state A or to activate a given signal A. If $a_i = \mathbb{0} = -\infty$ this means that thread A is not executed during the tick i, or that a state A is not reachable. This contrasts with $a_i = \mathbb{1} = 0$ which means A is executed during tick i with zero cost, or that the state A is active at the beginning of the tick. If $a_i > 0$ then thread A is executed taking at most a_i time to finish tick i, or state A is reached within a_i-time during the selected tick. We evaluate A with $X = \mathbb{1}$ for the worst-case time cost $A[\mathbb{1}] = max\{a_i \mid i \geq 0\}$ across all ticks.

Let $\mathbb{Z}_\infty[X]$ denote the set of fps over $\mathbb{Z}_\infty$. For a comprehensive discussion of formal power series in max-plus algebra the reader is referred to [2]. Constants $d \in \mathbb{Z}_\infty$ are naturally viewed as scalar fps $d = d \oplus \mathbb{0}X \oplus \mathbb{0}X^2 \oplus \cdots$. If we want d to be repeated indefinitely, we write an underscore $\underline{d} = d \oplus dX \oplus dX^2 \cdots$. For finite state systems the fps are ultimately periodic. For compactness of notation we write, e.g., $A = \bot{:}2{:}1{:}\underline{4}$ for the ultimately periodic sequence satisfying $A = 2X \oplus 1X^2 \oplus X^3 B$ and $B = 4 \oplus XB$. The semi-ring and logical operations $\star \in \{\oplus, \oslash, \vee, \wedge, \supset, \oslash\}$ are lifted to $\mathbb{Z}_\infty[X]$ in a tick-wise manner, $A \star B = \bigoplus_{i\geq 0}(a_i \star b_i)X^i$ and negation is $\neg A = \bigoplus_{i\geq 0} \neg a_i X^i$. For multiplication $\odot$ there are two ways to lift. First, the tick-wise lifting $A \otimes B = \bigoplus_{i\geq 0}(a_i \odot b_i)X^i$ models multi-threaded parallel composition. It executes A and B synchronously, adding the tick costs to account for the interleaving of instructions. The other "lifting" is *convolution* $A \odot B = \bigoplus_{i\geq 0} \bigoplus_{i=i_1+i_2}(a_{i_1} \odot b_{i_2})X^i$ modelling a form of sequential composition. A special case is *scalar multiplication* $d \odot A = \bigoplus_{i\geq 0}(d \odot a_i)X^i = \underline{d} \otimes A$. The structure $(\mathbb{Z}_\infty[X], \underline{\mathbb{0}}, \underline{\mathbb{1}}, \oplus, \oslash, \otimes, \odot)$ forms

a semi-ring for both "multiplications" $\odot$ and $\otimes$ and $(\mathbb{Z}_\infty[X], \underline{\bot}, \underline{\top}, \wedge, \vee, \supset, \neg)$ is a tick-wise Skolem-Gödel-Dummett logic. To stress the logical interpretation we will denote both as SGD$[X]$ in the sequel.

5 Equational Specification of Synchronous Control-Flow



(a) KEP Assembler Control Flow Graph (CKAG)

```
1  Esterel Program H:
2  weak abort
3     loop
4        pause;
5        emit R
6     end loop
7  when immediate I;
8  present E then
9     halt
10 end present;
11 emit S;
12 nothing;
```

(b) Esterel

```
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) KEP assembler

Fig. 2: The synchronous thread H.

We now go on to illustrate the application of SGD$[X]$ to specify the sequential control flow of our running example in Fig. 1a. We first focus on the thread H consisting of the fragment of nodes v_8–v_{15}, seen in Fig. 2. All edges are instantaneous except the edge $L13$ out of v_9, see below.

Let us assume that each of the basic instructions take 1 instruction cycle (ic) regardless of how they are entered or exited. This is a simplification of the situation in the KEP processor where the delays may be different. We also generally assume in this paper that the code has been checked for causality and that the control flow respects the signal dependencies. This means for the timing of signal communication that input signals may be treated as static booleans, satisfying the axiom $\neg\neg a = a$, or equivalently $a \vee \neg a = \top$, for $a \in \{E, I\}$.

We calculate the time delay to reach a given node A from $H0$ for each tick. More specifically, let $\mathbb{V}$ the set of control primitive variables and $A \in \mathbb{V}$. We identify A with the fps specifying the instants in which the control flow reaches

the control point A. The timing value $A[i]$ at tick i then is the maximal waiting time to reach A in tick i. If $A[i] = \bot = -\infty$ then A cannot be reached in this tick. If we are not interested in the time when A is activated but only whether it is reached, then we use the double negation $\neg\neg A$. This abstracts from the absolute costs and reduces A to a purely *boolean clock*. Sometimes it is useful to abstract not to a boolean but an *arithmetic clock* that is $\mathbb{1}$ when A is present and $\bot$ when it is absent. This collapse is done by the operation $\mathit{tick}(A) = \underline{\mathbb{1}} \wedge \neg\neg A$.

From Fig. 2 we see that edge $L12$ is reached instantaneously in each tick in which control reaches the start edge $H0$, and this is the only way in which $L12$ can be activated. This can be expressed by the equation

$$L12 = 1 \odot H0 = \underline{1} \otimes H0. \tag{2}$$

In each tick, the activation time of $L12$ is 1 instruction cycle (ic) larger than that of the upstream edge $H0$. The conditional branching through node v_{12} depends on the status of (static) signal E. In forward direction, the node v_{12} is:

$$H2 = (1 \odot H1) \wedge \neg E \qquad\qquad L16 = (1 \odot H1) \wedge E \tag{3}$$

The left equivalence states that $H2$ is active in a tick at some ic t iff E is absent and $H1$ was active 1 ic earlier. Analogously, $L16$ is active iff $H1$ was active one ic before and E is present. Algebraically, the equalities can be used to compute $H2$ and $L16$ from $H1$ and E.

Next, consider the pause node v_9. It can be entered by two controls, the line number $L12$ and the program label $H3$ and left via two exits, a non-instantaneous edge $L13$ and an instantaneous exit $H1$ (weak abortion). When a thread enters v_9 then either it terminates the current tick inside the node if I is absent or leaves through the weak abort $H1$ if I is present, thereby continuing the current tick, instantaneously. A thread entering v_9 never exits through $L13$ in the same tick. On the other hand, if a thread is started (resumed) from inside the pause node v_9 then control can only exit through $L13$. Algebraically, we specify the pause node as follows:

$$H1 = (1 \odot (L12 \oplus H3)) \wedge I \tag{4}$$

$$L13 = 1 \odot X \odot \mathit{tick}(\neg I \wedge \neg\neg(L12 \oplus H3)) \tag{5}$$

Equation (4) captures that if a set of schedules activates $H1$ then signal I must be present and one of $L12$ or $H3$ must have been activated 1 ic earlier. Since we are interested in the worst-case we take the maximum. Equation (5) deals with the non-instantaneous exit $L13$ from the pause. The control flow must first pause inside node v_9. This happens in each tick in which one of $L12$ or $H3$ is reached and I is absent. These instants are specified with boolean coefficients by the sub-expression $C = \neg I \wedge \neg\neg(L12 \oplus H3)$. The operator *tick* translates these pausing instances into the neutral element for sequential composition $\odot$. Specifically, $\mathit{tick}(C) = \underline{\mathbb{1}} \wedge \neg\neg C$ forces a coefficient $C = \top = +\infty$ describing presence to become $\mathit{tick}(C) = 0$. On the other hand, $C = \bot = -\infty$ for absence remains unchanged, $\mathit{tick}(C) = \bot$. Finally, the delay $1 \odot X$ shifts the whole time

sequence by one instant and adds a unit delay. This unit delay is the cost of exiting the pause node at the start of the next tick.

The second node with memory behaviour in thread H of Fig. 2 is the halt node v_{13}. Once control flow reaches v_{13} it pauses there forever. Using the auxiliary controls $in(v_{13})$ and $out(v_{13})$ for pausing inside v_{13} and resuming from it, respectively, we get

$$in(v_{13}) = 1 \odot (L16 \oplus out(v_{13})) \qquad out(v_{13}) = 1 \odot X \odot tick(in(v_{13})). \tag{6}$$

The left equation specifies the external entry $L16$ and the fact that exiting the pause immediately re-enters, with 1 ic delay. The right equation states that if the pause is entered it is left in the next tick. Finally, here is the remaining part of H's sequential control flow:

$$L14 = 1 \odot L13 \qquad H3 = 1 \odot L14 \tag{7}$$

$$L16 = 1 \odot (H1 \wedge E) \qquad L18 = 1 \odot H2 \qquad L19 = 1 \odot L18. \tag{8}$$

Well, not quite, we are missing the output signals emitted into the environment. Output responses are generated by thread H in nodes v_{10} and v_{14} as implications

$$R \supset 1 \odot L13 \qquad S \supset 1 \odot H2 \tag{9}$$

assuming a unit delay between activating the emission statement and the appearance of the signal. The implications express only upper bounds $R \leq L13+1$ and $S \leq H2+1$ on the emission of signals R and E. This permits other threads concurrent to H also to emit them, possibly at an earlier time.

The equations (2)–(8) form a recursive equation system with independent variables $H0$, I and E. The recursive dependency of variables $L13$, $L14$ and $H3$ on themselves is guarded by the X operator. Hence, for each fixed choice of the independents $H0$, I and E, all the dependents $L12$–$L19$ and $H1$–$H3$ can be solved uniquely. Let us go though the motions to see how this works. To power up the system as in example trace Fig. 1b we activate the start control $H0$ in the first and again in the fourth tick, with initial delay of 3 to account for the upstreaming fork, $H0 = 3{:}\bot{:}\bot{:}3{:}\underline{\bot}$. Signal I is absent initially and then present every second instant, and E is present every fourth tick, $I = \bot{:}\bot{:}(\top \oplus I)$ and $E = \top{:}\bot{:}\bot{:}\bot{:}E$. Note that $\neg I = \top{:}\top{:}(\bot{:}\underline{\top} \wedge \neg I) = \top{:}\top{:}\bot{:}\top{:}(\bot{:}\underline{\top} \wedge \neg I)$. First, it follows $L12 = 1 \odot H0 = 4{:}\bot{:}\bot{:}4{:}\underline{\bot}$. From (7) we get $H3 = 1 \odot L14 = 1 \odot 1 \odot L13 = 2 \odot L13$ and so equation (5) becomes

$$L13 = f(L13) = 1 \odot X \odot tick(\neg I \wedge \neg\neg(4{:}\bot{:}\bot{:}4{:}\underline{\bot} \oplus (2 \odot L13))). \tag{10}$$

This is solvable by least fixed point iteration starting with $L13_0 = \bot$ for which we get $L13_1 = f(L13_0) = \bot{:}1{:}\bot{:}\bot{:}1{:}\underline{\bot}$. The second iteration through (10) yields $L13_2 = f(L13_1) = \bot{:}1{:}1{:}\bot{:}1{:}\underline{\bot}$ which is already the fixed point, $L13 = L13_2 = f(L13_2)$. The solution $L13 = \bot{:}1{:}1{:}\bot{:}1{:}\underline{\bot}$ corresponds to the trace in Fig. 1b with the WCRT value guaranteeing $L13$ is always reached 1 ic after the beginning of the tick. The closed solution for $L13$ generates a closed solution

for $L14$ and $H3$ by simple substitution, viz. $L14 = 1 \odot L13 = \bot{:}2{:}2{:}\bot{:}2{:}\underline{\bot}$ and $H3 = 1 \odot L14 = \bot{:}3{:}3{:}\bot{:}3{:}\underline{\bot}$. Similarly, we obtain $H1$ from (4), $H1 = 1 \odot (I \wedge (L12 \oplus H3)) = \bot{:}\bot{:}4{:}\bot{:}4{:}\underline{\bot}$. Indeed $H1$ is activated exactly in ticks 2 and 4 with a delay of 4. Since E is absent in tick 2 but present in tick 4, control moves to $H2$ the first time and to $L16$ the second time: The equations give $H2 = 1 \odot (H1 \wedge \neg E) = 1 \odot (\bot{:}\bot{:}4{:}\bot{:}4{:}\underline{\bot} \wedge \bot{:}\top{:}\top{:}\top{:}\neg E) = \bot{:}\bot{:}5{:}\underline{\bot}$. Finally, for $L16$ we have $L16 = 1 \odot (H1 \wedge E) = \bot{:}\bot{:}\bot{:}\bot{:}5{:}\underline{\bot}$.

To sum up, equations (2)–(8) describe the cycle-accurate semantics of thread H in Fig. 2. It is timing and causality sensitive and fully parametric in environment signals. Note that the algebraic specification method outlined in this section is completely uniform and generalises to arbitrary CKAG concurrent control-flow graphs.

5.1 WCRT Component Model

The specification technique described above is fully expressive for Esterel-style synchronous control flow. It is compositional at the level of the primitive controls of the flat control flow graph. It is not modular, however, as it does not permit structural abstraction. An axiomatic specification language that permits behavioural abstraction for timed synchronous components, called *(first-order, elementary) WCRT-interfaces* has been proposed in [23]. It is based on realisability semantics for constructive logic and was formalised in [22]. These interfaces capture the purely combinational behaviour CKAGs, i.e., single ticks. They do not describe the sequential dependencies across sequences of ticks. By translating the model of [23, 22] into SGD[X] algebra we now extend WCRT interfaces for a full semantics of synchronous components.

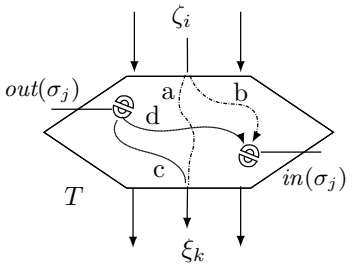


Fig. 3: The four types of thread paths: through path (a), sink path (b), source path (c), internal path (d) (taken from [23]).

The key for modularity is to move from primitive control variables $\mathbb{V}$ to a description based on *(synchronous) reactive blocks*. Fig. 3 depicts a program fragment T abstracted into a reactive block with entry and exit controls. The paths inside T seen in Fig. 3 illustrate the four ways in which a reactive block may participate in the execution of a logical tick: Threads may (a) arrive at some *entry control* ζ_i, pass straight through the block and leave at some *exit control* ξ_k; (b) enter through ζ_i but pause inside in some *state control* $in(\sigma_j)$, waiting there for the next tick; (c) start the tick inside the block from a state $out(\sigma_j)$ and eventually (instantaneously) leave through some exit control ξ_k, or (d) start and pause inside the block, not leaving it during the current tick. These paths are called *through paths* (a), *sink paths* (b), *source paths* (c) and *internal paths* (d), respectively.

Each block T is described by a multi-dimensional WCRT system function in SGD$[X]$ viewing it as a Mealy automaton over control variables. Let us suppose for the moment, that the block T has only one entry ζ, one exit ξ and one state control σ. The system function for such a block is given as a forward transformation matrix T which connects the logical interface controls in the $\{\oplus, \otimes\}$-fragment of SGD$[X]$:

$$\begin{pmatrix} \xi \\ in(\sigma) \end{pmatrix} = T \otimes \begin{pmatrix} \zeta \\ out(\sigma) \end{pmatrix} = \begin{pmatrix} T.thr & T.src \\ T.snk & T.int \end{pmatrix} \otimes \begin{pmatrix} \zeta \\ out(\sigma) \end{pmatrix} \tag{11}$$

All entries of the matrix are logical time series describing the *tick-wise* WCRT behaviour on the four types of control paths: $T.thr$ for the through path, $T.snk$ for the sink paths, $T.src$ for the source paths and $T.int$ for the internal paths. Blocks T with more than one entry, exit or state controls have a system matrix T with more columns and rows, accordingly. Unfolding the matrix multiplication (11) we get the SGD$[X]$ equations

$$\xi = (T.thr \otimes \zeta) \oplus (T.src \otimes out(\sigma)) \tag{12}$$

$$in(\sigma) = (T.snk \otimes \zeta) \oplus (T.int \otimes out(\sigma)) \tag{13}$$

The equation (12) determines the timing at exit ξ as the tick-wise worst-case $\oplus$ of two contributions, those activations arriving from entry ζ increased by the weight of the through path $T.thr$ and those arriving from a state control $out(\sigma)$ inside T increased by the weight of the source path $T.src$. The increase is achieved by $\otimes$ in SGD$[X]$ which is the tick-wise addition $\odot$ in SGD. In an analogous way, equation (13) captures the activities arriving at the state control $in(\sigma)$ which may also come from entry ξ or a state $out(\sigma)$. It is useful to split (11) column-wise

$$\big(\xi \;\; in(\sigma)\big)^{\intercal} = \big(\big(T.thr \;\; T.snk\big)^{\intercal} \otimes \zeta\big) \;\oplus\; \big(\big(T.src \;\; T.int\big)^{\intercal} \otimes out(\sigma)\big). \tag{14}$$

thereby obtaining what are called the *surface* and *depth* behaviours $T.srf = \big(T.thr \;\; T.snk\big)^{\intercal}$ and $T.dpt = \big(T.src \;\; T.int\big)^{\intercal}$, which can be manipulated separately.

The equation (11) expresses the purely combinational behaviour of T. The passage from one tick to the next arises by coupling $out(T)$ and $in(T)$ through the *register equation*

$$out(T) = 1 \odot X \odot tick(in(T)). \tag{15}$$

Note the generality of the pseudo-linear system model (11). All matrix entries $T.thr$, $T.src$, $T.snk$, $T.int$ and the input and output variables ζ, $out(\sigma)$, ξ and $in(\sigma)$ may be arbitrary SGD$[X]$ expressions involving arithmetical and logical operators. For instance, the main thread T of Fig. 1c has state control such as $\neg L11 \wedge in(v_9)$, capturing ticks in which child H is pausing in node v_9 while child G has already terminated in a previous tick, whence $L11$ has value $\bot$, and *a fortiori*, all other nodes v in G satisfy $\neg v$, too. In this way, the equation (11) can specify both the temporal and the logical behaviour of block T. This will become clear in the next section.

5.2 Module Abstraction

Pseudo-linear specifications like (11) generalise to composite blocks what the equations (2)–(8) do for primitive controls. The vector formulation can be applied as a component model at various levels of abstraction.

For instance, take the pause node v_9 in Fig. 2 as a primitive block with the "forward" equations (4) and (5). It has entry controls $L12$, $H3$ and exit controls $H1$ and $L13$. The auxiliary controls $in(v_9)$ and $out(v_9)$ express conditions for pausing inside the node and for exiting it, respectively. As shown below, equations (4)–(5) induce the surface and depth behaviours $v_9 = \big(v_9.srf \;\; v_9.dpt\big)$ with

$$\big(H1 \; L13 \; in(v_9)\big)^{\intercal} = (v_9.srf \otimes \big(L12 \; H3\big)^{\intercal}) \oplus (v_9.dpt \otimes out(v_9)) \tag{16}$$

$$v_9.srf = \begin{pmatrix} \underline{1} \wedge I & \underline{1} \wedge I \\ \underline{\bot} & \underline{\bot} \\ \neg I & \neg I \end{pmatrix} \qquad v_9.dpt = \begin{pmatrix} \underline{\bot} \\ \underline{\mathbb{1}} \\ \underline{\bot} \end{pmatrix}. \tag{17}$$

Notice how the entries combine timing with logical conditions. In particular, the constant $\underline{\bot}$ indicates where control flows are absent. If we unfold the matrix multiplications in (16) together with (17) we get the following explicit equations:

$$H1 = ((\underline{1} \wedge I) \otimes L12) \oplus ((\underline{1} \wedge I) \otimes H3) \oplus \bot \; out(v_9) \tag{18}$$

$$L13 = \bot \; L12 \oplus \bot \; H3 \oplus \mathbb{1} \; out(v_9) \tag{19}$$

$$in(v_9) = (\neg I \otimes L12) \oplus (\neg I \otimes H3) \oplus \bot \; out(v_9). \tag{20}$$

slightly simplified using the law $\underline{d} \otimes x = d\,x$. The first equation (18) can be seen as logically equivalent to (4) considering a number of laws, such as $\bot \odot x = \bot$, $x \oplus \bot = x$, $(\underline{d} \otimes x) \wedge I = (\underline{d} \wedge I) \otimes x$ for static signal I, and that both $\odot$ and $\wedge$ distribute over $\oplus$. Also one shows that (19) and (20) in combination with the register equation $out(v_9) = 1 \odot X \odot tick(in(v_9))$ is the same as (5).

At a higher level of the component hierarchy we can consider thread H in Fig 2 as a composite block. Its behaviour is given by the global 3x3 matrix

$$\begin{pmatrix} L19 \\ in(v_9) \\ in(v_{13}) \end{pmatrix} = \begin{pmatrix} \underline{5} \wedge I \wedge \neg E & \underline{7} \wedge I \wedge \neg E & \underline{\bot} \\ \underline{2} \wedge \neg I & \underline{4} \wedge \neg I & \underline{\bot} \\ \underline{4} \wedge I \wedge E & \underline{6} \wedge I \wedge E & \underline{1} \end{pmatrix} \otimes \begin{pmatrix} H0 \\ out(v_9) \\ out(v_{13}) \end{pmatrix} \tag{21}$$

which is the exact behavioural description of H equivalent to the equations (2)–(8), solely in terms of the external controls and the internal states v_9 and v_{13}.

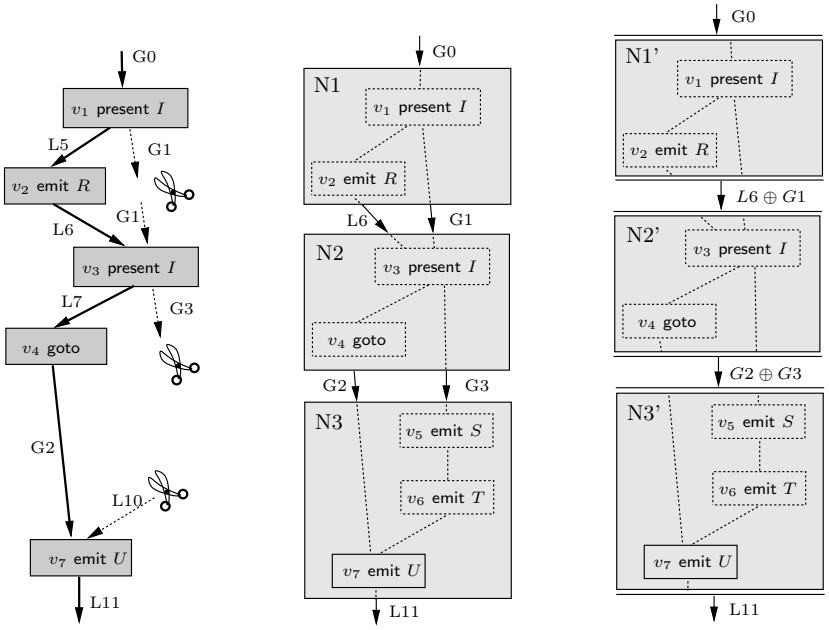
From here we may reduce the complexity and precision in various way. For instance, we may abstract from the state information, working with a single state control $in(H) = in(v_9) \oplus in(v_{13})$ and $out(H) = out(v_9) \oplus out(v_{13})$. This collapse is a "base transformation" achieved by pre- and post-multiplication of H with suitable matrices. Specifically, the expansions

$$\begin{pmatrix} L19 \\ in(H) \end{pmatrix} = \begin{pmatrix} \underline{0} & \underline{\bot} & \underline{\bot} \\ \underline{\bot} & \underline{0} & \underline{0} \end{pmatrix} \otimes \begin{pmatrix} L19 \\ in(v_9) \\ in(v_{13}) \end{pmatrix} \quad \begin{pmatrix} H0 \\ out(v_9) \\ out(v_{13}) \end{pmatrix} \leq \begin{pmatrix} \underline{0} & \underline{\bot} \\ \underline{\bot} & \underline{0} \\ \underline{\bot} & \underline{0} \end{pmatrix} \otimes \begin{pmatrix} H0 \\ out(H) \end{pmatrix}$$

permit us to approximate (21) via a 2x2 matrix H_1

$$\big(L19\ in(H)\big)^{\mathsf{T}} \leq H_1 \otimes \big(H0\ out(H)\big)^{\mathsf{T}} \tag{22}$$

$$H_1 = \begin{pmatrix} \underline{5} \wedge I \wedge \neg E & \underline{7} \wedge I \wedge \neg E \\ (\underline{2} \wedge \neg I) \oplus (\underline{4} \wedge I \wedge E) & (\underline{4} \wedge \neg I) \oplus (\underline{6} \wedge I \wedge E) \end{pmatrix} \tag{23}$$



(a) Path Decomposition (b) Net Decomposition (c) Bundle Decomposition

Fig. 4: Different Structural Decompositions of Thread G

Let us suppose we know that input signals E and I are always opposite values. The associated invariant $I = \neg E$ and $\neg I = E$ implies $x \wedge I \wedge \neg E = x \wedge \neg E$ as well as $x \oplus (y \wedge I \wedge E) = x \oplus \underline{\bot} = x$. In a next step we may decide to give up tracking signal E, abstracting from its value with the over-approximations $x \wedge \neg E \leq x$ and $x \wedge E \leq x$. This yields a sequence of approximated behaviours

$$H \leq H_1 = H_2 =_{df} \begin{pmatrix} \underline{5} \wedge \neg E & \underline{7} \wedge \neg E \\ \underline{2} \wedge E & \underline{4} \wedge E \end{pmatrix} \leq \begin{pmatrix} \underline{5} & \underline{7} \\ \underline{2} & \underline{4} \end{pmatrix} =_{df} H_3. \tag{24}$$

There are further combinatorial optimisations possible that can be justified algebraically in SGD[X]. For instance, the WCRT algorithm [4] reduces the dimensions of the surface and depth behaviours each by one. This exploits the fact that every schedule reaching $in(H)$ is pausing inside H and thus cannot be

extended to a longer instantaneous path of H. In other words, all paths that have length at least $1 \odot in(H)$ must be going through $L19$. Logically, this is the axiom $L19 \oplus d\, in(H) = L19$ for all $d \geq 1$. Under this assumption, the two systems

$$\begin{pmatrix} L19 \\ in(H) \end{pmatrix} = \begin{pmatrix} \underline{5} & \underline{7} \\ \underline{2} & \underline{4} \end{pmatrix} \otimes \begin{pmatrix} H0 \\ out(H) \end{pmatrix} \qquad L19 = (\underline{5}\ \underline{7}) \otimes \begin{pmatrix} H0 \\ out(H) \end{pmatrix}$$

are equivalent. This reduces the WCRT specification further from H_3 to $H_4 = (\underline{5}\ \underline{7})$ without loss of precision. The algorithm [4] exploits this interface optimisation aggressively, at all levels. This renders the analysis of parallel composition particularly efficient, as we shall see in Sec. 5.3.

In more recent work a different abstraction via so-called *tick cost automata (TCA)* has been proposed [29]. It abstracts from signal dependencies like [4], but preserves the dependency on state controls. Also, it is assumed that there are no through paths $T_{thr} = \underline{\bot}$ (Moore automaton) and the unique entry control ζ is connected to a single state s_0 with zero cost. These restrictions are without loss of generality as they can be achieved by judicious block decomposition. We can understand TCA in terms of SGD[X] using abbreviations $in(\boldsymbol{s}) = (in(s_0)\ in(s_1)\ \cdots\ in(s_{n-1})^\intercal$ and $out(\boldsymbol{s}) = (out(s_0)\ out(s_1)\ \cdots\ out(s_{n-1}))^\intercal$ for the state controls vectors. The general system equations then are $\xi = T_{exit} \otimes out(\boldsymbol{s})$, and $in(\boldsymbol{s}) = T_{tick} \otimes out(\boldsymbol{s})$ together with the entry $in(s_0) = \zeta$ and the register equation $out(\boldsymbol{s}) = X\, tick(in(\boldsymbol{s}))$. These system equations in which T_{exit} and T_{tick} consist of scalars $\mathbb{Z}_\infty$ are solved by numeric fixed point iteration.The work [29] implements these operations using max-plus algebra and explicit normal form TCAs representing the ultimately periodic system solutions.

$$\begin{pmatrix} L5 \\ G1 \end{pmatrix} = \begin{pmatrix} \underline{1} \wedge I \\ \underline{1} \wedge \neg I \end{pmatrix} \otimes G0$$
$$L11 = (\underline{1}\ \underline{1}) \otimes \begin{pmatrix} G2 \\ L10 \end{pmatrix}$$
$$\begin{pmatrix} L7 \\ G3 \end{pmatrix} = \begin{pmatrix} \underline{1} \wedge I & \underline{1} \wedge I \\ \underline{1} \wedge \neg I & \underline{1} \wedge \neg I \end{pmatrix} \otimes \begin{pmatrix} L6 \\ G1 \end{pmatrix}$$
$$G2 = \underline{1} \otimes L7$$
$$L6 = \underline{1} \otimes L5$$
$$L9 = \underline{1} \otimes G3$$
$$L10 = \underline{1} \otimes L9$$

Fig. 5: Basic blocks v_1–v_7 of thread G.

5.3 Module Composition

SGD[X] permits compositional specifications at different abstraction levels using (max-plus) pseudo-linear transformations. This is the key for dynamic programming techniques and suggests the composition of blocks by matrix multiplication. Depending on how we apply the algebra we can implement different strategies for practical WCRT analysis. We illustrate this for our example program in Fig. 1c. The starting point is the block description of thread G seen in Fig. 5.

Path Decomposition. 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. Each of these paths defines a sub-graph of G with specific side-inputs and side-outputs. For instance, path p_1 as indicated in Fig. 4a has the side-outputs $G1$, $G3$ and side-inputs $G1$, $L10$. Its SGD[X] reaction function $\big(G1\ G3\ L11\big)^{\mathsf{T}} = D_1 \otimes \big(G0\ L10\ G1\big)^{\mathsf{T}}$ has the system matrix in Fig. 6.

The entries measure if and how p_1 connects the respective controls. For instance, the entry $\underline{3} \wedge I \wedge \neg I$ is the delay between input $G0$ and output $G3$. This segment (see Fig. 4a) has delay 3 but is only sensitisable if signal I is simultaneously present and absent. This is impossible since $\underline{3} \wedge I \wedge \neg I = \bot$. The entries $\underline{\bot}$ in D_1 capture 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 fps max-plus algebra) the timing matrices of the individual nodes traversed by p_1.

$$D_1 = \begin{pmatrix} \underline{1} \wedge \neg I & \underline{\bot} & \underline{\bot} \\ \underline{3} \wedge I \wedge \neg I & \underline{\bot} & \underline{1} \wedge \neg I \\ \underline{5} \wedge I & \underline{1} & \underline{3} \wedge I \end{pmatrix}$$

Fig. 6: System matrix for path p_1.

If we are not interested in all combinations of side-inputs and side-outputs we can reduce the matrix D_1. The side-inputs $G1$ and $L10$ are eliminated by selecting only the first column of D_1, i.e., $D_1' = D_1 \otimes \big(\underline{0}\ \underline{\bot}\ \underline{\bot}\big)^T$, so that $\big(G1\ G3\ L11\big)^T = D_1' \otimes G0$. Getting rid of the side-outputs $G1$ and $G3$ is not so simple. We cannot simply drop the rows and write $L11 = (\underline{5} \wedge I) \otimes G0$. This would be unsound since not every execution of path p_1 exiting from $L11$ must necessarily originate in $G0$ and imply that I is present. What *is* correct, is to say that $L11$ is equivalent to $(\underline{5} \wedge I) \otimes G0$ if neither side-output $G1$ or $G3$ ever becomes active is the set of control flows determining the WCRT. Formally, this is $(\neg G1 \wedge \neg G3) \supset (L11 = (\underline{5} \wedge I) \otimes G0)$. Calculating all other paths through G in a similar fashion finally obtains:

$$p_1 : (\neg G1 \wedge \neg G3) \supset (L11 = D_1'' \otimes G0) \qquad D_1'' = (\underline{5} \wedge I) \tag{25}$$
$$p_2 : (\neg L5 \wedge \neg G3) \supset (L11 = D_2'' \otimes G0) \qquad D_2'' = (\underline{4} \wedge I \wedge \neg I) \tag{26}$$
$$p_3 : (\neg G1 \wedge \neg L7) \supset (L11 = D_3'' \otimes G0) \qquad D_3'' = (\underline{6} \wedge I \wedge \neg I) \tag{27}$$
$$p_4 : (\neg L5 \wedge \neg L7) \supset (L11 = D_4'' \otimes G0) \qquad D_4'' = (\underline{5} \wedge \neg I). \tag{28}$$

The path schedules (25)–(28) can now be woven together in SGD[X] algebra to obtain the final result $L11 = D \otimes G0$ where $D = D_1'' \oplus D_2'' \oplus D_3'' \oplus D_4'' = \underline{5}$. For this we exploit, among other laws, that $I \wedge \neg I = \underline{\bot}$, $I \oplus \neg I = \underline{\top}$ as well as that $x_i \supset (L11 = y_i)$ implies $\oplus_i x_i \supset (L11 = \oplus y_i)$, and the equation

$$(\neg G1 \wedge \neg G3) \oplus (\neg L5 \wedge \neg G3) \oplus (\neg G1 \wedge \neg L7) \oplus (\neg L5 \wedge \neg L7) \ \equiv\ \underline{\top}.$$

The latter is a consequence of the fact that G is single-threaded: Each activation 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.

Weaving Nets. WCRT analysis by path enumeration, though sound, is of worst-case exponential complexity. A more efficient way of going about is to exploit dynamic programming. In the following we illustrate this process in SGD$[X]$ algebra using the net decomposition of G seen in Fig. 4b. The strategy is to propagate WCRT information forward through G, composing sub-nets $N1$, $N2$, $N3$ rather than paths.

We obtain the system matrix of $N1$ first by combining the matrices of v_1 and v_2 from Fig. 5. To compose them we first lift v_2 as an equation in $L5$ and $G1$ to get $L6 = 1\,L5 \oplus \bot\,G1 = \begin{pmatrix}\underline{1} & \underline{\bot}\end{pmatrix} \otimes \begin{pmatrix}L5\; G1\end{pmatrix}^{\mathsf{T}}$. Since $G1 = \begin{pmatrix}\underline{\bot} & \underline{0}\end{pmatrix} \otimes \begin{pmatrix}L5\; G1\end{pmatrix}^{\mathsf{T}}$ we can compose with equation v_1:

$$\begin{pmatrix}L6\\ G1\end{pmatrix} = \begin{pmatrix}\underline{1} & \underline{\bot}\\ \underline{\bot} & 0\end{pmatrix} \otimes \begin{pmatrix}L5\\ G1\end{pmatrix} = \begin{pmatrix}\underline{1} & \underline{\bot}\\ \underline{\bot} & \underline{0}\end{pmatrix} \otimes \begin{pmatrix}\underline{1} \wedge I\\ \underline{1} \wedge \neg I\end{pmatrix} G0 = \begin{pmatrix}\underline{2} \wedge I\\ \underline{1} \wedge \neg I\end{pmatrix} G0. \quad (29)$$

In a similar fashion one obtains the specifications of sub-blocks $N2$ and $N3$:

$$\begin{pmatrix}G2\\ G3\end{pmatrix} = \begin{pmatrix}\underline{2} \wedge I & \underline{2} \wedge I\\ \underline{1} \wedge \neg I & \underline{1} \wedge \neg I\end{pmatrix} \otimes \begin{pmatrix}L6\\ G1\end{pmatrix} \qquad L11 = \begin{pmatrix}\underline{1} & \underline{3}\end{pmatrix} \otimes \begin{pmatrix}G2\\ G3\end{pmatrix}. \quad (30)$$

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:

$$L11 = \begin{pmatrix}\underline{1} & \underline{3}\end{pmatrix} \otimes \begin{pmatrix}\underline{2} \wedge I & \underline{2} \wedge I\\ \underline{1} \wedge \neg I & \underline{1} \wedge \neg I\end{pmatrix} \otimes \begin{pmatrix}\underline{2} \wedge I\\ \underline{1} \wedge \neg I\end{pmatrix} G0 = \underline{5}\,G0. \quad (31)$$

This is indeed the weight of the longest path p_3 through G.

Bundling Abstractions. There are of course other ways of arriving at 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 system equation for $N1$ into a single entry-exit delay $N1'$ specified as $L6 \oplus G1 = d\,G0$ which gives the maximal delay d for an execution entering through $G0$ to come out at $L6$ or $G1$, without distinguishing between paths exiting on $L6$ and those exiting on $G1$. This is applied also to $N2$ and $N3$ as indicated in Fig. 4c.

Algebraically, this compression is justified for $N1$ by pre-composing with $\begin{pmatrix}0\; 0\end{pmatrix}$ which yields $L6 \oplus G1 = \begin{pmatrix}\underline{0}\; \underline{0}\end{pmatrix} \otimes \begin{pmatrix}L6\; G1\end{pmatrix}^{\mathsf{T}} = \begin{pmatrix}\underline{0}\; \underline{0}\end{pmatrix} \otimes \begin{pmatrix}\underline{2} \wedge I\; \underline{1} \wedge \neg I\end{pmatrix}^{\mathsf{T}} \otimes G0 = (\underline{2} \wedge I \oplus \underline{1} \wedge \neg I) \otimes G0$. For $N2$ and $N3$ we also need compression on the input side. For $N2$ this is possible without losing precision and for $N3$ we need the approximation $\begin{pmatrix}G2\; G3\end{pmatrix}^{\mathsf{T}} \leq \begin{pmatrix}\underline{0}\; \underline{0}\end{pmatrix}^{\mathsf{T}} \otimes (G2 \oplus G3)$. We get approximations $N2'$ and $N3'$ from (29) and (30):

$$G2 \oplus G3 = \begin{pmatrix}\underline{0}\; \underline{0}\end{pmatrix} \otimes \begin{pmatrix}\underline{2} \wedge I & \underline{2} \wedge I\\ \underline{1} \wedge \neg I & \underline{1} \wedge \neg I\end{pmatrix} \otimes \begin{pmatrix}L6\\ G1\end{pmatrix} = (\underline{2} \wedge I \oplus \underline{1} \wedge \neg I)(L6 \oplus G1)$$

$$L11 = \begin{pmatrix}\underline{1} & \underline{3}\end{pmatrix} \otimes \begin{pmatrix}G2\\ G3\end{pmatrix} \leq \begin{pmatrix}\underline{1} & \underline{3}\end{pmatrix} \otimes \begin{pmatrix}\underline{0}\\ \underline{0}\end{pmatrix} (G2 \oplus G3) = 3\,(G2 \oplus G3).$$

Composing $N1'$, $N2'$, $N3'$ is more efficient than composing $N1$, $N2$, $N3$ since it involves only scalars rather than matrices.

Parallel Composition and WCRT Analysis. The main thread T in Fig. 1c is the parallel composition of threads G and H, synchronised by the fork and join nodes v_0 and v_{16}, respectively. Even without reducing threads G and H to their externally observable system functions (21) and (31) we can compose them in parallel. All we need are equations for the fork and join nodes. The fork node v_0 activates both $G0$ and $H0$, when it is reached, taking 3 ics (2 PAR and 1 PARE, see Fig. 1d):

$$\big(G0\ H0\big)^{\mathsf{T}} = \big(\underline{3}\ \underline{3}\big)^{\mathsf{T}} \otimes T0. \tag{32}$$

The join node v_{16} becomes active as soon as one of G or H reaches its termination control. The join finishes its activity in the tick when both have arrived. It then passes control and reactivates the parent at $L20$. At each input $L11$, $L19$ the join behaves like a synchroniser with latching behaviour. We define the operator $\mathsf{sync}(C, R)$, which waits for C to become active at which point it inherits the cost of C. From the next tick onwards it takes a constant cost[6], say 2 ics, until it is reset by R. This leads to the recursive definitions

$$\mathsf{sync}(C, R) = \neg XR \wedge (C \oplus X(\underline{2} \wedge \neg\neg\mathsf{sync}(C, R))) \tag{33}$$

$$L20 = \mathsf{sync}(L11, L20) \otimes \mathsf{sync}(L19, L20) \tag{34}$$

where $L20$ adds up the delays from both threads by $\otimes$ in line with the multi-threading model of execution.

The equations (32)–(34) for fork and join are a surprisingly simple and compositional way of specifying timed concurrent structures. To illustrate let us revisit our sample simulation from Sec. 5 (see also Fig. 1b). The threads G and H arrive at their termination points with $L11 = 6{:}\bot{:}\bot{:}6{:}\underline{\bot}$ and $L19 = \bot{:}\bot{:}7{:}\underline{\bot}$, respectively. Thread G terminates in tick 1 and 4 while H finishes only in tick 3. The cost arising from synchronising G is $\mathsf{sync}(L11, L19) = 6{:}2{:}\bot{:}6{:}\underline{2}$ which is 6 at G's first termination time, then 2 while waiting for H, again 6 at the next re-entry in tick 4, when G terminates a second time. But since then H never terminates, the join stays active, generating cost 2 in each subsequent tick. On the other side we have $\mathsf{sync}(L19, L11) = \bot{:}\bot{:}7{:}\underline{\bot}$, which is the completion time for H. There are no extra cost as H does not need to wait for G. The output of the join has cost $L20 = \bot{:}\bot{:}9{:}\underline{\bot}$ which at termination in tick 3 combines the 7 ic cost from H plus 2 ic overhead for the join.

We are now nearly complete with our story. The equations tells us for each stimulation environment and control $v \in \mathbb{V}$ if and when v is reachable in any tick. The equations can be used for formal analysis, compiler verification, program transformations, timing-accurate simulation or even directly for implementation.

Here we are interested in obtaining the total WCRT of a program. When concurrency is present, the WCRT of a thread t is not the WCRT of any single control, but the WCRT of a set of controls. It is the worst case cost, over all ticks, of any set of controls that are potentially *concurrent* in t. A set of controls $\mathbb{C} \subseteq \mathbb{V}$

[6] In the KEP processor the join is executed at each tick until both threads have terminated, during which time it invokes some constant overhead cost.

is *concurrent*, written $conc(\mathbb{C})$, if all its elements belong to different child threads. For instance, $\{L11, L14\}$ is concurrent but $\{L6, L11\}$ is not. Concurrent controls execute in independent KEP hardware threads which are interleaved, whence their costs are added. In the search for such $\mathbb{C}$ we may restrict to the *completion controls* $cmpl(t)$ of a thread t. These are the controls in which t may terminate or pause. For instance, $cmpl(G) = \{L11\}$ and $cmpl(H) = \{in(v_9), in(v_{13}), L19\}$. For parent threads these must be included, i.e., we have $cmpl(T) = cmpl(G) \cup cmpl(H) \cup \{L20\}$. The control $L20$ describes the situations in which T terminates. The controls in $cmpl(G)$ are concurrent to those in $cmpl(H)$ and vice versa. None of them is concurrent with $L20$ which happens in their parent.

The worst case reaction time $\mathsf{wcrt}(t)$ of a synchronous program t the maximal sum of WCRT of any set of concurrent completion controls in any tick,

$$\mathsf{wcrt}(t) = max\{(\bigotimes_{v\in\mathbb{C}} v)[\mathbb{1}] \mid \mathbb{C} \subseteq cmpl(t), conc(\mathbb{C})\}, \tag{35}$$

where $(\bigotimes_{v\in\mathbb{C}} v)[\mathbb{1}] = max\{\bigotimes_{v\in\mathbb{C}} v(i) \mid i \geq 0\} = max\{\sum_{v\in\mathbb{C}} v(i) \mid i \geq 0\}$. Explicit solutions of (35) are non-trivial as it maximises over an infinite number of ticks i and choices of sets $\mathbb{C}$ whose number may grow exponentially with the number of threads. We do not know of any algorithm to solve (35) in its general form, yet solutions exist for special cases.

For normal clock-guarded synchronous programs the fps v are rational and thus can be represented as finite input-output tick cost automata, called IO-BTCA [24]. A given sum $\bigotimes_{v\in\mathbb{C}} v$ of controls can then be obtained by synchronous composition of automata. This is a well-understood construction, though it requires symbolic reasoning on boolean signals and is subject to the state-space explosion problem. The period (number of states) in the fps $v_1 \otimes v_2$ may be the product of the periods of v_1 and v_2. The automata-theoretic approach has been explored in [26] for timed concurrent control flow graphs TCCFGs (similar to CKAGs) using UPPAAL, but it does not scale well.

The situation is simpler for autonomous systems without input signals, which reduce to ultimately periodic sequences over $\mathbb{Z}_\infty$. Any IO-BTCAs can be over-approximated to an autonomous system, called *tick cost automaton* TCA, by eliminating signal dependencies, as discussed in Sec. 5.2, replacing each reference to a signal S or its negation $\neg S$ by $\underline{\top}$. Such approximations are sound but ignore inter-thread communication. The advantage is that the autonomous case of (35) can be translated into an (0/1) ILP. This *implicit path enumeration (IPE)* technique for WCRT analysis yields much better results [30] compared to the automata-theoretic approach.

The IPE approach has been considered the most efficient technique for autonomous approximations until recently, when explicit algebraic solutions for (35) have been attempted. In [24] it is observed that for the natural class of so-called *patient* TCA the computation of the normal form for each v is polynomial. This reduces the problem of computing the tick-wise additions $\bigotimes_{v\in\mathbb{C}} v$ for ultimately periodic sequences v to the *tick alignment problem* studied in [20, 24] which can be solved using graph-theoretic algorithms. This has led to significant speed-up

in the original ILP implementation of [30]. Still, even under signal abstraction, the theoretical complexity of computing the periodic normal form of a control $v \in \mathit{cmpl}(T)$ and solving the tick alignment problem remain open problems. Rather interestingly, recent experiments implementing the explicit fixed point construction mentioned in Sec. 5.2 indicate that for autonomous systems both problems may be polynomial in practice [29], despite the theoretical exponential blow-up.

The fastest polynomial algorithm to date for solving (35), unsurprisingly, is also the most over-approximating one. The dynamic programming approach of [4] not only abstracts from signals but also from state dependencies, as explained in Sec. 5.2. It bundles all state controls σ_i of a given program block t into a single pair $\mathit{out}(t) = \oplus_i \mathit{out}(\sigma_i)$, $\mathit{in}(t) = \oplus_i \mathit{in}(\sigma_i)$. The system equation of t then becomes $\big(\xi\ \mathit{in}(t)\big)^{\mathsf{T}} = D_t \otimes \big(\zeta\ \mathit{out}(t)\big)^{\mathsf{T}}$ where D_t is a matrix of scalar constants. With the register equation $\mathit{out}(t) = X \odot \mathit{tick}(\mathit{in}(t))$ for the feedback, the closed solution is attainable in a single fixed point iteration, in $\mathcal{O}(1)$ time. Moreover, the fps for each control v is of the form $d_0{:}\underline{d_1}$ a delay for the initial tick and all subsequent ones being identical. Hence the calculation of $\bigotimes_{v\in\mathbb{C}} v$ is done in $\mathcal{O}(1)$ time, too. Moreover, the fact that each control has only two entries $v = v(0){:}\underline{v(1)}$ helps greatly in the maximisation over all $\mathbb{C}$: For each given i the tick-wise maximum $\mathsf{wcrt}_i(t) = \mathit{max}\{\bigotimes_{v\in\mathbb{C}} v(i) \mid \mathbb{C} \subseteq \mathit{cmpl}(t), \mathit{conc}(\mathbb{C})\}$ can be obtained bottom-up by induction over the thread hierarchy. The reason is that in the maximum $\mathsf{wcrt}_i(t) = \bigotimes_{v\in\mathbb{C}_{max}} v(i)$ the constituent controls $\mathbb{C}' = \mathbb{C}_{max} \cap \mathit{cmpl}(t')$ for each child t' of t are not only concurrent $\mathit{conc}(\mathbb{C}')$, but necessarily constitute the tick-specific maximum $\mathsf{wcrt}_i(t') = \bigotimes_{v\in\mathbb{C}'} v(i)$ for the child, too.

6 Related Work and Conclusions

A rudimentary version of the WCRT interface model has been proposed originally in [23]. That work focused on the algorithmic aspects of the modular timing analysis of synchronous programs. It was implemented in the backend of a compiler for Esterel, analysing reactive assembly code running on the Kiel Esterel Processor (KEP). A rigorous mathematical definition of the behavioural semantics of the interface models was presented in [22]. The axiomatic approach of [22] highlighted the essentially logical nature of the WCRT interfaces. It was shown how the logical interface language can specify, and relate with each other, standard analysis problems such as shortest path, task scheduling or max-flow problems. However, the logical theory developed by [23] and [22] was still restricted to the modelling of the purely combinational behaviour of a synchronous module, i.e., its reactive behaviour during a single tick. This yields the worst-case timing over *all* states rather than just the *reachable* ones. In general, this is an over-approximation of the exact WCRT. The tick dependency of WCRT behaviour, also called *tick alignment*, was subsequently studied in [24]. It was observed that the combinational timing of single ticks can be modelled in max-min-plus algebra, which is the intuitionistic algebra of SGD. This makes it possible to

express the timing behaviour of a synchronous module over arbitrary sequences of clock ticks as formal power series. The composition of synchronous systems arises from the lifting of SGD algebra to formal power series. The paper [24] investigates the tick alignment of timing in its pure form, i.e, without signal communication between concurrent synchronous threads. This induces a form of data abstraction which reduces the WCRT analysis to the maximum weighted clique problem on tick aligment graphs. It is shown in [24] how this reduction permits a considerable speed-up of an existing ILP algorithm that was proposed earlier. By exploiting the logical expressiveness of SGD algebra, formal power series can handle not only tick-dependent timing but also signal communication. This is applied in [29, 1] to obtain the full behavioural semantics of timed and concurrent synchronous control flow graphs in a structural fashion.

In this paper we revisit this earlier work on WCRT interface algebra and in doing so combine, for the first time, the algebraic semantics of [24, 29, 1] with the logical setting of [23, 22]. This is the first timing-enriched and causality-sensitive semantics of SCP which is modular and covers full tick behaviour. The SGD[X] equations constitute a cycle-accurate model and can be used for program analysis and verification. This can also be used to compile Esterel via CKAG control-flow graphs directly into data flow format. In future work it will be interesting to explore the possibility of generating hardware circuits and compare with existing hardware compilation chains for Esterel. On the theoretical side we plan to study algebraic axiomatisation for SGD[X] and its expressiveness, specifically its relationship with ILP.

Note that in this paper we apply SGD[X] to intermediate-level control-flow code which is assumed to be self-synchronised, i.e., free of causality issues. The code is a scheduled version of a high-level Esterel source program which has been checked for causality. If this cannot be assumed, a more complex "dual-rail" algebraic model is needed to capture the constructive semantics of Esterel. Each signal $E = (E^+, E^-)$ splits into two SGD[X] series for the positive and negative signal status separately. Constructive presence of E is $E^+ > \bot$ while $E^- = \bot$ whereas constructive absence is $E^+ = \bot$ and $E^- > \bot$. The value $E = (\bot, \bot)$ expresses that the status of E is unknown. Each logical operation is an operation on both rails, c.f. the ternary model of [21]). E.g., the negation $\neg$ in the ternary model is $\neg(A^+, A^-) = (A^-, A^+)$ and delay is $d \odot (A^+, A^-) = (d \odot A^+, d \odot A^-)$. For instance, consider an arbitration situation as a classic example of non-constructiveness: One thread emits signal A on absence of a signal B and another thread emits B on absence of A. If both emissions are with delay $d > 0$, this induces the equations $(A^+, A^-) = (d \odot B^-, d \odot B^+)$ and $(B^+, B^-) = (d \odot A^-, d \odot A^+)$ which imply $A^+ = 2d \odot A^+$ and $A^- = 2d \odot A^-$. This system has no bounded solution.

Dedication

The first author is indebted to Bernhard Steffen for his long continued guidance and encouragement both as a friend and mentor. My first training in academic writing was as a co-author of an article with Bernhard in 1989, when we were

both with the LFCS at Edinburgh University. Some years and many joint rounds of golf later, I spent a most enjoyable time as a member of the inspiring research environment which Bernhard had created at Passau University. Bernhard's support was not only instrumental for my successful habilitation at Passau. He also ensured, at the right moment, that I would feel pressured to find a secure permanent research job, rather than clinging to yet another limited term position. Talking research, is was him who suggested me to look to synchronous programming as an application for my work on concurrency theory and constructive logic. In this way, the work reported here originally started with a far-sighted vision of Bernhard's.

Acknowledgements

This work was supported by the German Research Council DFG under grant ME-1427/6-2 (PRETSY2).

References

1. Aguado, J., Mendler, M., Wang, J.J., Bodin, B., Roop, P.: Compositional timing-aware semantics for synchronous programming. In: Forum on Specification & Design Languages (FDL 2017). pp. 1–8. IEEE, Verona (September 2017)
2. Baccelli, F.L., Cohen, G., Olsder, G.J., Quadrat, J.P.: Synchronisation and Linearity. John Wiley & Sons (1992)
3. Berry, G., Cosserat, L.: The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In: Seminar on Concurrency, Carnegie-Mellon University. pp. 389–448. Springer LNCS 197 (1984)
4. Boldt, M., Traulsen, C., von Hanxleden, R.: Compilation and worst-case reaction time analysis for multithreaded Esterel processing. EURASIP Journal on Embedded Systems **2008**(1) (Apr 2008)
5. van Dalen, D.: Intuitionistic logic. In: Gabbay, D., Guenthner, F. (eds.) Handbook of Philosophical Logic, vol. III, chap. 4, pp. 225–339. Reidel (1986)
6. Dummett, M.: A propositional calculus with a denumerable matrix. Journal of Symbolic Logic **24**, 97–106 (1959)
7. Edwards, S.A., Lee, E.A.: The case for the precision timed (PRET) machine. In: DAC 2007. San Diego (USA) (June 2007)
8. Edwards, S.A., Kim, S., Lee, E.A., Liu, I., Patel, H.D., Schoeberl, M.: A disruptive computer design idea: Architectures with repeatable timing. In: Proc. of IEEE International Conference on Computer Design (ICCD 2009). IEEE (October 2009)
9. Fermüller, C.G.: Parallel dialogue games and hypersequents for intermediate logics. In: Maier, M.C., Pirri, F. (eds.) TABLEAUX 2003. LNCS 2796, Springer (2003)
10. Geilen, M., Stuijk, S.: Worst-case performance analysis of synchronous dataflow networks. In: CODES+ISSS'10. ACM, Scottsdale, Arizona, USA (October 2010)
11. Hájek, P.: Metamathematics of Fuzzy Logic. Kluwer (1998)
12. von Hanxleden, R., Li, X., Roop, P., Salcic, Z., Yoong, L.H.: Reactive processing for reactive systems. ERCIM News **66**, 28–29 (Oct 2006)
13. von Hanxleden, R., Mendler, M., Traulsen, C.: WCRT algebra and scheduling interfaces for Esterel-style synchronous multi-threading. Tech. Rep. 0807, Christian-Albrechts-Univ. Kiel, Dept. of Comp. Sci. (June 2008), `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-0807.pdf`

14. Hirai, Y.: A lambda calculus for Gödel-Dummett logic capturing waitfreedom. In: Schrijvers, T., Thiemann, P. (eds.) Proc. FLOPS 2012. pp. 151–165. LNCS 7294, Springer (2012)
15. Ju, L., Huynh, B.K., Chakraborty, S., Roychoudhury, A.: Context-sensitive timing analysis of Esterel programs. In: Proc. 46th Annual Design Automation Conference (DAC 2009). pp. 870–873. ACM, New York, NY, USA (2009)
16. Ju, L., Huynh, B.K., Roychoudhury, A., Chakraborty, S.: Performance debugging of Esterel specifications. Real-Time Systems **48**(5), 570–600 (2012)
17. Kuo, M., Sinha, R., Roop, P.S.: Efficient WCRT analysis of synchronous programs using reachability. In: Proc. 48th Design Automation Conference (DAC 2011). pp. 480–485 (2011)
18. Li, X., von Hanxleden, R.: Multi-threaded reactive programming—the Kiel Esterel Processor. IEEE Transactions on Computers **61**(3), 337–349 (Mar 2012)
19. Lickly, B., Liu, I., Kim, S., Patel, H.D., Edwards, S.A., Lee, E.A.: Predictable programming on a precision timed architecture. In: Proc. Conf. Compilers, Architectures, and Synthesis of Embedded Systems (CASES'08). pp. 137–146. Atlanta (USA) (October 2008)
20. Mendler, M., Bodin, B., Roop, P., Wang, J.J.: WCRT for synchronous programs: Studying the tick alignment problem. Tech. Rep. 95, University of Bamberg, Faculty for Information Systems and Applied Computer Sciences (August 2014)
21. Mendler, M., Shiple, T., Berry, G.: Constructive boolean circuits and the exactness of timed ternary simulation. Formal Methods in System Design **40**(3), 283–329 (2012)
22. Mendler, M.: An algebra of synchronous scheduling interfaces. In: Legay, A., Caillaud, B. (eds.) Proc. Foundations for Interface Technologies (FIT 2010). EPTCS, vol. 46, pp. 28–48. Paris, France (2010)
23. Mendler, M., von Hanxleden, R., Traulsen, C.: WCRT Algebra and Interfaces for Esterel-Style Synchronous Processing. In: Proc. Design, Automation and Test in Europe Conference (DATE 2009). Nice, France (Apr 2009)
24. Mendler, M., Roop, P.S., Bodin, B.: A novel WCET semantics of synchronous programs. In: In: Formal Modeling and Analysis of Timed Systems (FORMATS 2016). pp. 195–210. Quebec, QC, Canada (August 2016)
25. Raymond, P., Maiza, C., Parent-Vigouroux, C., Carrier, F., Asavoae, M.: Timing analysis enhancement for synchronous programs. Real-Time Systems **51**, 192–220 (2015)
26. Roop, P.S., Andalam, S., von Hanxleden, R., Yuan, S., Traulsen, C.: Tight WCRT analysis of synchronous C programs. Proc. Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2009) pp. 205–214 (2009)
27. Schoeberl, M.: Time-predictable computer architecture. EURASIP Journal on Embedded Systems **2009**, 2:1–2:17 (2009)
28. Waez, M.T.B., Dingel, J., Rudie, K.: A survey of timed automata for the development of real-time systems. Computer Science Review **9**, 1–26 (2013)
29. Wang, J., Mendler, M., Roop, P., Bodin, B.: Timing analysis of synchronous programs using WCRT algebra: Scalability through abstraction. ACM TECS **16**(5s), 177:1–177:19 (2017)
30. Wang, J.J., Roop, P.S., Andalam, S.: ILPc : A novel approach for scalable timing analysis of synchronous programs. In: CASES'13. pp. 20:1–20:10. Montreal, Canada (Sept–Oct 2013)
31. Yip, E., Roop, P.S., Biglari-Abhari, M., Girault, A.: Programming and timing analysis of parallel programs on multicores. In: Proc. Application of Concurrency to System Design (ACSD 2013). pp. 160–169. IEEE (2013)