An Algebra of Synchronous Scheduling Interfaces

Michael Mendler
Faculty of Information Systems and Applied Computer Sciences
Bamberg University
michael.mendler@uni-bamberg.de

In this paper we propose an algebra of synchronous scheduling interfaces which combines the expressiveness of Boolean algebra for logical and functional behaviour with the min-max-plus arithmetic for quantifying the non-functional aspects of synchronous interfaces. The interface theory arises from a realisability interpretation of intuitionistic modal logic (also known as Curry-Howard-Isomorphism or propositions-as-types principle). The resulting algebra of interface types aims to provide a general setting for specifying type-directed and compositional analyses of worst-case scheduling bounds. It covers synchronous control flow under concurrent, multi-processing or multi-threading execution and permits precise statements about exactness and coverage of the analyses supporting a variety of abstractions. The paper illustrates the expressiveness of the algebra by way of some examples taken from network flow problems, shortest-path, task scheduling and worst-case reaction times in synchronous programming.

1 Introduction

The algebra discussed in this paper aims at the specification of behavioural interfaces under the execution model of synchronous programming. Such interfaces abstract externally observable Boolean controls for components activated under the regime of a global synchronous scheduler familiar from data-flow oriented languages such as Lustre [11], Signal [8], Lucid Synchrone [24], or imperative control-flow oriented languages such as Statecharts [12, 23], Esterel [5] and Quartz [25]. In this model computations are coordinated under one or more global system clocks, which may be physical or logical. They divide physical time into a sequence of discrete *ticks*, or *instants*. During each instant the synchronous components interact using *broadcast signals*, which can have one of two statuses, *present* or *absent*. These signal statuses evolve monotonically as they are propagated through the system, generating the emission or inhibition of further signals and computations. Under the *synchrony hypothesis* [10] it is assumed that at each instant, outputs are synchronous with the inputs. In other words, computations take place instantaneously and appear to happen at each tick "all at once."

The synchrony hypothesis conveniently abstracts internal, possibly distributed computations into atomic reactions, making signals appear almost like Boolean variables and (stateful) interfaces almost like Mealy automata with Boolean labels. Unfortunately, this abstraction is not perfect, so that Boolean algebra is insufficient. First, it is well-known [14, 20] that classical two-valued Boolean analysis is inadequate to handle the causality and compositionality problems associated with the synchrony hypothesis adequately. E.g., Boolean algebra by itself cannot guarantee there are no races between signal presence and absence, thus guaranteeing unique convergence after a finite number of signal propagation steps. Some form of causality information needs to be preserved. Secondly, quite practically, in many applications we want to compute non-Boolean information about otherwise "instantaneous" control signals, such as latency or worst-case reaction times, maximal throughput, earliest deadlines, or other quantitative information about the scheduling process. This provides one way to motivate the work reported

Benoît Caillaud and Axel Legay (Eds.):
Foundations of Interface Technologies (FIT'10)
EPTCS 46, 2011, pp. 28–48, doi:10.4204/EPTCS.46.3

here, viz. the search for a fully abstract synchronisation algebra as an economic refinement of classical Boolean algebra in situations where Booleans are subject to synchronous schedules and quantitative resource consumption.

Another motivation may be drawn from the arithmetical point of view. One of the challenges in quantitative resource analysis is the clever interchange (distribution) of *max*, *min* and $+$. For instance, consider the analysis of worst-case reaction times (WCRT). In its simplest form, given a weighted dependency graph, the WCRT is the maximum of all sums of paths delays, an expression of the form $max(\sum_{i\in p_1} d_{i1}, \sum_{i\in p_2} d_{i2}, \ldots, \sum_{i\in p_n} d_{in})$ where p_j are execution paths of the system and d_{ij} the delay of path segment i in path p_j. As it happens, 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-maxes* (dynamic programming on dependency graphs) employing various forms of dependency abstraction. For illustration, imagine two alternative path segments of length d_1, e_1 sequentially followed by two alternative path segments of length d_2, e_2, respectively. The distribution $max(d_1+d_2, d_1+e_2, e_1+d_2, e_1+e_2) = max(d_1,e_1)+max(d_2,e_2)$ for efficiently calculating the longest possible path, 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 sum-of-maxes obtains but conservative over-approximations. E.g., assume the combination of d_1 with e_2 is infeasible. Then, the sum-of-maxes is not exact since $max(d_1,e_1)+max(d_2,e_2) \geq max(d_1+d_2, e_1+d_2, e_1+e_2)$. On the other hand, knowing the infeasibility of d_1+e_2 we would rather compute $max(d_1+d_2, e_1+max(d_2,e_2)) = max(d_1+d_2, e_1+d_2, e_1+e_2)$ which eliminates one addition and thus is both exact *and* more efficient than the full conservative max-of-sums. The same applies to min-plus problems such as shortest path or network flow. In the former, the efficient *sum-of-mins* is an under-approximation of the exact *min-of-sums* on all feasible paths. For network flow the arithmetic is complicated further by the fact that min/max do not distribute over $+$, i.e., $min(d, e_1+e_2) \neq min(d,e_1)+min(d,e_2)$ which obstructs simple linear programming techniques.

The art of scheduling analysis consists in finding a judicious trade-off between merging paths early in order to aggregate data on the one hand, and refining dependency paths by case analysis for the sake of exactness, on the other hand. A scheduling algebra for practicable algorithms must be able to express and control this trade-off. In this paper we present an interface theory which achieves this by coupling resource weights d with logic formulas ϕ. A pair $d:\phi$ specifies the semantic meaning of d within the control-flow of a program module. Logical operations on the formulas then go hand-in-hand with arithmetic operations on resources. E.g., suppose a schedule activates control points X and Y with a cost of d_1 and d_2, respectively, expressed $d_1 : \circ X \wedge d_2 : \circ Y$. If the threads are resource concurrent then both controls are jointly active within the maximum, i.e., $max(d_1,d_2) : \circ(X \wedge Y)$. If we are only concerned whether one of the controls is reached, then we take the minimum $min(d_1,d_2) : \circ(X \oplus Y)$. If activations of X and Y requires interleaving of resources, then we must use addition $d_1+d_2 : \circ(X \otimes Y)$.

Our interface theory combines min-max-plus algebra $(\mathbb{N}_\infty, min, max, +, 0, -\infty, +\infty)$, see e.g. [4], with a refinement of Boolean algebra to reason about logical control-flow. It features two conjunctions $\wedge$, $\otimes$ to distinguish concurrent from multi-threading parallelism, two disjunctions $\vee$, $\oplus$ to separate external from internal scheduling choices, respectively. A consequence of its constructive nature, our algebra replaces classical negation by a weaker and more expressive *pseudo-complement* for which $\overline{\overline{x}} = x$ and $x+\overline{x} = 1$ are no longer tautologies. This turns Boolean into a so-called Heyting algebra. The work presented here is an extension and adaptation of our earlier work on *propositional stabilisation theory* [21] which has been developed to provide a semantic foundation for combinational timing analyses.

The plan for the paper is as follows: To start with, Sec. 2 lays out the syntactic and semantical groundwork for our interface type theory which is then studied in some more detail in Sec. 3. For compactness we keep these theoretical Sections 2 and 3 fairly condensed, postponing examples to Secs. 4 and 5. In

the former, Sec. 4, we sketch applications to network flow, shortest path and task scheduling, while in Sec. 5 we discuss the problem of WCRT analysis for Esterel-style synchronous processing. The paper concludes in Sec. 6 with a discussion of related work.

2 Syntax and Semantics of Synchronous Scheduling Interfaces

Synchronous scheduling assumes that all dependencies in the control flow of a single instant are acyclic and the propagation of control, for all threads, is a monotonic process in which each atomic control point is only ever activated at most once. Let $\mathbb{V}$ be a set of *signals*, or *control variables*, which specify the atomic control points in the interface of a synchronous module. An *event* is a subset $E \subseteq \mathbb{V}$ of control variables. A *synchronous activation sequence*, or simply an *activation*, is a monotonically increasing function $\sigma \in \underline{n} \to 2^{\mathbb{V}}$ from $\underline{n} = \{0,1,\ldots,n-1\}$ into the set of events, i.e., $\sigma(i) \subseteq \sigma(j)$ for all $0 \leq i \leq j < n$. The *length* $|\sigma|$ of σ is the number of events it contains, i.e., $|\sigma| = \underline{n}$. The unique activation of length $\underline{0} = \emptyset$ is called the *empty* activation, also denoted $\emptyset$.

Activations model the monotonic process of signal propagation during one synchronous instant, i.e., between two ticks of the logical clock. They induce a Boolean valuation on the control variables in the sense that $A \in \mathbb{V}$ may be considered "present" for the instant if $A \in \sigma(i)$ for some $0 \leq i < |\sigma|$ and "absent" otherwise. In the former case, index i is the *activation level* for the presence of control A. In general, the domain $\underline{n}$ over which an activation is defined acts as a discrete domain of quantifiable resources which are consumed by control variables becoming active at different resource levels. In this way, activation sequences give an operational understanding of truth values that is faithful to causality and resource consumption. A canonical interpretation is the temporal reading: The length $|\sigma|$ is the duration of the synchronous instant, i.e., the overall reaction time, and $A \in \sigma(i)$ means that A is activated, or is present from micro-step i.

Definition 2.1 *Let $\sigma \in \underline{n} \to 2^{\mathbb{V}}$ be an activation.*

- *A* sub-activation *$\sigma' \subseteq \sigma$ of σ is an activation $\sigma' \in \underline{m} \to 2^{\mathbb{V}}$ such that there exists a strictly monotonic function $f \in \underline{m} \to \underline{n}$ with $\sigma'(i) = \sigma(f(i))$ for all $i \in \underline{m}$.*
- *We write $\sigma = \sigma_1 \cup \sigma_2$ to express that sub-activations $\sigma_1, \sigma_2 \subseteq \sigma$ form an* activation cover *of σ, or an* interleaving decomposition *in the sense that each event is contained in σ_1 or in σ_2, i.e., $\forall i \in |\sigma|.\, \exists j = 1,2.\, \exists k \in |\sigma_j|.\, i = f_j(k)$ where f_j are the index embeddings of σ_j, $j = 1,2$.*
- *For every $i \in \mathbb{N}$ we define the* shifted *activation $\sigma[i,:] : \underline{m} \to 2^{\mathbb{V}}$, where $m =_{df} \{j \mid 0 \leq j+i < n\}$ and $\sigma[i,:](j) =_{df} \sigma(j+i)$.*

A shifted activation is also a sub-activation, $\sigma[i,:] \subseteq \sigma$. We have $\sigma[i,:] = \emptyset$ if $\sigma = \emptyset$ or if $i \geq |\sigma|$. The shift operator is monotonic wrt sub-activations and antitonic wrt resource level, i.e., if $\sigma' \subseteq \sigma$ and $0 \leq i \leq j$ then $\sigma'[j,:] \subseteq \sigma[i,:]$. This depends on strict monotonicity of the index embedding in $\sigma' \subseteq \sigma$.

In order to model non-determinism (abstracting from internal parameters or external environment) our interfaces are interpreted over subsets Σ of activation sequences, called *(synchronous) schedules*. These schedules (of a program, a module, or any other program fragment) will be specified by a *scheduling type* ϕ generated by the logical operators

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

generated from control variables $A \in \mathbb{V}$. We will write $\Sigma \models \phi$ ($\sigma \models \phi$) to say that schedule Σ (activation σ) *satisfies* the type ϕ. The semantics of types is formally defined below in Def. 2.2. As a type specification,

each control variable $A \in \mathbb{V}$ represents the guarantee that "*A is active (the signal is present, the program label has been traversed, the state is activated) in all activations of* Σ". The constant *true* is satisfied by all schedules and *false* only by the empty schedule or the schedule which contains only the empty activation. The type operators $\neg$, $\supset$ are negation and implication. The operators $\vee$ and $\oplus$ are two forms of logical disjunction to encode internal and external non-determinism and $\wedge$, $\otimes$ are two forms of logical conjunction related to true concurrency and interleaving concurrency, respectively. Finally, $\circ$ is the operator to express resource consumption. The usual bracketing conventions apply: The unary operators $\neg$, $\circ$ have highest binding power, implication $\supset$ binds most weakly and the multiplicatives $\wedge$, $\otimes$ are stronger than the summations $\vee$, $\oplus$. Occasionally, bi-implication $\phi \equiv \psi$ is useful as an abbreviation for $(\phi \supset \psi) \wedge (\psi \supset \phi)$. Also, we note that $\neg\phi$ is equivalent to $\phi \supset \mathit{false}$.

A scheduling type ϕ by itself only captures the functional aspect of an interface. To get a full interface we need to enrich types by resource information. To this end, we associate with every scheduling type ϕ a set of *scheduling bounds* $\mathit{Bnd}(\phi)$ recursively as follows:

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

where $\underline{1} = \{0\}$ is a distinguished singleton set. Elements of the disjoint sum $\mathit{Bnd}(\phi) + \mathit{Bnd}(\psi)$ are presented as pairs $(0, f)$ where $f \in \mathit{Bnd}(\phi)$ or $(1, g)$ where $g \in \mathit{Bnd}(\psi)$. The set $\mathit{Bnd}(\phi) \times \mathit{Bnd}(\psi)$ is the Cartesian product of the sets $\mathit{Bnd}(\phi)$ and $\mathit{Bnd}(\psi)$ and $\mathit{Bnd}(\phi) \to \mathit{Bnd}(\psi)$ the set of total functions from $\mathit{Bnd}(\phi)$ to $\mathit{Bnd}(\psi)$. Intuitively, an element $f \in \mathit{Bnd}(\phi)$ may be seen as a form of generalised higher-order resource matrix for schedules of shape ϕ.

Definition 2.2 *A* scheduling interface *is a pair $f : \phi$ consisting of a scheduling type ϕ and a scheduling bound $f \in \mathit{Bnd}(\phi)$. An activation σ* satisfies *an interface $f : \phi$, or* satisfies *the scheduling type ϕ with bound f, written $\sigma \models f : \phi$, according to the following inductive rules:*

$$\begin{array}{lcll}
\sigma & \models & 0 : \mathit{false} & \mathit{iff} \quad |\sigma| = 0, \text{ i.e., } \sigma = \emptyset\\
\sigma & \models & 0 : \mathit{true} & \mathit{iff} \quad \mathit{always}\\
\sigma & \models & 0 : A & \mathit{iff} \quad \forall 0 \le i < |\sigma| \Rightarrow A \in \sigma(i)\\
\sigma & \models & (f,g) : \phi \wedge \psi & \mathit{iff} \quad \sigma \models f : \phi \text{ and } \sigma \models g : \psi\\
\sigma & \models & (0,f) : \phi \vee \psi & \mathit{iff} \quad \sigma \models f : \phi\\
\sigma & \models & (1,g) : \phi \vee \psi & \mathit{iff} \quad \sigma \models g : \psi\\
\sigma & \models & (f,g) : \phi \oplus \psi & \mathit{iff} \quad \sigma \models f : \phi \text{ or } \sigma \models g : \psi\\
\sigma & \models & f : \phi \supset \psi & \mathit{iff} \quad \forall \sigma' \subseteq \sigma.\, \forall g \in \mathit{Bnd}(\phi).\, (\sigma' \models g : \phi \Rightarrow \sigma' \models f\, g : \psi)\\
\sigma & \models & (d,f) : \circ\phi & \mathit{iff} \quad |\sigma| = 0 \text{ or } \exists i \in \mathbb{N}.\, 0 \le i \le d \text{ and } \sigma[i,:] \models f : \phi\\
\sigma & \models & (f,g) : \phi \otimes \psi & \mathit{iff} \quad \exists \sigma_1, \sigma_2 \subseteq \sigma.\, \sigma = \sigma_1 \cup \sigma_2 \text{ and } \sigma_1 \models f : \phi \text{ and } \sigma_2 \models g : \psi.
\end{array}$$

A schedule Σ satisfies ϕ with *bound f, written $\Sigma \models f : \phi$, if for all $\sigma \in \Sigma$, $\sigma \models f : \phi$. A schedule* satisfies *ϕ or is* bounded *for ϕ if there exists $f \in \mathit{Bnd}(\phi)$ such that $\Sigma \models f : \phi$.*

The semantics $\Sigma \models f : \phi$ as formalised in Def. 2.2 is a ternary relation: It links schedules, types and bounds. The symbol $\models$ separates the behavioural model Σ from the formal interface $f : \phi$. The latter, in turn, combines a qualitative and a quantitative aspect. The type ϕ captures the causal relationships between the control points and the bound $f \in \mathit{Bnd}(\phi)$ refines this quantitatively by weaving in concrete activation levels. The colon : is a binary connective which separates these concerns.

Proposition 2.3 *$\sigma \models f : \phi$ and $\sigma' \subseteq \sigma$ implies $\sigma' \models f : \phi$. Moreover, $|\sigma| = 0$ implies $\sigma \models f : \phi$.*

Prop. 2.3 says that interfaces are inherited by sub-activations. This is natural since a sub-activation selects a subset of events and thus (in general) contains more control variables with lower activation distances. The degenerated case is the empty activation which is inconsistent and thus satisfies all interfaces, including the strongest specification $0 : \mathit{false}$, viz. "*everything is true with zero resource consumption*".

The most general way to use the semantic relation of Def. 2.2 is to consider the set of (typically abstracted) activations for a given module P as a schedule Σ_P, and then determine a suitable interface for it. Any such $f : \phi$ with $\Sigma_P \models f : \phi$ may be taken as a valid interface specification of P giving a quantified behavioural guarantee for all activations $\sigma \in \Sigma_P$ under the given scheduling assumptions. Ideally, we are interested in the best fitting or *tightest* interface, if such exists. To measure the relative strength of an interface we employ Def. 2.2 to associate with every pair $f : \phi$ the schedule $[\![f : \phi]\!] = \{\, \sigma \mid \sigma \models f : \phi \,\}$ which is the semantic meaning of the interface. Interfaces may then be compared naturally. The smaller the set of associated activations $[\![f : \phi]\!]$ the tighter is the interface $f : \phi$. Formally, we write

$$f : \phi \preceq g : \psi \qquad \text{if} \qquad [\![f : \phi]\!] \subseteq [\![g : \psi]\!]$$

and $f : \phi \cong g : \psi$ in case $[\![f : \phi]\!] = [\![g : \psi]\!]$. We call an interface $f : \phi$ *tight* for Σ_P if it is minimal wrt $\preceq$, i.e., whenever $g : \psi \preceq f : \phi$ and $\Sigma_P \models g : \psi$ then $f : \phi \cong g : \psi$. A tight interface provides exact information about Σ_P in both the functional and the resource dimensions within the expressiveness of our typing language. Typically, however, we are given some schedule Σ_P together with a *fixed* type ϕ and ask for a minimal bound f such that $\Sigma_P \models f : \phi$. If such a tight bound exists and is unique we call it *worst-case for ϕ*.

We generalise equivalence to arbitrary types, taking $\phi \cong \psi$ to mean that for every $f \in \mathit{Bnd}(\phi)$ there is $g \in \mathit{Bnd}(\psi)$ such that $f : \phi \cong g : \psi$ and vice versa, for each $g \in \mathit{Bnd}(\psi)$ we can find $f \in \mathit{Bnd}(\phi)$ with $g : \psi \cong f : \phi$. The main purpose of the relations $\preceq$ and $\cong$ is to justify strengthening, weakening or semantics-preserving, transformations to handle interfaces as tightly as sensible. They are the basis of the interface algebra, some of whose laws will be studied next.

3 The Algebra of Scheduling Types

The set of scheduling bounds $\mathit{Bnd}(\phi)$ captures the amount of resource information associated with a type ϕ. In this respect the most simple class of types is that for which $\mathit{Bnd}(\phi)$ is (order) isomorphic to $\underline{1}$. Such types are called *pure* since they do not carry resource information and thus specify only functional behaviour. It will be convenient to exploit the isomorphisms $\mathit{Bnd}(\zeta) \cong \underline{1}$ and identify all bounds $f \in \mathit{Bnd}(\zeta)$ of a pure type canonically with the unique $0 \in \underline{1}$. Further, since it is unique, we may as well drop the (non-informative) bound and simply write ζ instead of $0 : \zeta$. This means, e.g., that $\zeta_1 \wedge \zeta_2$, $(0,0) : \zeta_1 \wedge \zeta_2$ and $0 : \zeta_1 \wedge \zeta_2$ are all identified.

Second, with this simplification on pure types in place, we may mix bounds and types and apply the type operators to full interfaces. Since $f : \phi$ specifies individual activations it formally behaves like an atomic statement. Hence, it is possible to use interfaces $f : \phi$ themselves as generalised "control variables" in types such as $(f : \phi) \wedge \psi$ or $\circ(f : \phi)$. We simply define

$$\mathit{Bnd}(f : \phi) =_{df} \underline{1} \qquad \sigma \models 0 : (f : \phi) \text{ iff } \sigma \models f : \phi$$

which turns an interface $f : \phi$ into a pure type. Then, e.g., $[\![f : \phi \wedge g : \psi]\!] = [\![(0,0) : (f : \phi \wedge g : \psi)]\!] = [\![0 : (f : \phi)]\!] \cap [\![0 : (g : \psi)]\!] = [\![f : \phi]\!] \cap [\![g : \psi]\!]$.

A few basic facts about the interface algebra arising from Def. 2.2 are readily derived. Not really surprisingly, *true* and *false* are complements, $\neg true \cong false$, $\neg false \cong true$ as well as neutral $false \otimes \phi \cong false \oplus \phi \cong true \wedge \phi \cong \phi$ and dominant elements $false \wedge \phi \cong false$, $true \oplus \phi \cong true \vee \phi \cong true \otimes \phi \cong true$. Shifting a type by $-\infty$ and $+\infty$ produces the strongest and weakest statements *false* and *true*, respectively:

Proposition 3.1 *For arbitrary types* ϕ, $-\infty : \circ\phi \cong false$ *and* $+\infty : \circ\phi \cong true$.

All operators $\vee$, $\wedge$, $\oplus$ and $\otimes$ are commutative. The pairs $\vee \leftrightarrow \wedge$ and $\oplus \leftrightarrow \wedge$ fully distribute over each other, while $\otimes$ distributes over both $\oplus$ and $\vee$, but not the other way round. Between $\otimes$ and $\wedge$ no distribution is possible, in general. One can show that the fragment $\vee$, $\wedge$, *false*, $\neg$, $\supset$ satisfies the laws of Heyting algebras seen in Prop. 3.2.

Proposition 3.2 *For arbitrary types* ϕ_1, ϕ_2, ψ*:*

$$\begin{array}{ll}
\psi \supset \psi \cong true & \phi_1 \supset (\phi_2 \supset \phi_1) \cong true \\
(\phi_1 \wedge \phi_2) \supset \psi \cong \phi_1 \supset (\phi_2 \supset \psi) & (\phi_1 \supset \phi_2) \wedge \phi_1 \cong \phi_1 \wedge \phi_2 \\
(\phi_1 \vee \phi_2) \supset \psi \cong (\phi_1 \supset \psi) \wedge (\phi_2 \supset \psi) & \psi \supset (\phi_1 \wedge \phi_2) \cong (\psi \supset \phi_1) \wedge (\psi \supset \phi_2) \\
false \supset \psi \cong true & \psi \supset true \cong true \\
\psi \supset false \cong \neg\psi & true \supset \psi \cong \psi.
\end{array}$$

It is worthwhile to observe that the classical principles of the Excluded Middle $A \oplus \neg A$ and $A \vee \neg A$ are both different and not universally valid in WCRT algebra. The latter says A is *static*, i.e., A is present in all activations or absent in all activations, the former that signal A is *stable*, i.e., in each activation individually, A is either present from the start or never becomes active. Clearly, not every signal is static or stable. The absence of the axioms $A \oplus \neg A$, $A \vee \neg A$, which arises naturally from the activation semantics, is a definitive characteristics of intuitionistic logic or Heyting algebra. This feature is crucial to handle the semantics of synchronous languages in a compositional and fully abstract way [20].

Boolean Types. An important sub–class of pure types are negated types $\neg\phi$. They express universal statements about each singleton event of each activation sequence in a schedule. For instance, $\Sigma \models \neg(A \otimes B)$ says that no event $\sigma(i) \subseteq \mathbb{V}$ $(0 \leq i < |\sigma|)$ in any $\sigma \in \Sigma$ contains A or B. Similarly, $\neg(A \supset B)$ states that A is present and B is absent in every event of every activation sequence, which is the same as $\neg\neg(A \wedge \neg B)$. Negated types are expressively equivalent to, and can be transformed into, *Boolean* types obtained from the following grammar, where ϕ is an arbitrary type:

$$\beta \quad ::= \quad true \mid false \mid A \mid \neg\beta \mid \beta \wedge \beta \mid \beta \otimes \beta \mid \phi \supset \beta.$$

Proposition 3.3 *The Boolean types form a Boolean algebra with* $\neg$, $\wedge$, $\otimes$ *as classical complement, conjunction and disjunction, respectively. Moreover,* $\Sigma \models \beta$ *iff for every* $\sigma \in \Sigma$ *and* $i \in |\sigma|$ *the event* $\sigma(i) \subseteq \mathbb{V}$ *satisfies* β *as a classical Boolean formula in control variables* $\mathbb{V}$.

A consequence of Prop. 3.3 is that the interface algebra contains ordinary classical Boolean algebra as the fragment of Boolean types. In particular, for Boolean types the Double Negation principle $\neg\neg\beta \cong \beta$ and Excluded Middle $\neg\beta \otimes \beta \cong true$ hold as well as the De-Morgan Laws $\neg(\beta_1 \wedge \beta_2) \cong \neg\beta_1 \otimes \neg\beta_2$ and $\neg(\beta_1 \otimes \beta_2) \cong \neg\beta_1 \wedge \neg\beta_2$. Boolean types, like all types satisfying $\neg\neg\phi \cong \phi$ or $\neg\phi \otimes \phi \cong true$, behave exactly like expressions of Boolean algebra, encapsulating a Boolean condition to be satisfied by each event in a sequence.

Pure Types. The sum operator $\oplus$ takes us outside the sub-language of Boolean types. The reason is that the truth of $\oplus$, e.g., in stability $A \oplus \neg A$, depends on the global behaviour of an activation and cannot be reduced to a *single* Boolean condition. This is highlighted by the difference between $\sigma \models A \oplus B$ which is the condition $\forall i \in |\sigma|, A \in \sigma(i)$ or $\forall i \in |\sigma|, B \in \sigma(i)$ and $\sigma \models A \otimes B$ which says $\forall i \in |\sigma|, A \in \sigma(i)$ or $B \in \sigma(i)$. The larger class of pure types, which includes $\oplus$, give us the possibility to express "Boolean" conditions *across* activations, as opposed to Boolean types which act *within* activations. The pure types, denoted by meta-variable ζ, are characterised syntactically as follows:

$$\zeta \quad ::= \quad \beta \mid \zeta \wedge \zeta \mid \zeta \oplus \zeta \mid \zeta \otimes \zeta \mid \phi \supset \zeta,$$

where β is Boolean and ϕ is an arbitrary type. Notice that not only every Boolean type, but also every negation $\neg\phi = \phi \supset \mathit{false}$, is pure according to this syntactic criterion.

Proposition 3.4 *Every pure type ζ has a representation $\zeta \cong \bigoplus_i \beta_i$ over Boolean types β_i.*

Elementary Types. Pure types have the special property that schedules Σ are bounded for them iff each individual activation $\sigma \in \Sigma$ is bounded, i.e., they express properties of individual activations. Formally, if $\Sigma_1 \models \zeta$ and $\Sigma_2 \models \zeta$ then $\Sigma_1 \cup \Sigma_2 \models \zeta$. Disjunctions $\zeta_1 \vee \zeta_2$ and resource types $\circ\zeta$, in contrast, do not share this locality property: Although each activation σ may satisfy ζ_1 or ζ_2, the schedule Σ as a whole need not be resource-bounded for $\zeta_1 \vee \zeta_2$ as this would mean all activations satisfy ζ_1 or all satisfy ζ_2. Similarly, each individual activation $\sigma \in \Sigma$ may validate ζ with some resource bound, without necessarily there being a *single* common bound for all activations in Σ.

A useful class of types containing $\vee$ and $\circ$ are those for which $\mathit{Bnd}(\phi)$ is canonically order-isomorphic to a Cartesian product of numbers, i.e., to $\mathbb{N}_\infty^n$ for some $n \geq 0$. These scheduling types ϕ with $\mathit{Bnd}(\phi) \cong \mathbb{N}_\infty^n$ are called *elementary*. They are generated by the grammar

$$\theta \quad ::= \quad \zeta \mid \theta \wedge \theta \mid \theta \oplus \theta \mid \theta \otimes \theta \mid \circ\zeta \mid \psi \supset \theta,$$

where ζ is pure and ψ is $\circ$-free. Elementary scheduling types are of special interest since their elements are first-order objects, i.e., vectors and matrices of natural numbers.

Elementary interfaces specify the resource consumption of logical controls. For instance, $\sigma \models (d,0) : \circ\zeta$, given $\zeta = \oplus_i \beta_i$ (see Prop. 3.4), says that σ enters and remains inside a region of events described by one of the Boolean conditions β_i and consumes at most d resource units to do that. The special case $\sigma \models d : \circ\mathit{false}$ says that σ consumes no more than d units during any instant. Similarly, $\sigma \models \zeta \supset (d,0) : \circ\xi$ with $\zeta = \oplus_i \beta_i$ and $\xi = \oplus_j \gamma_j$ says that every sub-activation $\sigma' \subseteq \sigma$ that runs fully inside one of the regions β_i must reach one of the regions γ_j with resources bounded by d. Then, $\sigma \models \zeta \supset (d,0) : \circ\mathit{false}$ means that σ consumes no more than d units while staying in any of the regions β_i.

To compactify the notation we will write tuples (d_1,d_2) for the bounds $((d_1,0),(d_2,0)) \in (\mathbb{N}_\infty \times \underline{1}) \times (\mathbb{N}_\infty \times \underline{1}) \cong \mathbb{N}_\infty \times \mathbb{N}_\infty$ of types such as $\circ\zeta_1 \oplus \circ\zeta_2$, $\circ\zeta_1 \wedge \circ\zeta_2$, $\circ\zeta_1 \otimes \circ\zeta_2$. We apply this simplification also to bounds $f \in \underline{1} \to \mathbb{N}_\infty \times \underline{1} \cong \mathbb{N}_\infty$ for types such as $\zeta_1 \supset \circ\zeta_2$: We write $[d] : \zeta_1 \supset \circ\zeta_2$, treating the bracketed value $[d]$ like a function $\lambda x.(d,0)$. In fact, $[d] : \zeta_1 \supset \circ\zeta_2$ is the special case of a 1×1 matrix. We will systematically write *column vectors* $[d_1;d_2]$ instead of $\lambda x.((d_1,0),(d_2,0))$ for the bounds of types such as $\zeta \supset \circ\zeta_1 \oplus \circ\zeta_2$, $\zeta \supset \circ\zeta_1 \wedge \circ\zeta_2$ or $\zeta \supset \circ\zeta_1 \otimes \circ\zeta_2$, and *row-vectors* $[d_1,d_2]$ in place of $\lambda x.\,\mathrm{case}\,x\,\mathrm{of}\,[(0,0) \to (d_1,0),(1,0) \to (d_2,0)]$ for types $\zeta_1 \vee \zeta_2 \supset \circ\zeta$. Our linearised matrix notation uses semicolon for row-wise and ordinary colon for columns-wise composition of sub-matrices. Specifically, $[d_{11};d_{21},d_{12};d_{22}]$ and $[d_{11},d_{12};d_{21},d_{22}]$ denote the same 2×2 matrix.

In the following Secs. 4 and 5 we are going illustrate different sub-algebras of specialised elementary types to manipulate combined functional and quantitative information and to facilitate interface abstractions. These generalise the algebra of dioids [4, 17] to full max-min-plus, obtaining an equally tight as uniform combination of scheduling algebra and logical reasoning.

4 Examples I: Network Flow, Shortest Path, Task Scheduling

The logical operations on types control the arithmetical operations on resource bounds. The next two Props. 4.1 and 4.2 sum up some important basic facts.

Proposition 4.1 *The arithmetic operations min, max and + compute worst-case bounds such that*

$$[d_1] : \zeta_1 \supset \circ\zeta_2 \;\wedge [d_2] : \zeta_2 \supset \circ\zeta_3 \;\preceq\; [d_1 + d_2] : \zeta_1 \supset \circ\zeta_3 \tag{1}$$
$$[d_1] : \zeta \supset \circ\zeta_1 \;\wedge [d_2] : \zeta \supset \circ\zeta_2 \;\preceq\; [max(d_1,d_2)] : \zeta \supset \circ(\zeta_1 \wedge \zeta_2) \tag{2}$$
$$[d_1] : \zeta \supset \circ\zeta_1 \;\wedge [d_2] : \zeta \supset \circ\zeta_2 \;\preceq\; [min(d_1,d_2)] : \zeta \supset \circ(\zeta_1 \oplus \zeta_2) \tag{3}$$
$$[d_1] : \zeta_1 \supset \circ\zeta \;\wedge [d_2] : \zeta_2 \supset \circ\zeta \;\preceq\; [max(d_1,d_2)] : (\zeta_1 \oplus \zeta_2) \supset \circ\zeta \tag{4}$$
$$[d_1] : \zeta_1 \supset \circ\zeta \;\wedge [d_2] : \zeta_2 \supset \circ\zeta \;\preceq\; [min(d_1,d_2)] : (\zeta_1 \wedge \zeta_2) \supset \circ\zeta. \tag{5}$$

The law (1) expresses a *sequential composition* of an offset by d_1 from control point ζ_1 to ζ_2 with a further shift of d_2 from ζ_2 to ζ_3. The best guarantee we can give for the cost between ζ_1 and ζ_3 is the addition $d_1 + d_2$. The bounds $[d_1]$ and $[d_2]$ act like typed functions with $[d_1 + d_2]$ being function composition, $[d_2] \cdot [d_1] = [d_1 + d_2]$. This is nothing but the multiplication of 1×1 matrices in max-plus or min-plus algebra. The law (2) is *conjunctive forking*: If it takes at most d_1 units from ζ to some control point ζ_1 *and* at most d_2 to ζ_2, then we know that within $max(d_1,d_2)$ we have activated both together, $\zeta_1 \wedge \zeta_2$. A special case of this occurs when $\zeta \cong true$, i.e., $d_1 : \circ\zeta_1 \wedge d_2 : \circ\zeta_2 \cong max(d_1,d_2) : \circ(\zeta_1 \wedge \zeta_2)$. Now suppose conjunction is replaced by sum $\zeta_1 \oplus \zeta_2$, i.e., we are only interested in activating one of ζ_1 *or* ζ_2, but do not care which. The worst-case bound for this *disjunctive forking* is the minimum, as seen in (3). Again, there is the special case $d_1 : \circ\zeta_1 \wedge d_2 : \circ\zeta_2 \cong min(d_1,d_2) : \circ(\zeta_1 \oplus \zeta_2)$. Dually, *disjunctive joins* (4) are governed by the maximum: Suppose that starting in ζ_1 activates ζ with at most d_1 cost and starting in ζ_2 takes at most d_2 resource units. Then, if we only know the activation starts from ζ_1 *or* ζ_2 but not which, we can obtain ζ if we are prepared to expend the maximum of both costs. If, however, we assume the schedule activates both ζ_1 *and* ζ_2, which amounts to *conjunctive join*, then the destination ζ is obtained with the minimum of both shifts, see (5).

Proposition 4.2 *Let ζ_1, ζ_2 be pure types which are persistent in the sense that whenever $\sigma(k) \models \zeta_i$ for $0 \le k < |\sigma|$, then $\sigma[k,:] \models \zeta_i$, too. Then,*

$$d_1 : \circ\zeta_1 \otimes d_2 : \circ\zeta_2 \;\preceq\; d_1 + d_2 : \circ(\zeta_1 \oplus \zeta_2) \tag{6}$$
$$(d_1 : \circ\zeta_1 \wedge (\zeta_1 \supset \zeta_2)) \otimes (d_2 : \circ\zeta_2 \wedge (\zeta_2 \supset \zeta_1)) \;\preceq\; d_1 + d_2 : \circ(\zeta_1 \wedge \zeta_2). \tag{7}$$

Consider (6) of Prop. 4.2. Suppose a schedule σ splits into two (sub-)threads $\sigma = \sigma_1 \cup \sigma_2$ each switching control ζ_1 and ζ_2 consuming at most d_1 and d_2 units, respectively. Since they can be arbitrarily interleaved and we do not know which one completes first, all we can claim is $\sigma(k) \models \zeta_i$ for some $k \le d_1 + d_2$ and $i = 1,2$. By persistence, this suffices to maintain ζ_i from level k onwards, so that $\sigma \models d_1 + d_2 : \circ(\zeta_1 \oplus \zeta_2)$. Without imposing further assumptions, a sub-thread may be allocated an unknown number of resource units, thereby stalling the progress of the other, unboundedly. The situation

changes, however, if the ζ_i are *synchronisation points* where the threads must give up control unless the other thread has passed its own synchronisation point ζ_j ($i \neq j$), too. This is the content of (7) and specified formally by the additional constraints $\zeta_i \supset \zeta_j$.

Prop. 4.1 and 4.2 highlight how the arithmetic of min-max-plus algebra are guided by the logical semantics of interface types. From this vantage point, resource analysis is nothing but a semantics-consistent manipulation of a collection of numbers: Whether $[d_1] : \phi_1$, $[d_2] : \phi_2$ are to be added, maximised or minimised depends on their types ϕ_1 and ϕ_2. In particular, keeping track of the types will make the difference between a max-of-sums (sum-of-mins) as opposed to a sum-of-maxes (min-of-sums).

4.1 Network Flow

Consider the dependency graph in Fig. 1 with control nodes $\mathbb{V} = \{A,B,C,D,E,F\}$ and dependency edges labelled by positive integers. Let us assume the graph models a communication network in which control nodes represent packet routers and edges are directed point-to-point connections of limited bandwidth. For instance, the router at node D receives packets from routers B and C through channels of bandwidth 1 and 4, respectively. It forwards the incoming traffic to routers E or F of bandwidth 5 and 4, respectively. The bandwidth measures the maximal amount of information that can travel across the channel per synchronisation instant. The analysis of the maximum throughput is a synchronous scheduling problem which can be modelled using interface types.

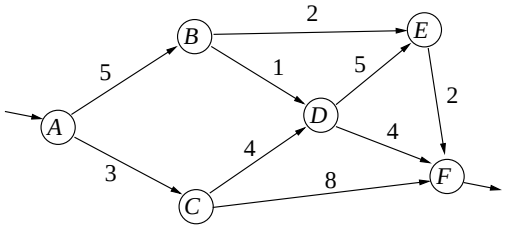


Figure 1: Scheduling Dependency Graph N

We associate with the network N a scheduling type ϕ_N, such that the amount of packets that can be pushed into a node X is given by the minimal d such that $\phi_N \preceq X \supset d : \circ\mathit{false}$, i.e., the maximal number of scheduling cycles that node X may be kept alive within any activation specified by ϕ_N. The idea is that if $\sigma \in [\![\phi_N]\!]$ is a valid activation of N then each cycle $i \in |\sigma|$ such that $X \in \sigma(i)$ represents a packet unit i sent through X. The event $\sigma(i) \subseteq \mathbb{V}$ encodes the packet's path, i.e., the set of all routers that payload unit i is passing on its journey through the network. The statement $\sigma \models X \supset d : \circ\mathit{false}$ then says that whenever X becomes alive in activation σ it handles no more d packets. This number may vary between activations. The minimal d, bounding all activations in this way, is the maximal throughput at X permitted by specification ϕ_N. Observe that both capacity values 0 and $-\infty$ are equivalent, $0 : \circ\mathit{false} \cong -\infty : \circ\mathit{false} \cong \mathit{false}$. In fact, the type $X \supset 0 : \circ\mathit{false}$ paraphrased "X forwards 0 packets" and $X \supset -\infty : \circ\mathit{false}$ saying "X does not forward any packets", are the same statements and equivalent to $\neg X$.

Now consider node D again. Within the synchronous measurement instant, all packets arriving at D must be scheduled to leave through channels $D \to E$ or $D \to F$. Consider an activation $\sigma \models D$, i.e., all $i \in |\sigma|$ are packets dispatched through D. Some of these will go to E, others to F and all go to one of the two. Hence there are sub-activations $\sigma = \sigma_1 \cup \sigma_2$ such that $\sigma_1 \models E$ and $\sigma_2 \models F$. Also, because of the channel limitations, there can be at most 5 packet units of the former and 4 of the latter type. Thus, $\sigma_1 \models E \wedge 5 : \circ\mathit{false}$ and $\sigma_2 \models F \wedge 4 : \circ\mathit{false}$. All in all, we have found the type specifying D and its connections in N to be $D \supset (E \wedge 5 : \circ\mathit{false}) \otimes (F \wedge 4 : \circ\mathit{false})$.

The tensor $\otimes$ is used to model the output branching at a node. Observe that if we increase one of the channel capacities to $+\infty$, say the one giving access to E, we get $D \supset (E \wedge +\infty : \circ\mathit{false}) \otimes (F \wedge 4 : \circ\mathit{false}) \cong D \supset E \otimes (F \wedge 4 : \circ\mathit{false})$ because $E \wedge +\infty : \circ\mathit{false} \cong E \wedge \mathit{true} \cong E$. This means the channel $D \to E$ does not impose any further constraints on the throughput besides what E prescribes. If we

decrease the capacity to 0, the type reduces to $D \supset (E \wedge 0 : \circ \mathit{false}) \otimes (F \wedge 4 : \circ \mathit{false}) \cong D \supset F \wedge 4 : \circ \mathit{false}$ since $E \wedge 0 : \circ \mathit{false} \cong E \wedge \mathit{false} \cong \mathit{false}$ and $\mathit{false} \otimes \phi \cong \phi$. Hence, a capacity of 0 behaves as if the channel was cut off completely. Consequently, the degenerated case of a node X without any exits would be specified by $X \supset \mathit{false}$ or $\neg X$. If we conjoin the types for all nodes of N as seen in Fig. 1, we get

$$\begin{aligned}
\phi_N =_{df} \quad & \mathit{true} \supset (A \wedge +\infty : \circ \mathit{false}) && (8)\\
& \wedge A \supset ((B \wedge 5 : \circ \mathit{false}) \otimes (C \wedge 3 : \circ \mathit{false})) && (9)\\
& \wedge B \supset ((E \wedge 2 : \circ \mathit{false}) \otimes (D \wedge 1 : \circ \mathit{false})) && (10)\\
& \wedge C \supset ((D \wedge 4 : \circ \mathit{false}) \otimes (F \wedge 8 : \circ \mathit{false})) && (11)\\
& \wedge D \supset ((E \wedge 5 : \circ \mathit{false}) \otimes (F \wedge 4 : \circ \mathit{false})) && (12)\\
& \wedge E \supset (F \wedge 2 : \circ \mathit{false}) && (13)\\
& \wedge F \supset (\mathit{true} \wedge +\infty : \circ \mathit{false}). && (14)
\end{aligned}$$

Type (8) designates A as the source node of the network. It formalises a source channel of infinite capacity permitting the global environment, represented by the logical control *true*, to push as many packets as possible into A. Analogously, destination node F (14) returns packets back to the external environment. Again, this sink channel has infinite capacity, since all packets arriving at F will delivered.

The throughput d_N of N is the smallest d such that $\phi_N \preceq d : \circ \mathit{false}$. To get the "exact" or "optimal" bound we must explore the network in breadth and depth. The analysis strategy involves non-linear global optimisation such as the Ford-Fulkerson or Goldberg's Preflow-Push algorithms. This is not the place to review these algorithm. We shall merely indicate how their logical content can be coded in type theory. Consider that each of the network implications (8)–(14) of the form $X \supset \otimes_Y (Y \wedge d_Y : \circ \mathit{false})$ can be used as an equation $X \cong X \wedge \otimes_Y (Y \wedge d_Y : \circ \mathit{false})$ for transformations by substitution. For example, proceeding forwards from the source A, breadth-first, we can derive

$$\begin{aligned}
\mathit{true} &\cong A\\
&\cong A \wedge ((B \wedge 5 : \circ \mathit{false}) \otimes (C \wedge 3 : \circ \mathit{false}))\\
&\cong A \wedge ((B \wedge ((E \wedge 2 : \circ \mathit{false}) \otimes (D \wedge 1 : \circ \mathit{false})) \wedge 5 : \circ \mathit{false})\\
&\qquad \otimes (C \wedge ((D \wedge 4 : \circ \mathit{false}) \otimes (F \wedge 8 : \circ \mathit{false})) \wedge 3 : \circ \mathit{false}))\\
&\cong ((A \wedge B \wedge E \wedge 2 : \circ \mathit{false}) && (15)\\
&\qquad \otimes (A \wedge B \wedge D \wedge 1 : \circ \mathit{false})) && (16)\\
&\qquad\quad \otimes (((A \wedge C \wedge D \wedge 3 : \circ \mathit{false}) && (17)\\
&\qquad\qquad \otimes (A \wedge C \wedge F \wedge 3 : \circ \mathit{false})) \wedge 3 : \circ \mathit{false}), && (18)
\end{aligned}$$

using the special $\wedge/\otimes$ distribution $X \wedge (\phi_1 \otimes \phi_2) \cong (X \wedge \phi_1) \otimes (X \wedge \phi_2)$ for atoms $X \in \mathbb{V}$, and the derivable laws $((\phi_1 \wedge d_1 : \circ \mathit{false}) \otimes (\phi_2 \wedge d_2 : \circ \mathit{false})) \wedge e : \circ \mathit{false} \cong (\phi_1 \wedge d_1 : \circ \mathit{false}) \otimes (\phi_2 \wedge d_2 : \circ \mathit{false})$ for $e \geq d_1 + d_2$ and $((\phi_1 \wedge d_1 : \circ \mathit{false}) \otimes (\phi_2 \wedge d_2 : \circ \mathit{false})) \wedge e : \circ \mathit{false} \cong (\phi_1 \wedge e : \circ \mathit{false}) \otimes (\phi_2 \wedge e : \circ \mathit{false}) \wedge e : \circ \mathit{false}$ for $e \leq \mathit{min}(d_1, d_2)$.

The type (15)–(18) describes the resource usage of packets entering the network up to a depth of 3 nodes, classifying them into 4 separate flows: The packets from (15) pass through $A \to B \to E$ and can occupy at most 2 bandwidth units, those from (16) follow the path $A \to B \to D$ and have a volume of at most 1 unit. Furthermore, the packets (17) travelling along $A \to C \to D$ or (18) on path $A \to C \to F$ each have at most volume 3, as specified by $A \wedge C \wedge D \wedge 3 : \circ \mathit{false}$ and $A \wedge C \wedge F \wedge 3 : \circ \mathit{false}$. Moreover, their sum must not exceed the limit 3 either, as enforced by the extra outer conjunct $3 : \circ \mathit{false}$. The maximal

flow through the network can be obtained by applying the (in-)equations (15)–(18) in this fashion until saturation is achieved, when all logical controls may be dropped, turning equation $\cong$ into inequation $\preceq$:

$$\begin{aligned}
\mathit{true} &\cong A \cong \cdots \\
&\cong ((A \wedge B \wedge E \wedge F \wedge 2 : \circ\mathit{false}) \\
&\quad \otimes (A \wedge B \wedge D \wedge F \wedge 1 : \circ\mathit{false})) \\
&\quad \otimes (((((A \wedge C \wedge D \wedge F \wedge 3 : \circ\mathit{false}) \\
&\qquad \otimes (A \wedge C \wedge D \wedge E \wedge F \wedge 2 : \circ\mathit{false})) \wedge 3 : \circ\mathit{false}) \\
&\qquad \otimes (A \wedge C \wedge F \wedge 3 : \circ\mathit{false})) \wedge 3 : \circ\mathit{false}) \\
&\preceq (2 : \circ\mathit{false} \otimes 1 : \circ\mathit{false}) \\
&\quad \otimes ((((3 : \circ\mathit{false} \otimes 2 : \circ\mathit{false}) \wedge 3 : \circ\mathit{false}) \otimes 3 : \circ\mathit{false}) \wedge 3 : \circ\mathit{false}) \cong 6 : \circ\mathit{false},
\end{aligned}$$

using the laws $d : \circ\mathit{false} \wedge e : \circ\mathit{false} \cong \mathit{min}(d,e) : \circ\mathit{false}$ and $d : \circ\mathit{false} \otimes e : \circ\mathit{false} \cong d + e : \circ\mathit{false}$, derived from (3) and (6), respectively.

This saturation process is a fixed-point construction which may be implemented using a standard "max-flow" algorithm. Specifically, the graph algorithms of Ford-Fulkerson or Goldberg are efficient decision procedures for deciding the algebra induced by the fragment of types appearing in (8)–(18). This sub-algebra of "logical numbers" provides a purely algebraic interpretation for these standard algorithms. It should be clear that the graph-theoretic information is coded in the syntactic structure of the types. However, in contrast to plain graphs, types are equipped with behavioural meaning in the form of scheduling sequences. They generate a plus-min algebra of scheduling sequences which is not a linear algebra, as it does not satisfy distribution. Specifically, $e : \circ\mathit{false} \wedge (d_1 : \circ\mathit{false} \otimes d_2 : \circ\mathit{false}) \cong \mathit{min}(e, d_1 + d_2) : \circ\mathit{false} \preceq \mathit{min}(e,d_1) + \mathit{min}(e,d_2) : \circ\mathit{false} \cong (e : \circ\mathit{false} \wedge d_1 : \circ\mathit{false}) \otimes (e : \circ\mathit{false} \wedge d_2 : \circ\mathit{false})$. This approximation offset, of course, is why max-flow problems are not linear matrix problems but require global search and relaxation methods.

4.2 Shortest Path

A different interpretation of the scheduling graph Fig. 1 reads the edge labels as *distances* and asks for the length of the shortest path through the network. This leads to an "inverted" network algebra: The sequential composition of edges is *addition* and the branching of edges at a node is associated with the *minimum* operation, whereas in the network flow situation of Sec. 4.1, sequential composition corresponds to minimum and branching is addition. Not surprisingly, the shortest path interpretation invokes a different fragment of the type theory. Again, each node is a control variable $\mathbb{V} = \{A,B,C,D,E,F\}$. An activation σ models a journey through the network activating control nodes as it passes them. If σ activates X at time i, then $X \in \sigma(i)$, and if it traverses an edge $X \to Y$ with distance label d, then for some $0 \leq k \leq d$, $Y \in \sigma(i+k)$. Hence σ satisfies the type $X \supset d : \circ Y$. If there are several outgoing edges $X \to Y_1$ and $X \to Y_2$ and σ reaches X, then, because we are interested in the *shortest* path, we permit σ to explore both branches "in parallel". Hence, σ fulfils both implications $X \supset d_1 : \circ Y_1$ and $X \supset d_2 : \circ Y_2$. Following this idea, the network N as given in Fig. 1 comes out as the type specification

$$\begin{aligned}
\phi_N =_{df} & A \supset 5 : \circ B \wedge A \supset 3 : \circ C \wedge B \supset 1 : \circ D \wedge B \supset 2 : \circ E \\
& \wedge C \supset 4 : \circ D \wedge C \supset 8 : \circ F \wedge D \supset 5 : \circ E \wedge D \supset 4 : \circ F \wedge E \supset 2 : \circ F. \qquad (19)
\end{aligned}$$

The length of the shortest path between X and Y is the minimal d such that $\phi_N \preceq X \supset d : \circ Y$. By (1), sequentially connecting edges $X \supset d_1 : \circ Y$ and $Y \supset d_2 : \circ Z$ yields $X \supset d_1 + d_2 : \circ Z$, and a choice of two

paths $X \supset d_1 : \circ Z$ and $X \supset d_2 : \circ Z$ between the same start and end node, by (3) implies $X \supset min(d_1,d_2) : \circ Z$ as desired. Now the values of 0 and $-\infty$ have different meaning: $X \supset 0 : \circ Y$ is equivalent to $X \supset Y$ modelling an edge without cost. In contrast, $X \supset -\infty : \circ Y$ is semantically the same as $X \supset \mathit{false}$ which says that no activation reaches control node X. A distance $+\infty$ expresses absence of a connection since $X \supset +\infty : \circ Y \cong X \supset \mathit{true} \cong \mathit{true}$ which does not give any information about how to reach Y from X.

It is well-known how to compute shortest paths by linear programming. This exploits the distribution law $\min(e+d_1, e+d_2) = e + min(d_1,d_2)$, which permits us to organise the scheduling bounds in the network theory (19) in form of matrices and to manipulate them using typed matrix multiplications. For instance, we can combine the two outgoing edges of A into a single type

$$(A \supset 5 : \circ B) \;\wedge\; (A \supset 3 : \circ C) \;\cong\; A \supset (5,3) : \circ B \wedge \circ C \;\cong\; [5;3] : A \supset \circ B \wedge \circ C, \tag{20}$$

where $[5;3]$ abbreviates the function $\lambda x.\,((5,0),(3,0))$ interpreted as a *column vector* of numbers. Dually, the two incoming edges into node D can be combined into a single type

$$(B \supset 1 : \circ D) \;\wedge\; (C \supset 4 : \circ D) \;\cong\; [1,4] : B \vee C \supset \circ D, \tag{21}$$

where $[1,4]$ is the function $\lambda x.$ case x of $[0 \to (1,0), 1 \to (4,0)]$ thought of as a *row* vector. The type algebra, essentially (1) and (3), proves that the conjunction of both (20) and (21) implies the matrix multiplication

$$([5;3] : A \supset \circ B \wedge \circ C) \wedge ([1,4] : B \vee C \supset \circ D) \;\preceq\; min(5+1, 3+4) : A \supset \circ D \;=\; [1,4] \cdot [5;3] : A \supset \circ D$$

in *min-plus* algebra. More generally, for every sub-network with source nodes $X_1, X_2, \ldots, X_m$ and sink nodes $Y_1, Y_2, \ldots, Y_n$ we have an elementary type $D : \vee_{i=1}^m X_i \supset \wedge_{j=1}^n \circ Y_j$ describing the shortest path between any source to any target, in which the scheduling bound $D \in Bnd((\vee_{i=1}^m X_i) \supset \otimes_{j=1}^n \circ Y_j)$ behaves like a $n \times m$ matrix in min-plus algebra. For instance, take the decomposition of N into the edge sets $N_1 =_{df} \{A \to B, A \to C\}$, $N_2 =_{df} \{B \to E, B \to D, C \to D, C \to F\}$ and $N_3 =_{df} \{D \to E, D \to F, E \to F\}$:

$$\begin{aligned} D(N_1) &= [5;3] \;\; : A \supset (\circ B \wedge \circ C) \\ D(N_2) &= [1;2;+\infty,4;+\infty;8] \;\; : (B \vee C) \supset (\circ D \wedge \circ E \wedge \circ F) \\ D(N_3) &= [4,2,0] \;\; : (D \vee E \vee F) \supset \circ F. \end{aligned}$$

The shortest path from A to F is then obtained by multiplying these matrices

$$[4,2,0] \cdot [1;2;+\infty,4;+\infty;8] \cdot [5;3] = [4,2,0] \cdot [6;7;11] = 9 : A \supset \circ F$$

in min-plus-algebra. The type-theoretic approach facilitates a compositional on-the-fly construction of the *shortest path matrix*. The pure algebraic technique would combine all the information in a global 6×6 network matrix $N : (\vee_{X \in \mathbb{V}} X) \supset (\wedge_{X \in \mathbb{V}} \circ X)$ where $(N)_{XY} = d < +\infty$ if there exists an edge $X \supset d : Y$ in ϕ_N. Then, the shortest path matrix is $N^* = Id \wedge N \wedge N^2 \wedge \cdots$, where Id is the identity matrix with 0s in the diagonal and $+\infty$ everywhere else and $\wedge$ is the operation of forming element-wise minimum, lifting the logical operation $d_1 : \circ X \wedge d_2 : \circ X \cong min(d_1,d_2) : \circ X$ to matrices. The entries in N^* are the shortest distances between any two nodes in the network.

This way of solving shortest paths is well-known, of course. But now the behavioural typing permits us safely to play over- and under-approximation games which are difficult to control in pure algebra or graph theory without model-theoretic semantics. Just to give a simple example, suppose we wanted to derive a lower bound on the shortest path. Such can be obtained by identifying some of the control

nodes, i.e., pretending we could jump between them on our path to reach the destination. For instance, assuming $C \equiv B$, we find that $\phi_N \wedge C \equiv B \preceq A \supset 7 : \circ F$ is the shortest distance. Since the conjunction $\phi_N \wedge C \equiv B$ specifies a subset of activations, the shortest distance between A and F relative to $\phi_N \wedge C \equiv B$ is a lower bound on the shortest distance relative to ϕ_N. It may be more efficient to compute since the network $\phi_N \wedge C \equiv B$ only has 5 different nodes rather than 6 as with ϕ_N.

4.3 Task Scheduling

In yet another interpretation of network N the nodes are tasks and edges scheduling dependencies associated with upper bounds for task completion. Computing the worst-case completion time for the overall schedule, sequential composition of edges corresponds to addition as in the shortest path scenario Sec. 4.2 but branching now involves maximum rather than the minimum. Again, this is induced by the logical nature of the problem, the fact that the input join now is conjunctive rather than disjunctive as before. For instance, task D in Fig. 1 cannot start before *both* tasks *C and B* have started with a set-up delay of 4 time units from the start of C and 1 unit from B. Let us assume the task activation times are included in these set-up delays. To model this type-theoretically we take the edges as the atomic control variables, i.e., $\mathbb{V} = \{AC, AB, CD, CF, BD, BE, DE, DF, F\}$. Whenever $XY \in \sigma(i)$, for $i \in |\sigma|$, this says that the edge XY is ready, i.e., the source task X is completed and the start token has arrived at the corresponding control input of target task Y. The node D establishes a logical-arithmetical relationship between its input edges CD, BD and its output edges DF, DE, given by $CD \wedge BD \supset (4 : \circ DF) \wedge (5 : \circ DE)$. Overall,

$$\begin{aligned}\phi_N =_{df} & (\mathit{true} \supset 3 : \circ AC \wedge 5 : \circ AB) \wedge (AC \supset 4 : \circ CD \wedge 8 : \circ CF) \\ & \wedge (AB \supset 1 : \circ BD \wedge 2 : \circ BE) \wedge ((CD \wedge BD) \supset 4 : \circ DF \wedge 5 : \circ DE) \\ & \wedge (DE \wedge BE \supset 2 : \circ EF) \wedge (CF \wedge DF \wedge EF \supset 0 : \circ F).\end{aligned}$$

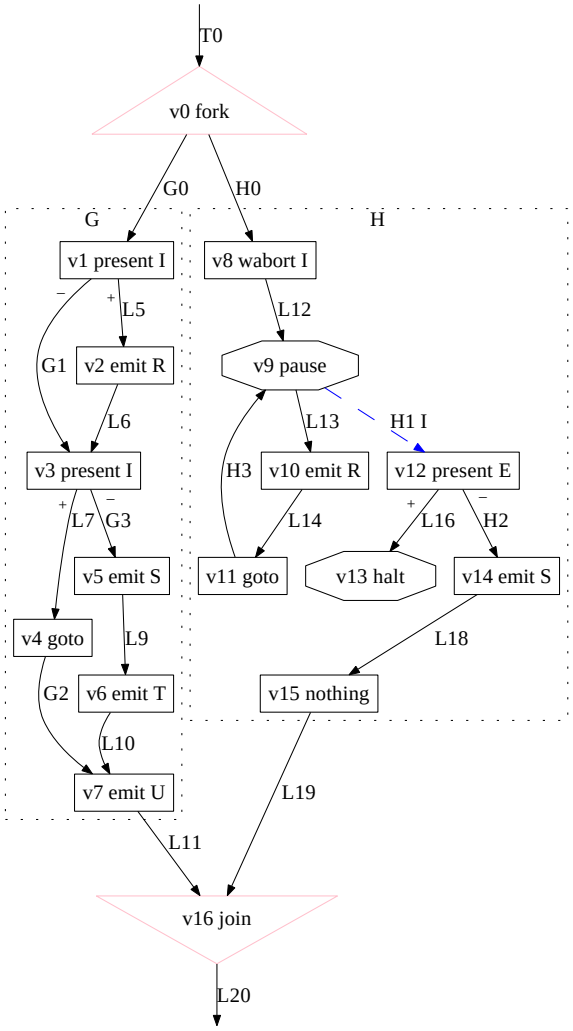
The critical path is the minimal d such that $\phi_N \preceq d : \circ F$. It can be computed by linear programming involving matrix multiplication in max-plus algebra using essentially the laws (1) and (2).

5 Examples II: Esterel-style Synchronous Multi-threading

Like task scheduling in Sec. 4.3, the timing analysis of Esterel programs [6, 22] involves max-plus algebra, yet takes place in an entirely different fragment of the type theory. Instead of implications $\zeta_1 \wedge \zeta_2 \supset \circ\xi_1 \wedge \circ\xi_2$ as in Sec. 4.3 we employ dependencies of the form $\zeta_1 \vee \zeta_2 \supset \circ\xi_1 \oplus \circ\xi_2$, which are handled by (1) and (4) rather than (1) and (2). In addition, we use the tensor $\otimes$ for capturing multi-threaded parallelism. Here we provide some further theoretical background for the work reported in [22].

Esterel programs communicate 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 end`), 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 activation 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 2b. 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, after having executed the skip statement `nothing`.



(a) CKAG

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

(b) Esterel module T

```
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
```

(c) KEP Assembler

Figure 2: Esterel module T (b) with control-flow graph (a) and resulting KEP Assembler (c).

The *concurrent KEP assembler graph* [18] (CKAG, see Fig. 2a) captures the control flow, both standard control and abortions, of an Esterel program. The CKAG is derived from the Esterel program by structural translation. For a given CKAG, the generation of assembly code for the Kiel Esterel Processor (KEP) [18, 19], executing synchronous parallelism by multi-threading, is straight-forward (see Fig. 2c).

Let $\mathbb{S}$, $\mathbb{L}$ and $\mathbb{M}$ be disjoint sets of (input or output) *signals*, control flow *labels* and synchronisation *states*, respectively. For the Esterel module in Fig. 2 we have $\mathbb{S} = \{I,E,R,S,T,U\}$, $\mathbb{L} = \{L0,\ldots,L20,G0,\ldots,G3,H0,\ldots,H3\}$. As synchronisation states we use the names of the atomic de-

lay nodes, i.e., the pause, halt and join nodes, $\mathbb{M} = \{v_9, v_{13}, v_{16}\}$. These describe the different state bits of the synchronous automaton coded by the program block T. To distinguish the cases of a thread starting from or ending in a given state $s \in \mathbb{M}$ during an instant we use the modifiers $out(s)$ and $in(s)$. The former expresses that the thread is leaving from s at the beginning of the instant and the latter that it enters and terminates the instant in s. The set $\mathbb{M}^+ =_{df} \{out(s), in(s) \mid s \in \mathbb{M}\}$ collects these atomic statements. The set of *control variables*, specifying the atomic control points of a program module, is the union $\mathbb{V} = \mathbb{S} \cup \mathbb{L} \cup \mathbb{M}^+$. All the controls $out(s)$ are stable, i.e., we may assume $out(s) \oplus \neg out(s)$. This is not true for controls $in(s)$ which are switched on dynamically as the schedule enters a delay node.

One possible activation of the Esterel module T in Fig. 2a would be as follows. Initially, control variable $T0$ 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 of Fig. 2c, each taking one instruction cycle (ic). The two PAR instructions set up internal counters for thread control, which does not change the set of events in the variables of Fig. 2a. Hence, $\sigma(1) = \sigma(2) = \{T0\}$. After the PARE both control variable $G0$, $H0$ 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. 2c assigns higher priority to H so that our activation 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 activation of pause either moves to v_{12} (weak immediate abort) or terminates. Let us assume the latter, i.e., $\sigma(5) = \{T0, G0, H0, L12, in(v_9)\}$, where thread H is finished up for the instant and has entered a wait state in node v_9. The activation continues with the first instruction of G, the present node v_1 at label $G0$. Since I is assumed absent, its activation effects a jump to label $G1$, i.e., $\sigma(6) = \{T0, G0, H0, L12, in(v_9), G1\}$. Thereafter, we run sequentially through nodes v_3, v_5, v_6, v_7 giving $\sigma(7) = \sigma(6) \cup \{G3\}$, $\sigma(8) = \sigma(7) \cup \{L9\}$ and $\sigma(9) = \sigma(8) \cup \{L10\}$.

Executing the final emit instruction v_7 hits the join at entry $L11$, so that $\sigma(10) = \{T0, G0, H0, L12, in(v_9), G1, G3, L9, L10, L11\}$. Now both threads G and H are finished. While G is terminated and hands over to the main thread T for good, H is still pausing in v_9. It takes one activation step of the join node v_{16} to detect this and to terminate the synchronous instant of T with the final event $\sigma(11) = \{T0, G0, H0, L12, in(v_9), G1, G3, L9, L10, L11, in(v_{16})\}$. Overall, we get an activation of the outer-most main thread of T, $\sigma = \sigma(0), \ldots, \sigma(11)$, starting from program label $T0$ consisting of 12 ics in total. In the next logical instant when T is resumed in v_{16} and v_9, with initial event $\sigma(0) = \{out(v_9), out(v_{16})\}$, 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 $L20$ and out of T instantaneously.

Activation sequences starting in control label $T0$ and ending in $L20$ are called *through paths*, those starting in $T0$ and pausing in a synchronisation state $in(s)$, $s \in \{v_9, v_{13}, v_{16}\}$, are *sink paths*; *source paths* begin in a state $out(s)$ and end in $L20$, while *internal paths* begin in a state and end in a state.

Esterel IO-Interface Types. Our normal form interfaces to describe Esterel-KEP modules are of the form $\theta = \phi \supset \psi$, with *input control* $\phi = \bigvee_{i=1}^{m} \zeta_i$ and *output control* $\psi = \bigoplus_{k=1}^{n} \circ \xi_k$ where the ζ_i and ξ_k are pure types. The former ϕ captures all the possible ways in which a program module (or any other fragment) of type θ can be started within an instant and the latter ψ sums up the ways in which it can be exited during the instant. Intuitively, $\Sigma \models \theta$ says that whenever the schedule Σ enters the fragment through one of the input controls ζ_i then within some bounded number of ics it is guaranteed to exit through one of the output controls ξ_k. The disjunction $\vee$ in the input control ϕ models the *external* non-determinism resolved by the environment which determines how a program block is started. On the output side ψ, the selection of which exit ξ_k is taken is expressed by $\oplus$ since it is an *internal* choice which

is dynamically resolved during each activation. Each delay operator $\circ$ stands for a possibly different delay depending on which output ξ_k is taken. Contrast this with an output control such as $\psi = \circ(\bigoplus_{k=1}^{n} \xi_k)$ which only specifies one bound for all exits ξ_k. An interface bound $T \in Bnd(\phi \supset \psi)$ can be understood as a $n \times m$ shaped timing matrix relative to the Boolean controls ζ_i and ξ_k serving as "base" vectors. The logical conjunction of these interfaces in a fixed set of such base controls corresponds to matrix multiplications in max-plus algebra. 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. Two important operations on IO-interfaces are matrix multiplication and the Kronecker product which in our scheduling algebra are now strongly typed and thus receive semantic meaning in logical spaces.

Transient and Sequential Submodules G and H. A full and exact WCRT specification encapsulating the synchronous block 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 $[6,4,3,1] : G0 \vee G1 \vee G3 \vee G2 \supset \circ L11$. This is still not the exact description of G since it neither expresses the dependency of the WCRT on signal I, nor the emissions of R, S, T, U. For instance, 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 $v_1+v_2+v_3+v_5+v_6+v_7$ with delay 6 is not executable. To capture this, we consider signal I as another control input and refine the WCRT interface type of G:

$$[5,5,3,4,3,1] : (G0 \wedge I) \vee (G0 \wedge \neg I) \vee (G1 \wedge I) \vee (G1 \wedge \neg I) \vee G3 \vee G2 \supset \circ L11. \tag{22}$$

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. 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 optimise 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 column vector $[0;0] : G0 \supset \circ(G0 \wedge I) \oplus \circ(G0 \wedge \neg I)$ is sound and can be used to compress the two entries of value 5 in (22) into a single value $5 = max(5,5)$ giving $[5,3,4,3,1] : G0 \vee (G1 \wedge I) \vee (G1 \wedge \neg I) \vee G3 \vee G2 \supset \circ L11$. 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 of precision. This finally yields $[5,3,4,1] : G0 \vee ((G1 \wedge I) \oplus G3) \vee (G1 \wedge \neg I) \vee G2 \supset \circ L11$. Still, if we only ever intend to use G as an encapsulated block with entry $G0$ and exit $L11$ the following typing is sufficient:

$$[5] : G0 \supset \circ L11. \tag{23}$$

Now we take a look at the sequential control flow which starts and terminates in pause and halt nodes. Consider the sub-module H from Fig. 2a consisting of nodes v_8–v_{15}. Nodes wabort, emit, goto, present, nothing are transient and specified as before for G. But now the instantaneous paths are broken by the delay nodes v_9 and v_{13}.

First, consider the pause node v_9. It can be entered by two controls, line number L12 and program label H3, and left via two exits, a non-instantaneous edge L13 and an 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 reaction, 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 as follows:

$$[1;1,1;1] \quad : \quad H3 \vee L12 \supset \circ H1 \oplus \circ in(v_9) \tag{24}$$

$$[1] \quad : \quad out(v_9) \supset \circ L13. \tag{25}$$

The interface (24) says that if pause is entered through H3 or L12 it can be left through H1 or terminate (*in*) inside the pause. In all cases activation takes 1 instruction cycle. Since there are no differences in the delays we could bundle the controls $H3$, $L12$ and compress the matrix (24) as $[1] : H3 \oplus L12 \supset \circ(H1 \oplus in(v_9))$ without losing information. We could also record the dependency of control on signal I, with the more precise interface $[1;-\infty,-\infty,1] : ((H3 \oplus L12) \wedge I) \vee ((H3 \oplus L12) \wedge \neg I) \supset \circ H1 \oplus \circ in(v_9)$. 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 (25) accounts for threads starting in the pause which must necessarily pass control to L13 within one instruction cycle.

The halt node v_{13} in Fig. 2a 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 specified by the interface $[1,1] : (out(v_{13}) \vee L16) \supset \circ in(v_{13})$. By composition from the WCRT interfaces of nodes v_{12}–v_{15} using matrix multiplications in max-plus algebra we get

$$H = [5;4,7;6] : H0 \vee out(H) \supset \circ L19 \oplus \circ in(H) \tag{26}$$

recording the lengths of the longest through path $v_8 + v_9 + v_{12} + v_{14} + v_{15}$, sink path $v_8 + v_9 + v_{12} + v_{13}$, source path $v_9 + v_{10} + v_{11} + v_9 + v_{12} + v_{14} + v_{15}$ and internal path $v_9 + v_{10} + v_{11} + v_9 + v_{12} + v_{13}$.

Multi-threading Composition: Fork and Join. Finally, consider the two blocks G and H as they are combined inside the Esterel module T (Fig. 2a) and synchronised by fork and join nodes v_0 and v_{16}. The main thread starts G and H in their initial controls, i.e., by activating $G0 \wedge H0$. Then, the executions of G and H are interleaved, depending on the priorities assigned by the compiler about which we shall make no assumptions. Child thread G can only run through its instantaneous path until it reaches L11 where it is stopped by the join. The sequential block H has two options: It can take its instantaneous through path stopping at $L19$ or it pauses in one of its delay nodes. In the former case we have reached $L11 \wedge L19$, where the synchronising join takes over letting the main thread continue by instantaneously activating L20 within the same instant. In the latter case we have activated $L11 \wedge in(H)$ where the synchronous instant is finished and the combined system pauses. Activation is resumed in the next instant from $L11 \wedge out(H)$, while G is still inactive and waiting at $L11$. Child thread H may either leave instantaneously through $L19$, giving $L11 \wedge L19$ overall, or once more pause internally, leading again to $L11 \wedge in(H)$.

This synchronous composition is obtained by the Kronecker product $GH =_{df} G' \otimes H'$ where G' and H' are the stand-alone interfaces of G (23) and H (26) instrumented for the synchronisation:

$$G' = Sync_1 \wedge [5,0] : G0 \vee L11 \supset \circ L11$$
$$H' = Sync_2 \wedge [5;4,7;6] : H0 \vee out(H) \supset \circ L19 \oplus \circ in(H).$$

G is extended by the additional input control $L11$ and trivial path $[0] : L11 \supset \circ L11$ to let G start an instant from $L11$ when H is pausing. The conjunct $Sync_1 =_{df} \neg L11$ expresses the synchronisation whereby G finishes once it reaches $L11$. Similarly, the conjunct $Sync_2 =_{df} \neg(L19 \oplus in(H))$ added to the interface (26)

stops H from continuing its activation instant past $L11$ or $in(H)$. The Kronecker product $G' \otimes H'$ now generates all possible interleaving of activations specified by type G' with those from type H':

$$\begin{aligned} G' \otimes H' \preceq [5,0] \otimes [5;4,7;6] \;&=\; [5 \cdot [5;4,7;6], 0 \cdot [5;4,7;6]] \;=\; [10;9,12;11,5;4,7;6] \\ &: (G0 \wedge H0) \vee (G0 \wedge out(H)) \vee (L11 \wedge H0) \vee (L11 \wedge out(H)) \supset \circ(L11 \wedge L19) \oplus \circ(L11 \wedge in(H)). \end{aligned}$$

In the synchronised composition GH we are only interested in the (surface) paths initiated by $G0 \wedge H0$ and the (depth) paths activated by the combination $L11 \wedge out(H)$. All other paths cannot be activated inside the fork and join context. Thus, we drop these column vectors and only continue with

$$\begin{aligned} GH \;&=\; [10;9,12;11,5;4,7;6] \cdot [0;-\infty;-\infty;-\infty,-\infty;-\infty;-\infty,0] \;=\; [10;9,7;6] \\ &: (G0 \wedge H0) \vee (L11 \wedge out(H)) \supset \circ(L11 \wedge L19) \oplus \circ(L11 \wedge in(H)). \end{aligned}$$

This models the concurrent composition of G and H but not yet the interface of the composite block T with fork and join as depicted in Fig. 2a. These are additional components specified as

$$\begin{aligned} \mathsf{join} &= [1;-\infty,-\infty;1] : (L11 \wedge L19) \vee (L11 \wedge in(H)) \supset \circ L20 \oplus \circ in(T) \\ \mathsf{fork} &= [3;-\infty,-\infty;0] : T0 \vee out(T) \supset \circ(G0 \wedge H0) \oplus \circ(L11 \wedge out(H)) \end{aligned}$$

with new state controls $in(T)$ and $out(T)$ for module T. The JOIN instruction in line 19 of Fig. 2c is always executed upon termination of both threads from G and H inside T and the associated activation time of one ic is accounted for in the join interface above. Specifically, this is a through path $[1] : (L11 \wedge L19) \supset \circ L20$ and source path $[1] : L11 \wedge in(H) \supset \circ in(T)$. The entry $[3] : T0 \supset \circ(G0 \wedge H0)$ of fork includes the ics for two PAR, one PARE from lines 1-3 of Fig. 2c. Adding fork and join on the input and output side then obtains

$$T = [1;-\infty,-\infty;1] \cdot [10;9,7;6] \cdot [3;-\infty,-\infty;0] \;= [14;13,8;7] \;:\; T0 \vee out(T) \supset \circ L20 \oplus \circ in(T)$$

for the composite module T. Indeed, 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 as 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 specific WCRT algorithms such as the one of [6] many of the matrix multiplications shown above are executed 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 for modular analyses are needed. Our logical interface algebra can be used to keep track of the semantic meaning of WCRT data. Even without data-dependencies, the WCRT interfaces presented here give rise to a depth-first search algorithm [22] which is already more precise than the one presented in [6].

6 Related Work

Most interface models in synchronous programming are restricted to causality issues, *i. e.*, dependency analysis without considering quantitative time. Moreover, the granularity of dependency is limited. E.g., the modules of André *et al.* [3] do not permit instantaneous interaction. Such a model is not suitable

for compositional, intra-instant, scheduling analysis. Hainque *et al.* [9] 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. This is fine for concurrent execution but too restricted to model single- or multi-threaded execution compositionally. The interface model also does not cover data dependencies and thus cannot deal with dynamic schedules. It also does not support quantitative resource information, either.

The causality interfaces of Lee *et al.* [17] are much more flexible. These are functions associating with every pair of input and output ports an element of a *dependency domain D*, which expresses if and how an output depends on some input. Causality analysis is then performed by multiplication on the global system matrix. Using an appropriate dioid structure *D*, one can perform the analyses of Hainque et. al. [9] as well as restricted forms of WCRT. Lee's interfaces presuppose a fixed static distinction between inputs and outputs and cannot express the difference between an output depending on the joint presence of several values as opposed to depending with each input individually. Similarly, there is no coupling of outputs, *e. g.*, that two outputs always occur together at "the same time." Thus, they do not support full AND- and OR-type synchronisation dependencies for representing multi-threading and multi-processing. Also, the model does not include data dependency. The work reported here can be seen as an extension of [17] to include such features. In particular, note that our scheduling interfaces can also be used in situations where linear algebra is not applicable, as in the case of network flow problems.

Recent works [27, 13] combining network calculus [4, 7] with real-time interfaces are concerned with the compositional modelling of regular execution patterns. Existing interface theories [17, 27, 13], which aim at the verification of resource constraints for real-time scheduling, handle timing properties such as task execution latency, arrival rates, resource utilisation, throughput, accumulated cost of context switches, and so on. The dependency on data and control flow is largely abstracted. For instance, since the task sequences of Henzinger and Matic [13] are independent of each other, their interfaces do not model concurrent forking and joining of threads. The causality expressible there is even more restricted than that by Lee *et al.* [17] in that it permits only one-to-one associations of inputs with outputs. The interfaces of Wandeler and Thiele [27] for modular performance analysis in real-time calculus are like those of Henzinger and Matic [13] but without sequential composition of tasks and thus do not model control flow. On the other hand, the approaches [27, 13] can describe continuous and higher-level stochastic properties which our interface types cannot.

AND- and OR-type synchronisation dependencies are important for synchronous programming since reachability of control nodes in general depends both conjunctively and disjunctively on the presence of data. Also, control branching may be conjunctive (as in multi-threading or concurrent execution) or disjunctive (as in single-threaded code). Moreover, execution may depend on the absence of data (negative triggering conditions), which makes compositional modelling 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 [27, 13] does not address causality issues arising from feedback and negative triggering conditions. The interface automata of Alfaro, Henzinger, Lee, Xiong [1, 15] model synchronous macro-states and assume that all stabilisation processes (sequences of micro-states) can be abstracted into atomic interaction labels. The introduction of *transient states* [16] alleviates this, but the focus is still on regular (scheduling) behaviour. The situation is different, however, for cyclic systems, in which causality information is needed. Our interface algebra is semantically sound with respect to feedback and indeed supports causality analysis as a special case: A signal A is *causal* if $\circ A \oplus \neg A$ can be derived in the type theory of a module. Because of the complications arising from causality issues, there is currently no robust component model for synchronous programming. We believe

that the interface types introduced in this paper, cover new ground towards such a theory.

Finally, note that our algebra is not intended as a general purpose interface model such as, e.g., the *relational interfaces* of Tripakis *et al.* [26]. While these relational interfaces permit contracts in first-order logic between inputs and outputs, our interfaces only describe propositional relations. Therefore, our algebra cannot describe the full functional behaviour of data processing (other than by coding it into finite Booleans). Our interfaces are logically restricted to express monotonic scheduling processes and the resource consumption inside synchronous instants. Because we use an intuitionistic realisability semantics (Curry-Howard) we obtain enough expressiveness to deal with causality problems and upper-bound scheduling costs. The interface algebra does not aim to cover behavioural aspects of sequences of instants such as in approaches based on temporal logics or the timed interfaces of Alfaro, Henzinger and Stoelinga [2], which build on timed automata. The scheduling problem addressed here is a simpler problem in the sense that it arises afresh *within* each synchronous step and does not need to carry (e.g., timing) constraints *across* steps. However, note that our algebra can fully capture finite-state sequential transition functions in the standard way by duplicating propositional state variables s using $out(s)$ and $in(s)$ as seen in Sec. 5. An *inter-instant* transition (instantaneous, no clock tick) between s_1 and s_2 is given by the implication $out(s_1) \supset \circ in(s_2)$ while the *intra-instant* transition (sequential, upon clock tick) is the weak implication $\neg in(s_1) \oplus out(s_2)$. In this way, we can derive exact state-dependent worst-case bounds across all reachable states of a finite state behaviour.

The scheduling algebra in this paper extends [21] in that it not only captures concurrent execution (as in combinational circuits) but also includes the tensor $\otimes$ for multi-threading. More subtly, while [21] is restricted to properties of activation sequences stable under the *suffix* preordering, here we consider the much richer lattice of *arbitrary* sub-sequences. This paper introduces the theory behind [22] which reported on the application to WCRT analysis for Esterel and also provides more detailed information on the modelling in Sec. 5.

Acknowledgements. The author would like to thank the anonymous reviewers for their suggestions to improve the presentation.

References

[1] L. de Alfaro & T. Henzinger (2001): *Interface automata*. In: *Proc. Foundations of Software Engineering*, ACM Press, pp. 109–120.

[2] L. de Alfaro, Th. Henzinger & Marielle Stoelinga (2002): *Timed interfaces*. In: *Proc. EMSOFT'02*.

[3] C. André, F. Boulanger, M.-A. Péraldi, J. P. Rigault & G. Vidal-Naquet (1997): *Objects and synchronous programming*. *European Journal on Automated Systems* 31(3), pp. 417–432.

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

[5] Gérard Berry & Georges Gonthier (1992): *The Esterel synchronous programming language: Design, semantics, implementation*. *Science of Computer Programming* 19(2), pp. 87–152.

[6] Marian Boldt, Claus Traulsen & Reinhard von Hanxleden (2008): *Worst case reaction time analysis of concurrent reactive programs*. *ENTCS* 203(4), pp. 65–79. Proc. SLA++P'07, March 2007, Braga, Portugal.

[7] J. Le Boudec & P. Thiran (2001): *Network Calculus - A theory of deterministic queuing systems for the internet*, *Lecture Notes in Computer Science* 2050. Springer.

[8] Paul Le Guernic, Thierry Goutier, Michel Le Borgne & Claude Le Maire (1991): *Programming real time applications with SIGNAL*. *Proceedings of the IEEE* 79(9).

[9] Olivier Hainque, Laurent Pautet, Yann Le Biannic & Eric Nassor (1999): *Cronos: A separate compilation toolset for modular Esterel applications.* In: Jeannette M. Wing, Jim Woodcock & Jim Davies, editors: *World Congress on Formal Methods, Lecture Notes in Computer Science* 1709, Springer, pp. 1836–1853.

[10] Nicolas Halbwachs (1998): *Synchronous programming of reactive systems, a tutorial and commented bibliography*. In: *Tenth International Conference on Computer-Aided Verification, CAV '98*, LNCS 1427, Springer Verlag, Vancouver (B.C.).

[11] Nicolas Halbwachs (2005): *A synchronous language at work: The story of Lustre*. In: *Third ACM-IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'05*, Verona, Italy.

[12] D. Harel, A. Pnueli, J. Pruzan-Schmidt & R. Sherman (1987): *On the formal semantics of Statecharts*. In: *LICS '87*, IEEE Computer Society Press, pp. 54–64.

[13] Th. Henzinger & S. Matic (2006): *An interface algebra for real-time components*. In: *Proceedings of the 12th Annual Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 253–266.

[14] C. Huizing (1991): *Semantics of Reactive Systems: Comparison and Full Abstraction*. Ph.D. thesis, Eindhoven Univ. of Technology.

[15] E. A. Lee & Y. Xiong (2001): *System-level types for component-based design*. In: *Workshop on Embedded Software EMSOFT 2001*, Lake Tahoe, CA, USA.

[16] E. A. Lee & Y. Xiong (2004): *A behavioral type system and its application in Ptolemy II*. *Formal Aspects of Computing* 13(3), pp. 210–237.

[17] E. A. Lee, H. Zheng & Y. Zhou (2005): *Causality interfaces and compositional causality analysis*. In: *Foundations of Interface Technologies (FIT'05)*, ENTCS, Elsevier.

[18] Xin Li, Marian Boldt & Reinhard von Hanxleden (2006): *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.

[19] Xin Li & Reinhard von Hanxleden (2010): *Multi-threaded reactive programming—The Kiel Esterel processor*. *IEEE Transactions on Computers* .

[20] G. Luettgen & M. Mendler (2002): *The intuitionism behind Statecharts steps*. *ACM Transactions on Computational Logic* 3(1), pp. 1–41.

[21] M. Mendler (2000): *Characterising combinational timing analyses in intuitionistic modal logic*. *The Logic Journal of the IGPL* 8(6), pp. 821–853.

[22] Michael Mendler, Reinhard von Hanxleden & Claus Traulsen (2009): *WCRT algebra and interfaces for Esterel-style synchronous processing*. In: *Proceedings of the Design, Automation and Test in Europe (DATE'09)*, Nice, France.

[23] Amir Pnueli & M. Shalev (1991): *What is in a step: On the semantics of Statecharts*. In: *TACS '91: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, Springer-Verlag, London, UK, pp. 244–264.

[24] Marc Pouzet (2006): *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI. Distribution available at: `www.lri.fr/~pouzet/lucid-synchrone`.

[25] Klaus Schneider (2002): *Proving the equivalence of microstep and macrostep semantics*. In: *TPHOLs '02: Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics*, Springer-Verlag, London, UK, pp. 314–331.

[26] S. Tripakis, B. Lickly, Th. A. Henzinger & E. A. Lee (2009): *On relational interfaces*. Technical Report UCB/EECS-2009-60, Electrical Enginering and Computer Sciences, Univ. of California at Berkely.

[27] E. Wandeler & L. Thiele (2005): *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)*.