The Earlier the Better: A Theory of Timed Actor Interfaces

Marc Geilen
Stavros Tripakis
Maarten Wiggers

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-130
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-130.html

October 7, 2010

The Earlier the Better: A Theory of Timed Actor Interfaces *

Marc Geilen Stavros Tripakis Maarten Wiggers

Abstract

Programming embedded and cyber-physical systems requires attention not only to functional behavior and correctness, but also to non-functional aspects and specifically timing and performance constraints. A structured, compositional, model-based approach based on stepwise refinement and abstraction techniques can support the development process, increase its quality and reduce development time through automation of synthesis, analysis or verification. For this purpose, we introduce in this paper a general theory of timed actor interfaces. Our theory supports a notion of refinement that is based on the principle of worst-case design that permeates the world of performance-critical systems. This is in contrast with the classical behavioral and functional refinements based on restricting or enlarging sets of behaviors. An important feature of our refinement is that it allows time-deterministic abstractions to be made of time-non-deterministic systems, improving efficiency and reducing complexity of formal analysis. We also show how our theory relates to, and can be used to reconcile a number of existing time and performance models and how their established theories can be exploited to represent and analyze interface specifications and refinement steps.

1 Introduction

Advances in sensor and computer hardware currently enable new classes of applications, often described under the terms *embedded* or *cyber-physical systems* (ECPS). Examples of such systems can be found in the domains of robotics, health care, transportation, and energy. ECPS are different from traditional computing systems, because in ECPS the computer is in tight interaction with a physical environment, which it monitors and possibly controls. The requirements of the closed-loop system (computer + environment) are not purely functional in the traditional computer-science view. Instead, they often involve timing or performance properties, such as throughput or latency.

Abstraction and compositionality have been two key principles in developing large and complex software systems. Although a large number of methods employing these principles exist in the literature to deal with functional properties (e.g., see [5, 32, 33, 11]), less attention has been paid to timing and performance. This paper contributes toward filling this gap. Our approach can be termed *model based*. High-level models that are suitable for (often automated) analysis are used as specifications or for design-space exploration. Refinement and abstraction steps are used to move between high-level models, lower-level models and implementations: see Section 2 for an example. The process guarantees that the results of the analysis (e.g., bounds on throughput or latency) are preserved during refinement. Our paper defines a general model and a suitable notion of abstraction and refinement that support this process. The model is compositional in the sense that refinement between models consisting of many components can be achieved by refining individual components separately.

Our treatment follows so-called *interface theories* [14], which can be seen as type theories focusing on dynamic and concurrent behavior. Our interfaces, called *actor interfaces*, are inspired by *actor-oriented* models of computation [1, 29] such as process networks [25] and data flow [17].

*Marc Geilen is with the Eindhoven University of Technology, m.c.w.geilen@tue.nl. Stavros Tripakis is with the University of California, Berkeley, stavros@eecs.berkeley.edu. Maarten Wiggers is with the University of Twente, m.h.wiggers@utwente.nl.

Actors operate by consuming and producing *tokens* on their input and output ports, respectively. Since our primary goal is timing and performance analysis, we completely abstract away from token values, and keep only the times in which these tokens are produced. Actors are then defined as relations between input and output sequences of discrete events occurring in a given time axis. Examples of such event sequences are shown in Figure 2.

The main novelty of our theory lies in its notion of refinement, which is based on the principle: *the earlier the better*. In particular, actor A refines actor B if, for the same input, A produces no fewer events and no later, in the worst case, than those produced by B. For example, an actor that non-deterministically delays its input by some time $t \in [1, 2]$ refines an actor that deterministically delays its input by a constant time 5. This is in sharp contrast with most standard notions of refinement which rely on the principle that the implementation should have fewer possible behaviors and thus be "more deterministic" (at the outputs) than the specification.

The earlier-is-better refinement principle is interesting because it allows *deterministic abstractions of non-deterministic systems*: Section 2 presents an example. System implementations are often viewed as time-non-deterministic because of high variability in execution and communication delays, dynamic scheduling, and other effects that are expensive or impossible to model precisely. Time-deterministic models, on the other hand, suffer less from state explosion problems, and are also more suitable for deriving analytic bounds.

The main contributions of this work are the following:

- We develop an interface theory of timed actors with a refinement relation that follows the earlier-is-better principle and preserves worst-case bounds on performance metrics (throughput, latency). (Sections 4–7).
- Our framework unifies existing models (SDF and variants, automata, service curves, etc.) by treating actors semantically, as relations on event sequences, rather than syntactically, as defined by specific models such as automata or dataflow. (Section 9).

More specifically, Section 4 introduces actors. Section 5 defines serial, parallel, and feedback composition operators on actors. Section 6 presents refinement and the conditions under which it is preserved by composition. Section 7 defines throughput and latency and shows that worst-case bounds on these metrics are preserved by refinement. We also define a notion of buffer requirements in Section 8 and show that worst-case bounds on these requirements are preserved by refinement. In Section 9, we show examples of models that can be cast into our framework, and provide algorithms for problems such as checking refinement and computing compositions, throughput and latency.

2 Motivating Example

To illustrate the use of our framework, we present an example of an MP3 play-back application. The application is based on a fragment of the car radio case study presented in [45]. Our goal is to show how such an application can be handled within our framework, using stepwise refinement from specification to implementation, such that performance guarantees are preserved during the process.

The layers of the refinement process are shown in Figure 1. The top layer captures the specification. It consists of a single actor SPEC, with a single output port, and a single event sequence τ at this port, defined by $\tau(n) = 50 + {}^{n}/_{44.1}$ ms, for $n \in \mathbb{N}$. That is, the n-th event in the sequence occurs at time $\tau(n)$. SPEC specifies the required behavior of an MP3 player where audio samples are produced at a rate of 44.1 kHz, starting with an initial 50 ms delay.

Note that SPEC has no inputs: for simplicity, we do not model input tokens, assuming they are always available for consumption. Also note that the system typically includes a component such as a digital-to-analog converter (DAC) which consumes the audio samples produced at port p, buffers them and reproduces them periodically. We omit DAC since it does not take part in the refinement process.

The next layer is an application model consisting of actors DEC (decoder), SRC (sample-rate converter), and actor D1 explained below. DEC and SRC are timed *synchronous data flow* (SDF) [28] actors. SDF actors communicate by conceptually unbounded FIFO queues. They "fire" as soon as a fixed number of tokens become available at their input queues and, after a fixed duration, produce a fixed number of tokens

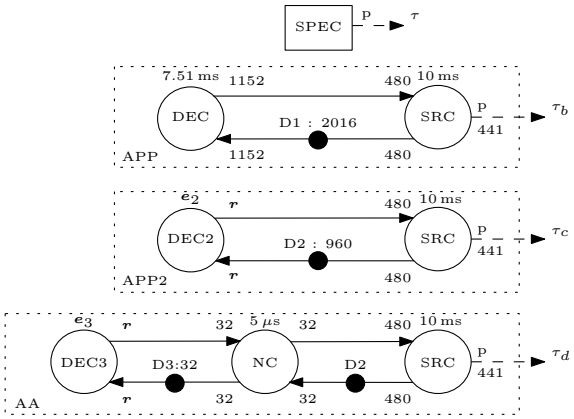


Figure 1: Successive refinements of an MP3 playback application.

at their output queues. For instance, DEC consumes and produces 1152 tokens per firing on the queues from and to the SRC actor. Each firing of DEC takes 7.51 ms. For a formal definition of SDF actors see Section 9.1. D1 is an actor modeling 2016 initial tokens on the queue from SRC to DEC. Formally, it is an instance of parameterized actor I_k defined in Example 17. All dataflow actors in Figure 1 (DEC, SRC, DEC2, DEC3 and NC) implicitly have a self-edge with a single initial token so that firings of the same actor do not overlap (i.e., each firing completes before the next one starts).

The global application model is a single composite actor APP obtained by composing the three actors above, first in sequence and then in feedback, and then hiding all ports except the output port p of SRC. Section 5 precisely defines the compositions and hiding. Because APP is an SDF model and hence deterministic, APP produces a single event sequence τ_b at p. We have captured APP in the dataflow analysis tool SDF3 (`http://www.es.ele.tue.nl/sdf3/`) and have used the tool to check that τ_b refines τ, i.e., that each event in τ_b occurs no later than the corresponding event in τ. As a result, APP refines SPEC.

The motivation for the third layer is buffer considerations. In this layer, the SDF actor DEC is replaced by the *cyclo-static data flow* (CSDF) [7] actor DEC2. This substitution results in smaller buffers on the queue from SRC to DEC2 [45]. CSDF actors generalize SDF actors by allowing the token consumption/production rates to vary periodically, as an SDF actor that cycles between a finite set of firing *phases*. In our example, DEC2 has 39 phases, captured by the notation $\boldsymbol{r} = [0, 0, [32]^{18}, 0, [32]^{18}]$. In the first two phases DEC decodes frame headers without consuming nor producing any tokens. The subsequent 18 phases each consume and produce 32 tokens, and are followed by a header decoding phase with no tokens consumed or produced. Finally there are 18 more phases that each consume and produce 32 tokens. This sequence of phases is repeated for each MP3 frame. The durations of these phases are given by $\boldsymbol{e}_2 = [670, 2700, [40]^{18}, 2700, [40]^{18}]\mu$s. That is, phase 1 takes 670 μs, phase 2 takes 2700 μs, and so on.

Using arguments similar to those presented later in Example 8, we can show that DEC2 refines DEC. The composite actor APP2 is produced by first refining DEC to DEC2, and then reducing the number of initial tokens from D1 to D2, while maintaining that APP2 refines APP. The latter is ensured by using SDF3 to compute τ_c for a given D2, and checking that τ_c refines τ_b.

The bottom layer is an *architecture aware* model (AA) that is close to a distributed implementation on a multiprocessor architecture with network-on-chip (NoC) communication. In this layer DEC2 is replaced by the composition of DEC3, D3 and NC. DEC3 is identical to DEC2 except for its firing durations which are reduced to $\boldsymbol{e}_3 = [670, 2700, [30]^{18}, 2700, [30]^{18}]\mu$s, because the communication is modeled separately. NC is an SDF actor that models the NoC behavior. It can be shown that the composition of DEC3, NC and D3 refines DEC2. This and our compositionality Propositions 5, 6 and 7 imply that AA refines APP2.

The final implementation (not shown in the figure) can be compositionally shown to refine the AA model. For instance, the NC actor conservatively abstracts the NoC implementation [20]. It is important to mention

that although implementations are time-non-deterministic for multiple reasons, e.g., software execution times or run-time scheduling, the models in Figure 1 are time-deterministic.

3 Related Work

Abstraction and compositionality have been extensively studied from an untimed perspective, focusing on functional correctness (e.g., see [5, 32, 33, 11]). Timing has also been considered, implicitly or explicitly, in a number of frameworks. Our treatment has been inspired in particular by interface theories such as interface automata [14]. Although such automata have no explicit notion of time, discrete time can be implicitly modeled by adding a special "tick" output. A synchronous and symbolic version of the theory of [14] is proposed in [42], where discrete time is implicitly measured by synchronous rounds. Our work has been inspired by [42] which also uses a relational framework, although different from the one used in this paper. [15] follows [14] but uses timed automata [2] instead of discrete automata. However, a notion of refinement is not defined in [15]. [13] extends [15] with a notion of refinement in the spirit of alternating simulation [3], adapted for timed systems. Other refinement variants for timed automata include the simulation-relation in [38], the trace-inclusion based refinement in [10] and the one used for conformance testing in [27].

The refinement notions used in all works above differ from ours in a fundamental way: in our case, earlier is better, whereas in the above works, if the implementation can produce an output a at some time t, then the (refined) specification must also be able to produce a at the *same* time t. Thus, an implementation that can produce a only at times $t \leq 1$ does not refine a specification that can produce a only at times $t \geq 2$.Another major difference is that performance metrics such as throughput and latency are not considered in any of the above works.

Our work is about non-deterministic models and worst-case performance bounds and as such differs from probabilistic frameworks such as Markov decision processes, or stochastic process algebras or games (e.g., see [34, 24, 22, 16]). Worst-case performance bounds can be derived using techniques from the *network calculus* (NC) [8] or *real-time calculus* (RTC) [39]. Refinement relations have been considered recently in these frameworks [21, 40]. Semantically, these relations correspond to trace containment at the outputs and as such do not follow the earlier-is-better principle. An important feature of NC and RTC is that they can model resources, e.g., available computation power, and therefore be used in applications such as schedulability analysis. We do not explicitly distinguish resources in our framework. In NC and RTC, behaviors are typically captured by arrival or service curves. These have limited expressiveness compared to our framework where traces can be captured, for instance, by automata. The same can be said of real-time scheduling theory (e.g., see [9]). Automata-based models have been used for scheduling and resource modeling, e.g., as in [44], where tasks are described as ω-regular languages representing sets of admissible schedules. Refinement is not considered in this work, and although it could be defined as language containment, this would not follow the earlier-is-better principle.

$(\max, +)$ algebra and its relatives (e.g., see [4]) are used as an underlying system theory for different discrete event system frameworks, including NC, RTC and SDF. $(\max, +)$ algebra is mostly limited to deterministic, $(\max, +)$-linear systems. Our framework is more general: it can capture non-determinism in time, an essential property in order to be able to relate time-deterministic specification models such as SDF to implementations that have variable timing. Time Petri nets have also been used for performance evaluation although in a non-compositional context (e.g., see [35]).

There are different ways to mathematically model the event sequence abstraction. The event sequences used in this paper can be seen as a special case of the general tagged signals of [30], with signal values abstracted. Note, however, that composition in [30] is intersection, whereas we use a demonic interpretation and also that no specific refinement relation or compositionality guarantees are provided in [30]. As we show in Section 9.2, event sequences can also equivalently be expressed in a dual form as arrival functions [8]. Counter and dater functions are similar structures known from $(\max, +)$ algebra [4].

Our work has also been inspired by the work in [46], where task graph implementations are conservatively abstracted to timed dataflow specifications.

4 Actors

We consider actor interfaces (in short *actors*) as relations between finite or infinite sequences of input tokens and sequences of output tokens. We abstract from token content, and instead focus on arrival or production times represented as timestamps from some totally ordered, continuous time domain $(\mathcal{T}, \leq)$. $\mathcal{T}$ contains a minimal element denoted 0. We also add to $\mathcal{T}$ a maximal element denoted ∞, so that $t < \infty$ for all $t \in \mathcal{T}$. $\mathcal{T}^\infty$ denotes $\mathcal{T} \cup \{\infty\}$. $\mathbb{N}$ denotes the set of natural numbers and we assume $0 \in \mathbb{N}$. $\mathbb{R}$ denotes the set of real numbers and $\mathbb{R}^{\geq 0}$ the set of non-negative reals.

Definition 1 (Event sequences)**.** *An* event sequence *is a total mapping $\tau : \mathbb{N} \to \mathcal{T}^\infty$, such that τ is weakly monotone, that is, for every $k, m \in \mathbb{N}$ and $k \leq m$ we have $\tau(k) \leq \tau(m)$.*

$\tau(n) = \infty$ is interpreted as event n being *absent*. Then, monotonicity of τ and maximality of ∞ implies that all events $n' > n$ are also absent. Because of this property, an event sequence τ can also be viewed as a finite or infinite sequence of timestamps in $\mathcal{T}$. The *length* of τ, denoted $|\tau|$, is the smallest $n \in \mathbb{N}$ such that $\tau(n) = \infty$, and with somewhat abusive notation $|\tau| = \infty$ if $\tau(n) < \infty$ for all $n \in \mathbb{N}$. If $|\tau| = \infty$ then τ is infinite, otherwise it is finite. We use ϵ to denote the *empty* event sequence, $\epsilon(n) = \infty$ for all n. Given event sequence τ and timestamp $t \in \mathcal{T}$ such that $t \leq \tau(0)$, $t \cdot \tau$ denotes the event sequence consisting of t followed by τ. The set of all event sequences is Tr.

Event sequences are communicated over *ports*. For a set P of ports, $Tr(P)$ denotes $P \to Tr$, the set of total functions that map each port of P to an event sequence. Elements of $Tr(P)$ are called *event traces over P*. For $x \in Tr(P)$, we sometimes use the notation $(p, n, t) \in x$ instead of $x(p)(n) = t$.

Definition 2 (Earlier-than and prefix orders)**.** *For $\tau, \tau' \in Tr$, τ is said to be* earlier than *τ', denoted $\tau \leq \tau'$, iff $|\tau| = |\tau'|$ and for all $n < |\tau|$, $\tau(n) \leq \tau'(n)$. $\leq$ is called the* earlier-than *relation. In addition we consider the* prefix *relation: $\tau \preceq \tau'$ iff $|\tau| \leq |\tau'|$ and for every $n < |\tau|$, $\tau(n) = \tau'(n)$. We lift $\leq$ and $\preceq$ to event traces $x, x' \in Tr(P)$ in the usual way: $x \leq x'$ iff for all $p \in P$, $x(p) \leq x'(p)$; $x \preceq x'$ iff for all $p \in P$, $x(p) \preceq x'(p)$.*

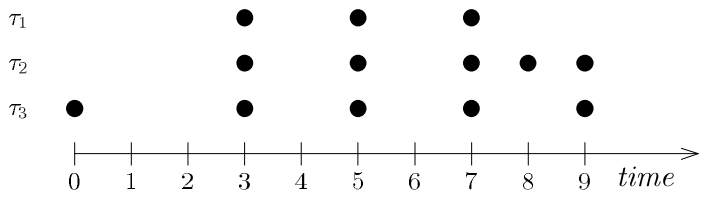


Figure 2: Three event sequences.

Example 1. *Figure 2 shows three event sequences τ_1, τ_2 and τ_3 visualized as black dots on a horizontal time line: $\tau_1 = 3 \cdot 5 \cdot 7 \cdot \epsilon$, $\tau_2 = 3 \cdot 5 \cdot 7 \cdot 8 \cdot 9 \cdot \epsilon$ and $\tau_3 = 0 \cdot 3 \cdot 5 \cdot 7 \cdot 9 \cdot \epsilon$. τ_1 is a prefix of τ_2: $\tau_1 \preceq \tau_2$; and τ_3 is earlier than τ_2: $\tau_3 \leq \tau_2$. But $\tau_1 \not\preceq \tau_2$.*

Recall [12] that a poset (partially ordered set) is a set equipped with a partial order, i.e., a reflexive, transitive, and antisymmetric binary relation. A chain is a countable sequence (possibly infinite) of ordered elements in the set. A complete poset (CPO) is a poset with a least element and such that every chain has a least upper bound. A structure similar to a CPO, but (possibly) without a least element is called a pre-CPO. The least upper bound of chain C in a pre-CPO with order $\sim$ is denoted $\bigsqcup_\sim C$, or just $\bigsqcup C$ if the order is evident.

Lemma 1. *$(Tr, \leq)$ and $(Tr(P), \leq)$ are pre-CPOs. $(Tr, \preceq)$ and $(Tr(P), \preceq)$ are CPOs.*

Proof. It is easy to show that both $\leq$ and $\preceq$ are partial orders on event sequences and traces. The empty sequence ϵ is the least element in $(Tr, \preceq)$. In $(Tr(P), \preceq)$ the least element is the event trace which assigns ϵ

to every $p \in P$. On the other hand, $\leq$ does not have a least element because sequences of different lengths are incomparable w.r.t. $\leq$. The least upper bound of an infinite strictly increasing chain w.r.t. $\preceq$ is the unique infinite sequence whose finite prefixes are prefixes of sequences in the chain. The event sequences in an increasing chain $\{\tau_k\}$ in $(Tr, \leq)$ are all of the same length l and for any $n < l$, $\{\tau_k(n)\}$ is a chain in $(\mathcal{T}^\infty, \leq)$ with a least upper bound t_n (possibly ∞). Then the event sequence τ of length l such that $\tau(n) = t_n$ is the least upper bound of $\{\tau_k\}$. □

If x_1 is an event trace over ports P_1 and x_2 is an event trace over ports P_2, and P_1 and P_2 are disjoint, then $x_1 \cup x_2$ denotes the event trace over $P_1 \cup P_2$ such that $(x_1 \cup x_2)(p) = x_1(p)$ if $p \in P_1$ and $(x_1 \cup x_2)(p) = x_2(p)$ if $p \in P_2$. $x \downarrow Q$ is identical to event trace x, but with the domain restricted to the set of ports Q. $x \uparrow Q$ is identical to x, but with all ports in Q removed from the domain, i.e., if $x \in Tr(P)$ then $x \uparrow Q = x \downarrow (P \setminus Q)$.

Definition 3 (Actors)**.** *An* actor *is a tuple $A = (P, Q, R_A)$ with a set P of input ports, a set Q of output ports and an event trace relation $R_A \subseteq Tr(P) \times Tr(Q)$. We use xAy to denote $(x, y) \in R_A$ when we leave the three-tuple of A implicit.*

Definition 4 (Legal input traces)**.** *For an actor A with input ports P and output ports Q, in_A denotes the set of all* legal *input traces of A: $\mathrm{in}_A = \{x \in Tr(P) \mid \exists y \in Tr(Q) : xAy\}$.*

Definition 5 (Input-closures, monotonicities and continuities)**.** *Let A be an actor with input ports P and output ports Q. A is called* input-complete *iff $\mathrm{in}_A = Tr(P)$. Given a partial order $\trianglelefteq$ on $Tr(P)$ and $Tr(Q)$, A is called (inverse) $\trianglelefteq$-*input-closed *iff for every $x \in \mathrm{in}_A$ and $x' \in Tr(P)$, $x' \trianglelefteq x$ ($x \trianglelefteq x'$) implies $x' \in \mathrm{in}_A$. A is called (inverse) $\trianglelefteq$-monotone iff for every x, y and x' such that xAy, $x' \in \mathrm{in}_A$ and $x \trianglelefteq x'$ ($x' \trianglelefteq x$), there exists y' such that $y \trianglelefteq y'$ ($y' \trianglelefteq y$) and $x'Ay'$. Assuming $\trianglelefteq$ yields pre-CPOs, A is called $\trianglelefteq$-*continuous *iff for every pair $\{x_k\}$ and $\{y_k\}$ of chains of event traces w.r.t. $(Tr(P), \trianglelefteq)$ and $(Tr(Q), \trianglelefteq)$ respectively, if $x_k A y_k$ for all k, then $(\bigsqcup_{\trianglelefteq}\{x_k\})A(\bigsqcup_{\trianglelefteq}\{y_k\})$.*

Definition 6 (Determinism)**.** *An actor $A = (P, Q, R_A)$ is called* deterministic*, if R_A is a partial function.*

Note that a definition of continuity in the reverse prefix order would be equivalent to inverse $\preceq$-monotonicity because prefix is well-founded.

If $x \in Tr(P)$ and $t \in \mathcal{T}$, then $x : t$ denotes the event trace x' such that for every $p \in P$ and $n \in \mathbb{N}$, $x'(p)(n) = x(p)(n)$ if $x(p)(n) \leq t$, and $x'(p)(n) = \infty$ otherwise. In order words, $x : t$ is the restriction of x up to time t.

Definition 7 (Temporal causality)**.** *A $\preceq$-input-closed actor A is called* temporally causal *if for every x, y such that xAy, and for any p and n, if $y(p)(n) = t$ then there exists y' such that $y' \preceq y$, $(x : t)Ay'$ and $y'(p)(n) = t$. A is* strictly temporally causal *if there exists $t' < t$ such that $y' \preceq y$, $(x : t')Ay'$ and $y'(p)(n) = t$.*

In the rest of this section we give some examples of actors. More examples are provided in Section 9.

Example 2 (Delay actors)**.** *A* variable delay *actor $\Delta_{[d_1,d_2]}$ with minimum and maximum delay $d_1, d_2 \in \mathbb{R}^{\geq 0}$, where $d_1 \leq d_2$, is an actor with one input port p, one output port q, time domain $\mathcal{T} = \mathbb{R}^{\geq 0}$, and such that*

$$\begin{aligned} x\Delta_{[d_1,d_2]}y \text{ iff } |x(p)| = |y(q)| \wedge \forall n < |x(p)| : \\ x(p)(n) + d_1 \leq y(q)(n) \leq x(p)(n) + d_2 \\ \wedge\, n > 0 \implies y(q)(n) \geq y(q)(n-1). \end{aligned}$$

$\Delta_{[d_1,d_2]}$ is input-complete, $\preceq$- and $\leq$-monotone in both directions, and $\preceq$- and $\leq$-continuous, but not deterministic in general. The constant delay *actor Δ_d is the deterministic variable delay actor $\Delta_{[d,d]}$. We show that the constant delay actor is $\leq$-continuous. Let $\{x_k\}$ and $\{y_k\}$ be chains as in the definition of $\leq$-continuity, with least upper bounds x and y respectively and let y' such that $x\Delta_d y'$. Then, if $x_k(p)(n) < \infty$ for all k, $y(q)(n) = \bigsqcup\{y_k(q)(n)\} = \bigsqcup\{x_k(p)(n)+d\} = \bigsqcup\{x_k(p)(n)\}+d = x(p)(n)+d = y'(q)(n)$. Otherwise, if $x_m(p)(n) = \infty$ for some m (and thus $x_k(p)(n) = \infty$ for all $k \geq m$), then $y_k(q)(n) = \infty$ for all $k \geq m$ and $y(q)(n) = \infty = y'(q)(n)$. Thus $y'(q)(n) = y(q)(n)$ for all n, $y'(p) = y(q)$ and $y' = y$.*

Example 3 (Merge actor)**.** *A merge actor G is a deterministic actor with two input ports p_1 and p_2 and one output port q in the time domain $\mathcal{T} = \mathbb{R}^{\geq 0}$ such that*

$$xGy \text{ iff } y(q) = x(p_1)|x(p_2)$$

where a merge operator $|$ on event sequences is defined as the continuous extension of the following inductive definition for finite event sequences:

$$\tau_1|\tau_2 = \begin{cases} \tau_2 & \text{if } \tau_1 = \epsilon \\ \tau_1 & \text{if } \tau_2 = \epsilon \\ t_1 \cdot (\tau_1'|\tau_2) & \text{if } \tau_1 = t_1 \cdot \tau_1', \tau_2 = t_2 \cdot \tau_2', t_1 \leq t_2 \\ t_2 \cdot (\tau_1|\tau_2') & \text{if } \tau_1 = t_1 \cdot \tau_1', \tau_2 = t_2 \cdot \tau_2', t_2 \leq t_1 \end{cases}$$

G is not $\preceq$-monotone in either direction, because an event added to or removed from the end of the sequence of one port, may be inserted at or removed from the middle of the output trace. G is $\leq$-monotone in both directions and temporally causal.

We show that G is $\leq$-monotone. Let $x_1 \leq x_2$ and let $x_1 G y_1$ and $x_2 G y_2$, then $y_1(q) = x_1(p_1)|x_1(p_2)$ and $y_2(q) = x_2(p_1)|x_2(p_2)$. $|y_i(q)| = |x_i(p_1)| + |x_i(p_2)|$ if both are finite and $|y_i(q)| = \infty$ otherwise, so clearly $|y_1(q)| = |y_2(q)|$. We only need to show further that for all $n < |y_1(q)|$, $y_1(q)(n) \leq y_2(q)(n)$. By definition of the merge operator, $x_1(p_1)$ and $x_1(p_2)$ together have at most n events labeled strictly smaller than $y_1(n)$. Because $x_1 \leq x_2$, $x_2(p_1)$ and $x_2(p_2)$ together have at also most n events labeled strictly smaller than $y_1(n)$. Hence, $y_1(q)(n) \leq y_2(q)(n)$. Inverse $\leq$-monotonicity is proven in a similar fashion.

Example 4 (Temporal anomaly)**.** *Actor H is an actor with one input port p and one output port q in the time domain $\mathcal{T} = \mathbb{R}^{\geq 0}$ such that*

$$xHy \text{ iff } \forall n < |x(p)| : \begin{array}{l} 2n \leq x(p)(n) < 2n+2 \text{ and} \\ y(n) = x(n) + 2 \text{ if } 2n \leq x(n) < 2n+1, \\ y(n) = x(n) \text{ if } 2n+1 \leq x(n) < 2n+2 \end{array}$$

H is inverse $\preceq$-monotone, $\leq$-monotone in both directions, temporally causal (not strictly).

Example 5 (Timeout actor)**.** *Actor T_d is an actor with one input port p and one output port q in the time domain $\mathcal{T} = \mathbb{R}^{\geq 0}$ such that*

$$xT_d y \quad \text{iff} \quad \forall n : y(q)(n) = x(p)(n) \text{ if } x(p)(n) \leq d \text{ and } y(q)(n) = \infty \text{ otherwise.}$$

If event n does not arrive before time d, discard it otherwise output. A timeout actor is $\preceq$-monotone in both directions and $\preceq$-continuous. It is inverse $\leq$-monotone, but not $\leq$-monotone and not $\leq$-continuous.

5 Compositions

Actor interfaces can be composed to yield new actor interfaces. The composition operators defined in this paper are illustrated in Figure 3. Parallel composition composes two interfaces side-by-side without interaction:

Definition 8 (Parallel composition)**.** *Let A and B be two actors with disjoint input ports P_A and P_B and disjoint output ports Q_A and Q_B respectively. Then the* parallel composition *$A||B$ is an actor with input ports $P_A \cup P_B$, output ports $Q_A \cup Q_B$, and relation $A||B = \{(x_1 \cup x_2, y_1 \cup y_2) \mid x_1 A y_1 \wedge x_2 B y_2\}$.*

Parallel composition is clearly associative and commutative. It is also easy to see that it preserves all monotonicity, continuity and closure properties if both actors have them.

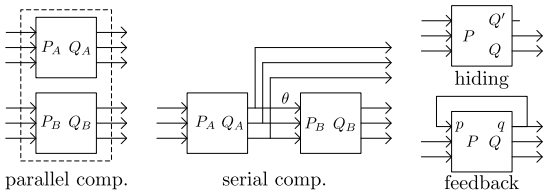


Figure 3: Actor compositions.

Definition 9 (Serial composition)**.** *Let A and B be two actors with disjoint input ports P_A and P_B and disjoint output ports Q_A and Q_B respectively. Let θ be a bijective function from Q_A to P_B. Then the* serial composition *$A\theta B$ is an actor with input ports $P_\theta = P_A$, output ports $Q_\theta = Q_A \cup Q_B$, and whose relation is defined as follows. First, we lift the mapping of ports to event traces: $\theta(y) = \{(\theta(p), n, t) \mid (p, n, t) \in y\}$. The input-output relation of the composite actor $A\theta B$ is then defined as:*

$$A\theta B = \{(x_1, y_1 \cup y_2) \in \mathrm{in}_\theta \times Tr(Q_\theta) \mid x_1 A y_1 \wedge \theta(y_1) B y_2\}$$

where: $\mathrm{in}_\theta = \{x \in \mathrm{in}_A \mid \forall y_1 : xAy_1 \implies \theta(y_1) \in \mathrm{in}_B\}$.

in_θ captures the set of legal inputs of the composite actor $A\theta B$. In the spirit of [5, 14], we adopt a "demonic" interpretation of non-determinism, where an input x is legal in $A\theta B$ only if *any* intermediate output that the first actor A may produce for x is a legal input (after relabeling) of the second actor B.

Definition 10 (Receptiveness)**.** *Actor B is* receptive *to actor A w.r.t. θ iff $\forall x, y : xAy \implies \theta(y) \in \mathrm{in}_B$.*

An input-complete actor is clearly receptive to any other actor. If B is receptive to A or A is deterministic, then $A\theta B$ reduces to standard composition of relations. Moreover, if both A and B are deterministic (respectively, input-complete) then so is $A\theta B$.

Note that the requirement that θ is total and onto is not restrictive. For example, suppose A has two output ports q_1, q_2 and B has two input ports p_1, p_2, but we only want to connect q_1 to p_1. To do this, we can extend A with additional input and output ports p_{p_2} and q_{p_2}, respectively, corresponding to p_2. A acts as the identity function on p_{p_2} and q_{p_2}, that is, for all x, y such that xAy, $y(q_{p_2}) = x(p_{p_2})$. Then we can connect q_{p_2} to p_2. Similarly, we can extend B with additional input and output ports p_{q_2} and q_{q_2}, and connect q_2 to p_{q_2}.

A hiding operator can be used to make internal event sequences unobservable.

Definition 11 (Hiding)**.** *Let $A = (P, Q, R_A)$ be an actor and let $Q' \subseteq Q$. The* hiding of Q' in A *is the actor*

$$A\backslash Q' = (P, Q\backslash Q', \{(x, y \uparrow Q') \mid xAy\}).$$

Note that $\mathrm{in}_{A\backslash Q'} = \mathrm{in}_A$. Hiding preserves all forms of monotonicity and continuity, as well as determinism.

Definition 12 (Feedback)**.** *Let $A(P, Q, R_A)$ be an actor and let $p \in P$ and $q \in Q$. The* feedback composition of A on (p, q) *is the actor*

$$A(p = q) = (P\backslash\{p\}, Q, \{(x \uparrow \{p\}, y) \mid xAy \wedge x(p) = y(q)\}).$$

It is well-known from the study of systems with feedback that the behavior of such a system may not be unique, even if the system is deterministic, or that the behavior may not be constructively computable from the behavior of the actor, depending on the nature of the actor[23]. To effectively apply feedback we will typically require additional constraints on the actor. In the following proposition we describe a case in which a solution can be constructively characterized by a method reminiscent of those used in Kahn Process Networks (KPN) [25]. Our result can also handle non-deterministic actors, however. See also the related Proposition 6.

Proposition 1. *If actor A is input-complete, $\preceq$-monotone and $\preceq$-continuous, then $A(p = q)$ is input-complete, $\preceq$-monotone and $\preceq$-continuous.*

Proof. Let $A = (P, Q, R_A)$, with $p \in P$ and $q \in Q$. If x is an event trace and p a port of x, then $x[p \to \tau]$ denotes the event trace obtained from x by setting the sequence at p to τ and leaving the sequences at all other ports unchanged.

(input-completeness) Let $x \in Tr(P \backslash \{p\})$, then by input-completeness, $x_0 = x[p \to \epsilon] \in \text{in}_A$. Hence there is some y_0 such that $x_0 A y_0$. Now let $x_1 = x[p \to y_0(q)]$. Clearly $x_0 \preceq x_1$. Further by $\preceq$-continuity, there exists y_1 such that $x_1 A y_1$ and $y_0 \preceq y_1$. Repeating the procedure with $x_{k+1} = x[p \to y_k(q)]$ we create two chains $\{x_k\}$ and $\{y_k\}$ in the prefix CPO, such that for all k, $x_k A y_k$. Let $x' = \bigsqcup_{\preceq}\{x_k\}$ and $y' = \bigsqcup_{\preceq}\{y_k\}$, then by construction of the chains and by $\preceq$-continuity, respectively $x'(p) = y'(q)$ and $x' A y'$. Therefore, $x' \uparrow \{p\} = x \in \text{in}_{A(p=q)}$. Thus, $A(p = q)$ is input-complete.

($\preceq$-monotonicity) Let $x_a, x_b \in Tr(P \backslash \{p\})$, $x_a \preceq x_b$ and $x'_a A y'_a$ such that $x'_a \uparrow \{p\} = x_a$ and $x'_a(p) = y'_a(q)$. That is, $x_a A(p = q) y'_a$. Let $x'_{b,0} = x_b[p \to y'_a(q)]$. Then $x'_a \preceq x'_{b,0}$. By $\preceq$-monotonicity and input-completeness of A, there exists $y'_{b,0}$ such that $x'_{b,0} A y'_{b,0}$ and $y'_a \preceq y'_{b,0}$. We repeat the construction as follows. For $k > 0$, let $x'_{b,k} = x'_{b,k-1}[p \to y'_{b,k-1}(q)]$. Then $x'_{b,k-1} \preceq x'_{b,k}$. By $\preceq$-monotonicity and input-completeness of A, there exists $y'_{b,k}$ such that $x'_{b,k} A y'_{b,k}$ and $y'_{b,k-1} \preceq y'_{b,k}$. It follows inductively that for all k, $x'_{b,k} \uparrow \{p\} = x_b$, $y'_a \preceq y'_{b,k}$ and $x'_{b,k} A y'_{b,k}$. If we let x'_b and y'_b be the least upper bounds of the chains $\{x'_{b,k}\}$ and $\{y'_{b,k}\}$ respectively, then $x'_b \uparrow \{p\} = x_b$, $x'_b(p) = y'_b(q)$ and $y'_a \preceq y'_b$. Moreover, by $\preceq$-continuity of A, $x'_b A y'_b$. Thus, $x_b A(p = q) y'_b$, and we have shown $\preceq$-monotonicity of $A(p = q)$.

($\preceq$-continuity) Proving preservation of least upper bounds is straightforward. Assume we have two chains of inputs and outputs of $A(p = q)$, $\{x_k\}$ and $\{y_k\}$ respectively, such that $x_k A(p = q) y_k$. Then there exist $\{x'_k\}$ such that $x_k = x'_k \uparrow \{p\}$, $x'_k A y_k$ with $x'_k(p) = y_k(q)$. From this it follows that also $\{x'_k\}$ is a chain in the prefix CPO. Thus we can apply event-continuity of A and obtain that with $x' = \bigsqcup_{\preceq}\{x'_k\}$ and $y = \bigsqcup_{\preceq}\{y_k\}$, we have $x' A y$ and $x'(p) = y(q)$. Moreover, by construction, $x' \uparrow \{p\} = \bigsqcup_{\preceq}\{x_k\}$ and thus $(\bigsqcup_{\preceq}\{x_k\}) A(p = q) (\bigsqcup_{\preceq}\{y_k\})$. □

The assumptions used in the above result may appear strong at first sight. Note, however, that similar assumptions are often used in fixpoint theorems, even for deterministic systems. Although we could have restricted our attention to actors that have such properties by definition, we chose not to do so, since one of our goals is to be as general as possible and to examine the required assumptions on a case-by-case basis. Note that some actor formalisms (e.g., SDF) ensure these properties by definition, however, other formalisms (e.g., automata) don't.

Let us look at some examples of actors in feedback.

Example 6 (Identity actor in feedback)**.** *The* identity *actor $I_{p,q}$ with a single input port p and a single output port q is such that $x I_{p,q} y$ iff $x(p) = y(q)$. $I_{p,q}(p = q)$ has no input ports, a single output port q, and is such that $I_{p,q}(p = q) = Tr(\{q\})$, i.e., it can non-deterministically produce any event sequence on its output q.*

Notice that, since $I_{p,q}$ is deterministic, the above example also shows that determinism is not preserved by feedback.

Example 7 (Constant delay actor in feedback)**.** *Consider the constant delay actor Δ_d from Example 2. For $d = 0$, the actor behaves like the identity actor, therefore, $\Delta_0(p = q) = I_{p,q}(p = q) = Tr(\{q\})$. On the other hand, for any $d > 0$, $\Delta_d(p = q) = \{\epsilon\}$, since only the empty sequence satisfies the condition $\forall n < |\tau| : \tau(n) = \tau(n) + d$.*

A case of non-deterministic actor in feedback is provided in Example 10.

Proposition 2. *Serial composition is associative. Feedback is commutative.*

Proof. (associativity of serial composition) We have to show that $A\theta_1(B\theta_2 C) = (A\theta_1 B)\theta_2 C$. Let $x A\theta_1(B\theta_2 C) y$. Then there exist y_A, x_B, y_B, x_C such that $x A y_A$, $x_B B y_B$, $x_C C y$, $\theta_1(y_A) = x_B$ and $\theta_2(y_B) = x_C$. Because of the demonic non-determinism input restrictions, we need to show that (i) for all y'_A such that $x A y'_A$,

$\theta_1(y'_A) \in \text{in}_B$ and (ii) for all y'_B such that $xA\theta_1By'_B$, $\theta_2(y'_B) \in \text{in}_C$. Ad (i), we know that for all such y'_A, $\theta_1(y'_A) \in \text{in}_{B\theta_2C} \subseteq \text{in}_B$. Ad (ii), let y'_B be such and let y'_A and x'_B such that xAy'_A, $x'_BBy'_B$ and $\theta_1(y'_A) = x'_B$. Then $x'_B \in \text{in}_{B\theta_2C}$ and therefore, $\theta_2(y'_B) \in \text{in}_C$. Thus we may conclude that $x(A\theta_1B)\theta_2Cy$.

Conversely, let $x(A\theta_1B)\theta_2Cy$. Then again there exist y_A, x_B, y_B, x_C such that xAy_A, x_BBy_B, x_CCy, $\theta_1(y_A) = x_B$ and $\theta_2(y_B) = x_C$. However, we have different input conditions to check. We need to show that (i) for all y'_A such that xAy'_A, $\theta_1(y'_A) \in \text{in}_{B\theta_2C}$ and (ii) for all y'_B such that $x_BBy'_B$, $\theta_2(y'_B) \in \text{in}_C$. Ad (i), we know that for all such y'_A, $\theta_1(y'_A) \in \text{in}_B$ and that for all y'_B such that $xA\theta_1By'_B$, $\theta_2(y'_B) \in \text{in}_C$. Let y''_B be such that $\theta_1(y'_A)By''_B$, then $xA\theta_1By''_B$ and hence $\theta_2(y''_B) \in \text{in}_C$ and thus $y'_A \in \text{in}_{B\theta_2C}$. (ii) follows directly from the fact that $xA\theta_1By'_B$. Thus we conclude also that $xA\theta_1(B\theta_2C)y$.

(commutativity of feedback) Let $A = (P, Q, R_A)$ be an actor and let $p_1, p_2 \in P$ and $q_1, q_2 \in Q$. Let $xA(p_1 = q_1)(p_2 = q_2)y$. Then with $x_2 = x[p_2 \to y(q_2)]$, $x_2A(p_1 = q_1)y$ and with $x_{1,2} = x_2[p_1 \to y(q_1)] = x[p_1 \to y(q_1), p_2 \to y(q_2)]$, $x_{1,2}Ay$. Since $x_{1,2}(p_2) = y(q_2)$, with $x_1 = x_{1,2} \uparrow \{p_2\}$, $x_1A(p_2 = q_2)y$. Finally, since $x_1(p_1) = y(q_1)$ and $x_1 \uparrow \{p_1\} = x$, we have that $xA(p_2 = q_2)(p_1 = q_1)y$. □

Let K be a set of feedback connections, $K = \{(p_1, q_1), ..., (p_n, q_n)\}$. Let A be an actor with input ports P and output ports Q, such that for all $i = 1, ..., n$, $p_i \in P$ and $q_i \in Q$. We denote by $K(A)$ the interface $A(p_1 = q_1) \cdots (p_n = q_n)$. By commutativity of feedback, the resulting actor is independent from the order of application of feedback connections.

Note that, contrary to what one might expect, serial composition is not equivalent to parallel composition followed by feedback. The reason for this is the demonic interpretation of non-determinism in serial composition. Serial composition $A\theta B$ can, however, be expressed by parallel composition and feedback if B is receptive to A w.r.t. θ:

Proposition 3. *Let A and B be actors with disjoint input ports P_A and P_B and disjoint output ports Q_A and Q_B respectively. Suppose B is receptive to A w.r.t. θ. Let $Q_A = \{q_1, \ldots, q_n\}$ and $P_B = \{p_1, \ldots, p_n\}$. Let $K = \{(p_1, q_1), \ldots, (p_n, q_n)\}$. Let θ be the bijection from Q_A to P_B with $\theta(q_i) = p_i$. Then $A\theta B = K(A||B)$.*

Proof. Let $xA\theta By$. Let xAy_A, $\theta(y_A) = x_B$ and x_BBy. Then $(x \cup x_B)A||B(y_A \cup y)$ and for all $q_i \in Q_A$, $(x \cup x_B)(\theta(q_i)) = (x \cup x_B)(p_i) = (y_A \cup y)(q_i)$. Thus, $xK(A||B)y$. Conversely, if $xK(A||B)y$, then there exists x' such that $x' \uparrow P_B = x$, $x'A||By$ and for all $1 \leq k \leq n$, $y(q_k) = x'(p_k)$. Then $xA(y \downarrow Q_A)$ and $\theta(y \downarrow Q_A)B(y \downarrow Q_B)$. Because actor B is input complete, $xA\theta By$. □

6 Refinement

Refinement is a relation between two actors A and B, allowing one to replace actor A by actor B in a given context and obtain "same or better" results, in the worst case. If τ_A and τ_B are event sequences produced by A and B, respectively, then "τ_B is same or better than τ_A" means the following: τ_B should have at least as many events as τ_A and for every event they have in common, the event should be produced in τ_B no later than in τ_A. We first capture this relation on event sequences and event traces.

Definition 13 (Refinement on event sequences and event traces)**.** *Event sequence τ* refines *event sequence τ', denoted $\tau \sqsubseteq \tau'$, iff for all $n \in \mathbb{N}$, $\tau(n) \leq \tau'(n)$. $\sqsubseteq$ is lifted to event traces $x, x' \in Tr(P)$ in the standard way: $x \sqsubseteq x'$ iff for all $p \in P$, $x(p) \sqsubseteq x'(p)$.*

For example, for the event sequences shown in Figure 2, we have $\tau_3 \sqsubseteq \tau_2$, $\tau_2 \sqsubseteq \tau_1$, but $\tau_1 \not\sqsubseteq \tau_2$.

Lemma 2. *The refinement relations on event sequences and event traces are partial orders, i.e., reflexive, transitive and antisymmetric.*

Proof. Straightforward. □

Lemma 3. *The set of traces $(Tr(P), \sqsubseteq)$ equipped with the refinement order is a lattice. The supremum and infimum of traces is the point-wise supremum and infimum respectively. The event sequence $\vec{0}$, defined by $\vec{0}(n) = 0$ for all $n \in \mathbb{N}$, is the least element. The empty sequence ϵ, defined by $\epsilon(n) = \infty$ for all $n \in \mathbb{N}$, is the greatest element.*

Proof. $x_1 \sqsubseteq x_2$ iff for all $p \in P$ and $n \in \mathbb{N}$, $x_1(p)(n) \leq x_2(p)(n)$. Because $(\mathcal{T}^\infty, \leq)$ is a lattice with min and max and the (infinite) point-wise product of lattices is itself a lattice, the result follows. □

Lemma 4. *For all $x, x' \in Tr(P)$: (1) $x' \preceq x$ implies $x \sqsubseteq x'$; (2) $x \leq x'$ implies $x \sqsubseteq x'$; (3) $x \sqsubseteq x'$ iff there exists $x'' \in Tr(P)$ such that $x'' \preceq x$ and $x'' \leq x'$; (4) $x \sqsubseteq x'$ implies $\forall p \in P : |x'(p)| \leq |x(p)|$; (5) if both x and x' are infinite, then $x \sqsubseteq x'$ iff $x \leq x'$.*

Proof. (1) and (2) follow easily from the definitions and the fact that $t \leq \infty$ holds for any $t \in \mathcal{T}$. The 'if' direction of (3) follows from (1) and (2), and the fact that $\sqsubseteq$ is transitive (Lemma 2). For the 'only if' direction of (3), we define x'' such that for all $p \in P$, $|x''(p)| = |x'(p)|$ and $\forall n < |x''(p)| : x''(p)(n) = x(p)(n)$. We claim that $x'' \preceq x$ and $x'' \leq x'$. Consider some $p \in P$, and let $\tau = x(p)$, $\tau' = x'(p)$ and $\tau'' = x''(p)$. We need to show that $\tau'' \preceq \tau$ and $\tau'' \leq \tau'$. The latter follows by definition of x'' and $\tau \sqsubseteq \tau'$. To show $\tau'' \preceq \tau$, we distinguish cases:

(a) $|\tau| < |\tau'|$: this implies that for some $n \in \mathbb{N}$ we have $\tau(n) = \infty$ but $\tau'(n) \neq \infty$. Then $x(p)(n) > x'(p)(n)$, which contradicts the hypothesis $x \sqsubseteq x'$. Thus, this case is not possible.

(b) $|\tau| \geq |\tau'|$: then $|\tau| \geq |\tau''|$, and by definition of x'', $\forall n < |\tau''| : \tau''(n) = \tau(n)$, therefore, $\tau'' \preceq \tau$.

(4) follows from (3) and the facts $|x''(p)| \leq |x(p)|$ and $|x''(p)| = |x'(p)|$.

(5) follows from (2) and (3), and the fact that of x and x' are infinite, then $x'' = x$. □

Knowing what refinement of event traces means, we can now define a refinement relation on actors.

Definition 14 (Actor refinement)**.** *Let $A = (P, Q, R_A)$ and $B = (P, Q, R_B)$ be actors.* B refines A*, denoted $B \sqsubseteq A$, iff (1) $\text{in}_A \subseteq \text{in}_B$; and (2) $\forall x \in \text{in}_A, \forall y : xBy \implies \exists y' : y \sqsubseteq y' \wedge xAy'$.*

Condition (1) states that for actor B to refine actor A, B must accept at least all the inputs that actor A accepts. Condition (2) states that any behavior of actor B is no worse than the worst-case behavior of A on the same input. Note that the standard notion of refinement, implementing the "more output deterministic" principle, amounts to using the stronger constraint $y = y'$ instead of $y \sqsubseteq y'$ in Condition (2).

The requirement that both A and B have the same sets of input and output ports is not restrictive. Every output port of A (resp. input port of B) must also be an output port of B (resp. input port of A): otherwise replacing A by B in certain contexts may result in open inputs. Any output port of B (resp. input port of A) not originally in A (resp. B) can be added to it as a "dummy" port.

A conditional notion of refinement is often useful, where in both Conditions (1) and (2) in_A is replaced by $\text{in}_A \cap X$, for some given (typically $\sqsubseteq$-closed) set of input traces $X \subseteq Tr(P)$. X models assumptions on the inputs of A and B. This conditional refinement relation is denoted by $B \sqsubseteq_X A$.

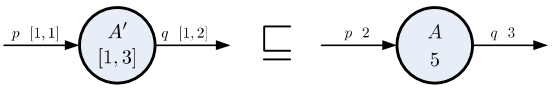


Figure 4: SDF actor A refined by CSDF actor A'.

Example 8 (CSDF actor refining SDF actor)**.** *Figure 4 shows an SDF actor A refined by a CSDF actor A'. At each firing, which takes 5 time units to complete, A consumes 2 and produces 3 tokens. A' cycles between two firing phases: in the first, which takes 1 time unit, 1 token is consumed and 1 is produced; in the second, which takes 3 time units, 1 token is consumed and 2 are produced. We observe that for the same number of input tokens, A' produces no fewer (and sometimes strictly more) output tokens than A, because A' can fire on a single input token, whereas A requires two. Moreover, because of the earlier activation, as well as the shorter processing time, A' produces outputs no later than A. Therefore, A' refines A.*

It is worth noting that the refinement in the above example would not hold had we used $y \leq y'$ instead of $y \sqsubseteq y'$ in Definition 14. This is because, for the input sequence containing a single token, A' produces strictly more tokens than A. As the example of Section 2 shows, it is important to be able to replace SDF

actors by CSDF actors in applications, which partly motivated our definition of refinement. Additional motivation for refining SDF to CSDF or other less "monolithic" models is provided in [41], in the context of modular code generation.

Proposition 4. *The refinement relation on actors is a pre-order, i.e. it is reflexive and transitive .*

Proof. Straightforward, using Lemma 2. □

The refinement relation on actors is not antisymmetric, as the following example shows.

Example 9 (Refinement not antisymmetric)**.** *Consider the constant delay actor Δ_d and the variable delay actor $\Delta_{[d_1,d_2]}$, introduced in Example 2. We show that if $d \leq d_2$, then $\Delta_d \sqsubseteq \Delta_{[d_1,d_2]}$: (1) $\text{in}_{\Delta_{[d_1,d_2]}} \subseteq \text{in}_{\Delta_d}$, because both are input-complete. (2) Let $x \in \text{in}_{\Delta_{[d_1,d_2]}}$ and $x\Delta_d y$. Then $|x(p)| = |y(q)|$ and for all $n < |x(p)|$, $y(q)(n) = x(p)(n) + d$. Let y' be defined as follows. $|y'(q)| = |y(q)|$ and $y'(q)(n) = x(p)(n) + d_2$, then $y \sqsubseteq y'$ and $x\Delta_{[d_1,d_2]}y'$.*

Similarly, if $d_2 \leq d$, then $\Delta_{[d_1,d_2]} \sqsubseteq \Delta_d$, therefore, if $d = d_2$, then the two actors refine each other. But the two actors are generally not equal. In particular, if $d_1 < d_2$, then $\Delta_{[d_1,d_2]}$ is a non-deterministic actor and contains pairs (x, y) not contained in Δ_{d_2} (those for which $y(q)(n) < x(p)(n) + d_2$ for some n).

Lemma 5. *(1) A is $\leq$-input-closed and inverse $\preceq$-input-closed iff A is $\sqsubseteq$-input-closed. (2) A is inverse $\leq$-input-closed and $\preceq$-input-closed iff A is inverse $\sqsubseteq$-input-closed. (3) If A is inverse $\preceq$-monotone and $\leq$-monotone, then A is $\sqsubseteq$-monotone. (4) If A is $\preceq$-monotone and inverse $\leq$-monotone, then A is inverse $\sqsubseteq$-monotone. (5) If A is $\leq$-continuous, then it is $\sqsubseteq$-continuous.*

Proof. Parts of the proof are omitted. For (1) and (2), the 'if' direction is trivial, because $x_1 \sqsubseteq x_2$ is implied by $x_1 \leq x_2$ and $x_2 \preceq x_1$. (1) For the 'only if' direction, let $x \in \text{in}_A$ and $x' \sqsubseteq x$. By $x' \sqsubseteq x$ and Lemma 4, there exists x'' such that $x'' \preceq x'$ and $x'' \leq x$. $\leq$-input-closure of A, $x \in \text{in}_A$ and $x'' \leq x$, imply $x'' \in \text{in}_A$. The latter, together with $x'' \preceq x'$ and inverse $\preceq$-input-closure of A, imply $x' \in \text{in}_A$.

(3) Let x, y and x' be such that xAy, $x' \in \text{in}_A$ and $x \sqsubseteq x'$. By $x \sqsubseteq x'$ and Lemma 4, there exists x'' such that $x'' \preceq x$ and $x'' \leq x'$. By inverse $\preceq$-monotonicity of A, there exists $y'' \preceq y$ such that $x''Ay''$. By $\leq$-monotonicity of A, there exists y' such that $y'' \leq y'$ and $x'Ay'$. By Lemma 4, $y \sqsubseteq y'$.

(5) Let $\{x_k\}$ and $\{y_k\}$ be chains in the refinement order, i.e., $x_k \sqsubseteq x_{k+1}$ and $y_k \sqsubseteq y_{k+1}$ such that x_kAy_k, for all k. Since the result is trivial for finite chains, assume that the chains are infinite. By Lemma 3, the chains have least upper bounds x and y respectively. For any port p, individual event sequences $x_k(p)$ cannot increase in length: this is because $x_k \sqsubseteq x_{k+1}$ implies $x_k(p) \sqsubseteq x_{k+1}(p)$, and the latter implies $|x_{k+1}(p)| \leq |x_k(p)|$, by Lemma 4. Similarly, for any port q, individual sequences $y_k(q)$ cannot increase in length. Therefore, every individual event sequence on some port p in $x \cup y$ is either of infinite length for all k, or eventually becomes of finite length and then there must be some $m(p)$ such that for all $k \geq m(p)$ its length does not further decrease. Let M be the maximum of $m(p)$ of all ports p for which the length becomes finite, and $M = 0$ if there is no such port. Then the chains $\{x_{M+k} \mid k \geq 0\}$ and $\{y_{M+k} \mid k \geq 0\}$ contain only event sequences of equal length. On event sequences of equal length, $\sqsubseteq$ is equivalent to $\leq$, therefore, the above chains are chains in the pre-CPO $(Tr(P), \leq)$, thus, have least upper bounds x' and y' w.r.t. $\leq$. Because $\leq$ is equivalent to $\sqsubseteq$ in this case, $x' = x$ and $y' = y$. By $\leq$-continuity of A it follows that xAy. □

Refinement is preserved by serial composition under natural conditions, namely, consuming actor B should not refuse better input and should not produce worse output on better input:

Proposition 5. *(I) If $A' \sqsubseteq A$ and B is $\sqsubseteq$-input-closed and $\sqsubseteq$-monotone, then $A'\theta B \sqsubseteq A\theta B$. (II) If $B' \sqsubseteq B$ then $A\theta B' \sqsubseteq A\theta B$.*

Proof. (I) Let $A' \sqsubseteq A$. We need to show that $A'\theta B \sqsubseteq A\theta B$. (1) Let $x \in \text{in}_{A\theta B}$. Then there is some y such that $xA\theta By$, i.e., there exist y_1, x_2 and y such that xAy_1, x_2By and $x_2 = \theta(y_1)$ and $\forall y_1 : xAy_1 \implies \theta(y_1) \in \text{in}_B$. We need to show that there exists y' such that $xA'\theta By'$, i.e., (a) there exist y_1', x_2', y_2' such that $xA'y_1', x_2'By_2', x_2' = \theta(y_1')$ and (b) $\forall y_1 : xA'y_1 \implies \theta(y_1) \in \text{in}_B$.

(a) because $A' \sqsubseteq A$, there is some y_1' such that $xA'y_1'$ and there is some y_1'' such that $y_1' \sqsubseteq y_1''$ and xAy_1''. Thus $\theta(y_1'') \in \text{in}_B$ and by $\sqsubseteq$-input-closure of B we can conclude, because $\theta(y_1') \sqsubseteq \theta(y_1'')$, that there exists y_2' such that $\theta(y_1')By_2'$.

(b) like part (a) the same can be shown for any y_1 such that $xA'y_1$.

Thus, $x \in \text{in}_{A'\theta B}$, which proves the first requirement of refinement $A'\theta B \sqsubseteq A\theta B$.

(2) For the second requirement, let $x \in \text{in}_{A\theta B}$ and y such that $xA'\theta By$. Then there exist y_1, x_2 and y_2 such that $xA'y_1$, x_2By_2, $y = y_1 \cup y_2$ and $x_2 = \theta(y_1)$. Because $A' \sqsubseteq A$ and $x \in \text{in}_A$, there is y_1' such that xAy_1' and $y_1 \sqsubseteq y_1'$, thus also $\theta(y_1) \sqsubseteq \theta(y_1')$. Because $x \in \text{in}_{A\theta B}$, $\theta(y_1') \in \text{in}_B$ and there exists y_2' with $\theta(y_1')By_2'$. By $\sqsubseteq$-monotonicity of B, there exists such a y_2' for which $y_2 \sqsubseteq y_2'$. Thus, with $y' = y_1' \cup y_2'$, $xA\theta By'$ and $y \sqsubseteq y'$.

(II) Let $B' \sqsubseteq B$. We need to show that $A\theta B' \sqsubseteq A\theta B$. (1) Let $x \in \text{in}_{A\theta B}$. Then there exist y_1, x_2 and y_2 such that xAy_1, x_2By_2, $x_2 = \theta(y_1)$ and $\forall y : xAy \implies \theta(y) \in \text{in}_B$. Because $B' \sqsubseteq B$, $\text{in}_B \subseteq \text{in}_{B'}$. Therefore $\theta(y) \in \text{in}_B$ implies $\theta(y) \in \text{in}_{B'}$, and $\forall y : xAy \implies \theta(y) \in \text{in}_{B'}$ also holds. Thus $x \in \text{in}_{A\theta B'}$.

(2) Let $x \in \text{in}_{A\theta B}$ and y such that $xA\theta B'y$. Then there exist y_1, x_2 and y_2 such that xAy_1, $x_2B'y_2$, $y = y_1 \cup y_2$ and $x_2 = \theta(y_1)$. $x_2 \in \text{in}_B$ because $x \in \text{in}_{A\theta B}$. Because $B' \sqsubseteq B$, there is some y_2' such that x_2By_2' and $y_2 \sqsubseteq y_2'$. With $y' = y_1 \cup y_2'$, $y \sqsubseteq y'$ and $xA\theta By'$. □

Feedback preserves refinement under the following conditions:

Proposition 6. *Let A be an inverse $\sqsubseteq$-input-closed, $\sqsubseteq$-monotone and $\sqsubseteq$-continuous actor, and let A' be an input-complete, $\preceq$-monotone and $\preceq$-continuous actor such that $A' \sqsubseteq A$. Then $A'(p = q) \sqsubseteq A(p = q)$.*

Proof. Requirement (1) of refinement $A'(p = q) \sqsubseteq A(p = q)$ follows from the fact that $A'(p = q)$ is input-complete, which follows from Proposition 1.

Requirement (2): Let $x \uparrow p \in \text{in}_{A(p=q)}$ and y' such that $(x \uparrow p, y') \in A'(p = q)$. Then $xA'y'$ and $x(p) = y'(q)$. From $A' \sqsubseteq A$, there exists y such that xAy and $y' \sqsubseteq y$, thus also $x(p) \sqsubseteq y(q)$. However, we do not necessarily have $x(p) = y(q)$, hence, $(x \uparrow p, y)$ is not necessarily a behavior of $A(p = q)$. Nevertheless, we show how we can construct such a behavior. Let $x_0 = x[p \to y(q)]$. Then $x \sqsubseteq x_0$. By $\sqsubseteq$-monotonicity and inverse $\sqsubseteq$-input-closure of A, there exists y_0 such that x_0Ay_0 and $y \sqsubseteq y_0$. Let for $k \geq 0$, $x_{k+1} = x_k[p \to y_k(q)]$ and hence $x_k \sqsubseteq x_{k+1}$, and let y_{k+1} be chosen such that $x_{k+1}Ay_{k+1}$ and $y_k \sqsubseteq y_{k+1}$. Then for any $k \geq 0$, $x \sqsubseteq x_k$ and $y' \sqsubseteq y \sqsubseteq y_k$. By Lemma 3 and $\sqsubseteq$-continuity of A, $x'' = \bigsqcup_{\sqsubseteq}\{x_k\}$ and $y'' = \bigsqcup_{\sqsubseteq}\{y_k\}$, $x''Ay''$, $x'' \uparrow p = x \uparrow p$, $y' \sqsubseteq y''$ and $x''(p) = y''(q)$. □

It is worth noting that Lemma 5 can be used to ensure some of the preconditions of Propositions 5 and 6. For instance, the $\sqsubseteq$-input-closure requirement on B in Proposition 5 can be ensured by showing that B is $\leq$-input-closed and inverse $\preceq$-input-closed, $\sqsubseteq$-monotonicity can be ensured by inverse $\preceq$-monotonicity and $\leq$-monotonicity, $\sqsubseteq$-continuity can be ensured by $\leq$-continuity, and so on.

Example 10. *Consider actors $A = (\{p\}, \{q\}, R_A)$ and $A' = (\{p\}, \{q\}, R_{A'})$ with input-output relations*

$$\begin{aligned} R_A =& \{(x, y) \mid (\forall n : y(q)(n) = x(p)(n) + 2) \vee \\ & (y(q)(0) = 0 \wedge \forall n : y(q)(n + 1) = x(p)(n))\}, \text{ and} \\ R_{A'} =& \{(x, y) \mid y(q)(0) = 0 \wedge \forall n : y(q)(n + 1) = x(p)(n) + 1\}. \end{aligned}$$

Both A and A' are input-complete, $\preceq$-monotone in both directions, $\preceq$-continuous and $\leq$-monotone in both directions. A is non-deterministic but A' is deterministic. A' refines A because the unique output sequence of A' can be matched with the (later) output sequence of A produced by the first disjunct.

If we connect A in feedback, we get $A(p = q)$ with a single (output) port q, and producing either the empty sequence $y(q) = \epsilon$ or the zero sequence $y(q)(n) = 0$ for all n. A' in feedback produces a single sequence $y'(q)(n) = n$. Any sequence refines ϵ, therefore, $A'(p = q) \sqsubseteq A(p = q)$.

It is interesting to see how the fixed-point iteration used in the proof of Proposition 6 evolves. The behavior (x, y') of A' leading to the unique feedback behavior is $x(p)(n) = y'(q)(n) = n$. $A' \sqsubseteq A$ implies that A has a behavior (x, y) such that $y' \sqsubseteq y$, which can only be $y(q)(n) = n + 2$. Putting that in the fixed-point iteration we get that $x_{k+1}(p)(n) = y_k(p)(n) = n + k + 2$. Both chains of event sequences have ϵ as their least upper bound: indeed, ϵ is the only behavior of $A(p = q)$ which is refined by y'.

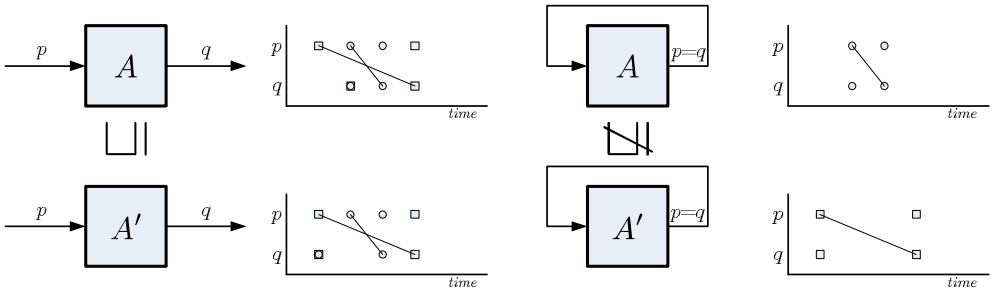


Figure 5: Feedback does not preserve refinement without $\sqsubseteq$-monotonicity.

The following example illustrates why $\sqsubseteq$-monotonicity is needed for refinement to be preserved by feedback (Proposition 6), even when the actors are input-complete, deterministic and $\preceq$-continuous.

Example 11. *Figure 5 depicts two actors A' and A with a single input port p and a single output port q such that $A' \sqsubseteq A$. The graphs shown to the right of the actors display two pairs of sets of event traces on $\{p, q\}$, related by A and A', respectively. One pair is drawn by circles and the other by squares. A line between events indicates a dependency, i.e., the output event is present iff the input event is present. We use these event sequences for illustration. A and A' are input-complete and their complete behavior is defined as follows. A maps any τ such that $1 \cdot \epsilon \preceq \tau$ to $2 \cdot 4 \cdot \epsilon$; any τ such that $2 \cdot \epsilon \preceq \tau$ to $2 \cdot 3 \cdot \epsilon$; and every other τ to $2 \cdot \epsilon$. A' maps any τ such that $1 \cdot \epsilon \preceq \tau$ to $1 \cdot 4 \cdot \epsilon$; any τ such that $2 \cdot \epsilon \preceq \tau$ to $1 \cdot 3 \cdot \epsilon$; and every other τ to $1 \cdot \epsilon$. Observe that A is not $\sqsubseteq$-monotone: indeed, $1 \cdot \epsilon \leq 2 \cdot \epsilon$, thus also $1 \cdot \epsilon \sqsubseteq 2 \cdot \epsilon$, but $2 \cdot 4 \cdot \epsilon \not\leq 2 \cdot 3 \cdot \epsilon$, and because the two sequences have same length, $2 \cdot 4 \cdot \epsilon \not\sqsubseteq 2 \cdot 3 \cdot \epsilon$. It can be checked that $A' \sqsubseteq A$. But when connected in feedback, $A'(p = q) \not\sqsubseteq A(p = q)$: indeed, $A(p = q) = \{2 \cdot 3 \cdot \epsilon\}$ and $A'(p = q) = \{1 \cdot 4 \cdot \epsilon\}$, but $1 \cdot 4 \cdot \epsilon \not\sqsubseteq 2 \cdot 3 \cdot \epsilon$.*

Proposition 7. *Refinement is preserved by hiding.*

Proof. Let $A' \sqsubseteq A$, Q the output ports of A and $Q' \subseteq Q$. Then we need to show that $A'\backslash Q' \sqsubseteq A\backslash Q'$. (1) It is easy to see that $\text{in}_{A\backslash Q'} \subseteq \text{in}_{A'\backslash Q'}$. (2) Let $x \in \text{in}_A$ and y such that $xA'\backslash Q'y$. Then there exists y' such that $y = y' \uparrow Q'$ and $xA'y'$. By refinement of A', there exists y'' such that $y' \sqsubseteq y''$ and xAy''. Now $y \sqsubseteq y'' \uparrow Q'$ and $xA\backslash Q'(y'' \uparrow Q')$. □

7 Performance Metrics

We often care about the performance of our systems in terms of specific metrics such as throughput or latency [8, 39, 31, 19]. In this section we show that our notion of refinement is strong enough to provide guarantees on performance under the refinement process. Throughout this section, we assume that $\mathcal{T} = \mathbb{R}^{\geq 0}$.

We begin by defining throughput for an infinite event sequence τ. A first attempt is to define throughput as the limit behavior of the average number of tokens appearing in the sequence per unit of time: $T(\tau) = \lim_{n\to\infty} \frac{n}{\tau(n)}$. By the usual definition of the limit, it exists and is equal to T if

$$\forall \varepsilon > 0 : \exists K > 0 : \forall n > K : T - \varepsilon < \frac{n}{\tau(n)} < T + \varepsilon.$$

But because this limit may not always exist for a given τ, and because among all possible behaviors of an actor, there may be some for which the limit does not exist, we consider instead throughput *bounds*, which are more robust against such effects.

Definition 15 (Event sequence throughput bounds)**.** *Given infinite event sequence τ, its lower bound on throughput is*

$$T^{lb}(\tau) = \sup\{T \in \mathbb{R}^{\geq 0} \mid \exists K > 0 : \forall n > K : n > \tau(n) \cdot T\}$$

and its upper bound on throughput is

$$T^{ub}(\tau) = \inf\{T \in \mathbb{R}^{\geq 0} \mid \exists K > 0 : \forall n > K : n < \tau(n) \cdot T\}$$

where by convention we take $\sup \mathbb{R}^{\geq 0} = \inf \emptyset = \infty$.

$T^{lb}(\tau)$ is the greatest lower bound on the asymptotic average throughput of τ, and similarly, $T^{ub}(\tau)$ defines the least upper bound. Multiplying both sides of the inequalities by $\tau(n)$ avoids division by zero problems. Note that for a *zeno* sequence τ, where timestamps do not diverge to ∞, i.e., $\exists t \in \mathbb{R}^{\geq 0} : \forall n \in \mathbb{N} : \tau(n) < t$, we have $T^{lb}(\tau) = \sup \mathbb{R}^{\geq 0} = \infty$ and $T^{ub}(\tau) = \inf \emptyset = \infty$. This holds in particular for the zero sequence $\vec{0}$ with $\vec{0}(n) = 0$ for all n.

Proposition 8. *For any two infinite event sequences* τ_1 *and* τ_2*, if* $\tau_1 \sqsubseteq \tau_2$*, then* $T^{lb}(\tau_1) \geq T^{lb}(\tau_2)$ *and* $T^{ub}(\tau_1) \geq T^{ub}(\tau_2)$.

Proof. Both τ_1 and τ_2 are infinite, thus, by Lemma 4, $\tau_1 \sqsubseteq \tau_2$ reduces to $\tau_1 \leq \tau_2$. Thus for any n, $\tau_1(n) \leq \tau_2(n)$, and the result follows by the fact that sup is monotone w.r.t. $\subseteq$ and inf is monotone w.r.t. $\supseteq$. □

We next define the throughput bound for an actor A. An actor may have multiple output ports with generally different throughputs. For a given port, the throughput at that port generally depends on the input trace as well as on non-deterministic choices of the actor. We therefore consider the worst-case scenario.

Definition 16 (Actor throughput lower bound)**.** *For an actor* A *with input ports* P *and output ports* Q*, and given some specific output port* $q \in Q$ *and input trace* $x \in Tr(P)$*, we define the following lower bound on throughput of* A*:*

$$T^{lb}(A, x, q) = \inf\{T^{lb}(\tau) \mid \exists y : xAy \wedge \tau = y(q)\}.$$

We can then define the throughput at all output ports simultaneously as a vector indexed by output ports:

$$T^{lb}(A, x) = \left(T^{lb}(A, x, q)\right)_{q \in Q}.$$

We can similarly define upper bounds $T^{ub}(A, x, q)$ *and* $T^{ub}(A, x)$.

For example, for the actor SPEC of Section 2, which has no inputs and a unique output port, we have $T^{lb}(\text{SPEC}) = T^{lb}(50 + {}^{n}/_{44.1}) = \sup\{T \in \mathbb{R}^{\geq 0} \mid \exists K > 0 : \forall n > K : n > (50 + {}^{n}/_{44.1})T\} = \sup\{T \in \mathbb{R}^{\geq 0} \mid T < 44.1\} = 44.1$.

For two actors A and B with the same sets of input and output ports P and Q, respectively, we shall write $T^{lb}(A, x) \leq T^{lb}(B, x)$ to mean $T^{lb}(A, x, q) \leq T^{lb}(B, x, q)$ for all $q \in Q$. This notation is used in Proposition 10 below.

Example 12. *Consider the constant and variable delay actors* Δ_d *and* $\Delta_{[d_1,d_2]}$ *from Example 2. Both are input-complete and have a single input port* p *and a single output port* q*. Suppose* $d, d_1, d_2 \in \mathbb{R}^{\geq 0}$ *such that* $d > 0$ *and* $d_2 > d_1 > 0$*. Let* x *be the input event trace defined by* $x(p)(n) = n$*, for all* $n \in \mathbb{N}$*. Then there is a single output event trace* y *such that* $x\Delta_d y$*, and* $y(q)(n) = n + d$*, for all* $n \in \mathbb{N}$*. As expected,* $T^{lb}(\Delta_d, x, q) = T^{lb}(y(q)) = T^{ub}(\Delta_d, x, q) = T^{ub}(y(q)) = 1$*. Similarly,* $T^{lb}(\Delta_{[d_1,d_2]}x, q) = T^{ub}(\Delta_{[d_1,d_2]}, x, q) = 1$.

We next turn to latency. Different definitions are possible (e.g., see [31, 19, 43]). We define latency as the smallest upper bound on observed time differences between related input and output events. The pairs of events that we want to relate are explicitly specified as follows:

Definition 17. *An* input-output event specification *(IOES) for a set* P *of input ports and a set* Q *of output ports is a relation* $\mathcal{E} \subseteq 2^{P \times \mathbb{N}} \times 2^{Q \times \mathbb{N}}$*.* $\mathcal{E}$ *is called* valid for $(x, y) \in Tr(P) \times Tr(Q)$ *iff for every* $(E_P, E_Q) \in \mathcal{E}$*, if* $x(p)(m) \neq \infty$ *for every* $(p, m) \in E_P$*, then* $y(q)(n) \neq \infty$ *for every* $(q, n) \in E_Q$*.* $\mathcal{E}$ *is called* valid for an actor $A = (P, Q, R_A)$ *iff it is valid for every* $(x, y) \in R_A$.

A pair $(E_P, E_Q) \in \mathcal{E}$ says that we want to measure the maximum latency between an input event in E_P and an output event in E_Q, provided all events in E_P have arrived. See Example 13, given below, for an illustration.

Definition 18 (Trace latency upper bound)**.** *Let $\mathcal{E}$ be a valid IOES for $(x, y) \in Tr(P) \times Tr(Q)$. We define:*

$$D^{\mathcal{E}}(x, y) = \sup\{y(q)(n) - x(p)(m) \mid (E_P, E_Q) \in \mathcal{E}, E_P \subseteq \mathrm{dom}(x), (p, m) \in E_P, (q, n) \in E_Q\}$$

where by convention $\sup \emptyset = 0$ and $\mathrm{dom}(x)$ denotes the set of all pairs (p, n) such that $x(p)(n) \neq \infty$.

$D^{\mathcal{E}}(x, y)$ is the largest among all delays between an input and an output event that occur in x and y and are related by $\mathcal{E}$, provided all other events in the same input group are also in x. Notice that, by the assumption of validity of $\mathcal{E}$ for (x, y), $E_P \subseteq \mathrm{dom}(x)$ implies $E_Q \subseteq \mathrm{dom}(y)$, for every $(E_P, E_Q) \in \mathcal{E}$.

Example 13. *Consider a deterministic actor A with two input ports p_1, p_2 and a single output port q. Suppose A consumes one token from each input port, and for every such pair, produces a token at q, after some constant delay, say $d \in \mathbb{R}^{\geq 0}$. Let x_1 and x_2 be two input event traces, with $x_1 = \{(p_1, 2{\cdot}\epsilon), (p_2, 4{\cdot}\epsilon)\}$ and $x_2 = \{(p_1, 2{\cdot}5{\cdot}\epsilon), (p_2, 4{\cdot}\epsilon)\}$. For both x_1 and x_2, A produces the same output event trace $y = \{(q, (4{+}d){\cdot}\epsilon)\}$. This is because, in x_2, A waits for a second input to arrive at p_2 before it can produce a second output. To measure the latency of A, we can define $\mathcal{E}$ to be the set*

$$\mathcal{E} = \{(\{(p_1, n), (p_2, n)\}, \{(q, n)\}) \mid n \in \mathbb{N}\}$$

This makes $\mathcal{E}$ a valid IOES for x_1, y, as well as for x_2, y, and gives us $D^{\mathcal{E}}(x_1, y) = D^{\mathcal{E}}(x_2, y) = d$, as is to be expected.

Keeping the reference input trace fixed, refinement of output traces is guaranteed to not worsen latency:

Proposition 9. *Let x, y_1 and y_2 be event traces such that $y_1 \sqsubseteq y_2$. Suppose $\mathcal{E}$ is valid for x and y_2. Then $\mathcal{E}$ is valid for x and y_1 and $D^{\mathcal{E}}(x, y_1) \leq D^{\mathcal{E}}(x, y_2)$.*

Proof. (i) $y_1 \sqsubseteq y_2$ implies $\mathrm{dom}(y_1) \supseteq \mathrm{dom}(y_2)$, i.e., y_1 provides at least as many output events as y_2. Thus, if $\mathcal{E}$ is valid for x and y_2 then it is also valid for x and y_1. (ii) $y_1 \sqsubseteq y_2$ implies $y_1(q)(n) \leq y_2(q)(n)$, therefore, $D^{\mathcal{E}}(x, y_1) = \sup\{y_1(q)(n) - x(p)(m) \mid (E_P, E_Q) \in \mathcal{E}, E_P \subseteq \mathrm{dom}(x), (p, m) \in E_P, (q, n) \in E_Q\} \leq \sup\{y_2(q)(n) - x(p)(m) \mid (E_P, E_Q) \in \mathcal{E}, E_P \subseteq \mathrm{dom}(x), (p, m) \in E_P, (q, n) \in E_Q\} = D^{\mathcal{E}}(x, y_2)$. □

Definition 19 (Actor latency upper bound)**.** *An IOES $\mathcal{E}$ is valid for an actor A iff $\mathcal{E}$ is valid for every (x, y) such that xAy. For a valid $\mathcal{E}$, the worst-case latency of A on input event trace x is*

$$D^{\mathcal{E}}(A, x) = \sup_{y \ s.t. \ xAy} \{D^{\mathcal{E}}(x, y)\}.$$

The worst-case latency of A over all input traces is

$$D^{\mathcal{E}}(A) = \sup_{x \in \mathrm{in}_A} \{D^{\mathcal{E}}(A, x)\}.$$

Example 14. *Consider the variable delay actor $\Delta_{[d_1, d_2]}$ from Example 2. A suitable IOES for its latency would be $\mathcal{E} = \{(\{(p, n)\}, \{(q, n)\}) \mid n \in \mathbb{N}\}$. $\mathcal{E}$ is valid for $\Delta_{[d_1, d_2]}$ and $D^{\mathcal{E}}(\Delta_{[d_1, d_2]}, x) = d_2$, for any non-empty input event trace x.*

The following states the main preservation results for performance bounds under refinement:

Proposition 10. *Let $B \sqsubseteq A$ and $\mathcal{E}$ be a valid IOES for A. Then for any $x \in \mathrm{in}_A$, $T^{lb}(B, x) \geq T^{lb}(A, x)$ and $D^{\mathcal{E}}(B, x) \leq D^{\mathcal{E}}(A, x)$.*

Proof. For any xBy, there is some y' s.t. xAy' and $y \sqsubseteq y'$ and for any p, $y(p) \sqsubseteq y'(p)$ and $T^{lb}(y'(p)) \leq T^{lb}(y(p))$. Therefore, the infimum for A can not be larger than the infimum for B. Similar argument for latency. □

8 Consumption of Tokens and Buffer Capacities

On a queue between two dataflow actors, tokens are buffered until they are consumed. An interesting question is what is the maximum number of buffered tokens on each queue. Although our formalism does not explicitly model events representing token consumption, we can easily capture this aspect of behavior, as follows. We add for a given input port p of an actor A, a corresponding *output consumption port* p'. The events on p' represent the moments when the tokens arriving at p are actually consumed. Since the consumption of a token cannot happen earlier than its production, the following must hold: $\forall x, y : xAy \implies x(p) \sqsubseteq y(p')$.

Once we have such an actor, we can define the number of tokens pending in the buffer of input port p at some given time t:

$$\mathsf{B}(A, x, y, p, t) = \overline{x(p)}(t) - \overline{y(p')}(t)$$

where, for an event sequence τ, $\overline{\tau}(t)$ is the total number of events occurring in τ up to, and including, time t:

$$\overline{\tau}(t) = \sup\{n + 1 \mid \tau(n) \leq t\}.$$

We call such a function an *arrival function*, because it is identical to the arrival functions in NC [8]. Then the buffer capacity required by actor A at input port p for an input trace x is:

$$\mathsf{B}(A, x, p) = \sup_{y:xAy} \sup_{t \in \mathcal{T}} \{\mathsf{B}(A, x, y, p, t)\}.$$

Using this modeling construction with output consumption ports, we can also capture *back pressure* behavior, i.e., the slowing-down of a producer A due to lack of space in the input buffer of the consumer B. To do this, the output consumption port of B can be used as an input to A, expressing the dependency on available buffer space. If desired, an extra actor of type I_k can be inserted in the feedback from B to A, to model a fixed buffer capacity. Note that this is exactly what the right-to-left edges in the models in Figure 1 represent.

When output consumption ports are present, the refinement constraint $A' \sqsubseteq A$ means that actor A' must not consume an input later than A would. Indeed, when there are dependencies by other actors on the token consumption, then these actors must not be delayed by A' more than they would be delayed by A. The upcoming proposition 11 states that our framework has this property. First we need two lemmas.

Lemma 6. *Let τ and α be a event sequence and its corresponding arrival function respectively. Then*

$$\tau(n) > t \Leftrightarrow \alpha(t) < n + 1.$$

Proof. ($\Rightarrow$) $\alpha(t) = \sup\{k + 1 \mid \tau(k) \leq t\}$. Since $\tau(n) > t$, the condition $\tau(k) \leq t$ implies by monotonicity of τ, that $k < n$ and hence that $\alpha(t) < n + 1$. ($\Leftarrow$) $\tau(n) = \inf\{u \mid n + 1 \leq \alpha(u)\}$. Since $\alpha(t) < n + 1$, $n + 1 \leq \alpha(u)$ implies, by monotonicity and left-continuity of α that $u > t + d$ for some $d > 0$ and thus $\tau(n) > t$. □

Lemma 7. *Let τ_1 and τ_2 be two event sequences. Then*

$$\tau_1 \sqsubseteq \tau_2 \Leftrightarrow \forall t \in \mathbb{R} : \overline{\tau_2}(t) \leq \overline{\tau_1}(t)$$

Proof. Lemma 6 gives us $\tau_1(n) > t \Leftrightarrow \overline{\tau_1(t)} < n + 1$ and $\tau_2(n) \leq t \Leftrightarrow \overline{\tau_2(t)} \geq n + 1$. This gives $\tau_2(n) \leq t < \tau_1(n) \Leftrightarrow \overline{\tau_2(t)} \geq n + 1 > \overline{\tau_1(t)}$ which implies $\tau_2(n) < \tau_1(n) \Leftrightarrow \overline{\tau_2(t)} > \overline{\tau_1(t)}$. Therefore, $\tau_1(n) \leq \tau_2(n) \Leftrightarrow \overline{\tau_1(t)} \geq \overline{\tau_2(t)}$. □

Proposition 11. *Let actors A and A' be such that $A' \sqsubseteq A$. Let p be an input port of these actors, with a corresponding output consumption port p'. Then*

$$\forall x \in \text{in}_A : \mathsf{B}(A', x, p) \leq \mathsf{B}(A, x, p).$$

Proof. $A' \sqsubseteq A \implies \forall x \in \text{in}_A, \forall y' : xA'y' \implies \exists y : xAy \wedge y' \sqsubseteq y$. By Lemma 7, $y'(p') \sqsubseteq y(p') \Leftrightarrow \overline{y'(p')} \geq \overline{y(p')}$. Let $\sup_{t\in\mathcal{T}} \{\overline{x(p)}(t) - \overline{y'(p')}(t)\}$ equal $\mathsf{B}(A', x, p)$, then as $\exists y : xAy \wedge \overline{y'(p')} \geq \overline{y(p')}$, $\sup_{t\in\mathcal{T}} \{\overline{x(p)}(t) - \overline{y(p')}(t)\} \geq \mathsf{B}(A', x, p)$. □

This shows that it is sufficient to do input buffer analysis for an abstract consumer and the buffer sizes are guaranteed to be sufficient also for refinements of that consumer. Note, however, that refinement of the producer to a port p may increase the required capacity for p, since the input trace x may change.

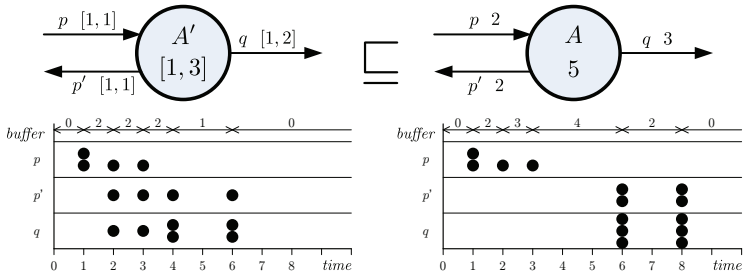


Figure 6: Actors of Figure 4 with added output ports p' modeling token consumptions at input ports p.

Example 15. *Figure 6 revisits Example 8. It shows an SDF actor A and a CSDF actor A' that refines A. Output consumption ports p' are added and example traces are shown at the bottom of the figure. The buffer sizes $\mathsf{B}(A, x, y, p, t)$ and $\mathsf{B}(A', x, y, p, t)$ are shown at the top of these traces, for different times t. Since the actors are working on the input data they are consuming, the space they are occupying is released sometime during the firing. The actual consumption of tokens and hence the release of the space is commonly modeled conservatively to occur at the end of the firing. It is easy to verify that the output event sequences of A' at ports p' and q refine the corresponding sequences of A.*

9 Finite Representations and Algorithms

So far, our treatment has been semantical, regarding actors as sets of input-output event traces. In this section, we consider syntactic, finite representations. We show that the semantics commonly associated with these representations can be embedded naturally in our theory. We also provide algorithms to check refinement and compute compositions and performance metrics on such representations. Our intention in this section is not to be complete, but rather to give examples of how our theory can be instantiated and automated.

9.1 Synchronous Data Flow

We have informally used timed SDF actors in previous examples. In this section we formally define them. Typically, in timed SDF models the time domain is the non-negative reals or integers. In the rest of this subsection, we therefore assume that $\mathcal{T} = \mathbb{R}^{\geq 0}$ or $\mathcal{T} = \mathbb{N}$.

Definition 20 (SDF actors)**.** *An actor $A = (P, Q, R_A)$ is a* homogeneous SDF actor with firing duration *$d \in \mathcal{T}$, iff*

$$R_A = \{(x, y) \mid \forall q \in Q : |y(q)| = \min_{p\in P} |x(p)| \wedge \forall n < |y(q)| : y(q)(n) = \max_{p\in P} x(p)(n) + d\}.$$

That is, the n-th firing of A starts as soon as the n-th token has arrived on every input. The firing takes d time units, after which a single output token is produced on each output. A is an SDF actor with token

transfer quanta $\boldsymbol{r} : P \cup Q \to \mathbb{N}$ and firing duration $d \in \mathcal{T}$ *iff*

$$\begin{aligned} R_A \;=\; & \{(x,y) \mid \forall q \in Q : |y(q)| = \boldsymbol{r}(q) \cdot \min_{p \in P} \big(|x(p)| \div \boldsymbol{r}(p)\big) \\ & \wedge\ \forall n < |y(q)| : y(q)(n) = d + \max_{p \in P} \max_{0 \le m < \boldsymbol{r}(p)} x(p)\big((n \div \boldsymbol{r}(q)) \cdot \boldsymbol{r}(p) + m\big)\} \end{aligned}$$

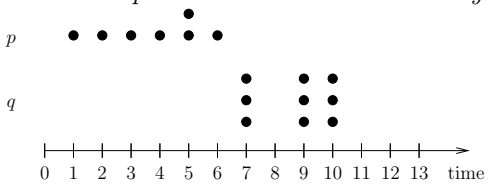
where $\div$ *denotes the quotient of the integer division. Because of monotonicity of event traces, the above is equivalent to:*

$$\begin{aligned} R_A \;=\; & \{(x,y) \mid \forall q \in Q : |y(q)| = \boldsymbol{r}(q) \cdot \min_{p \in P} \big(|x(p)| \div \boldsymbol{r}(p)\big) \\ & \wedge\ \forall n < |y(q)| : y(q)(n) = d + \max_{p \in P} x(p)\big((n \div \boldsymbol{r}(q) + 1) \cdot \boldsymbol{r}(p) - 1\big)\}. \end{aligned}$$

Homogeneous SDF actors are thus SDF actors that have all quanta equal to 1.

SDF actors are deterministic and have constant delays d. In SDF literature they are often implicitly understood to abstract behaviours with varying (non-deterministic) execution times in a conservative way.

Example 16. *Consider the SDF actor A shown in Figure 4. A has an input port p (quantum 2) and an output port q (quantum 3). Its firing duration is 5. An example input-output event trace of A is shown below. The horizontal axis represents discrete time $\mathcal{T} = \mathbb{N}$. Each bullet represents an event and multiple events happening at the same time are stacked on top of each other.The firings of A start at times 2, 4 and 5 and overlap in time. Note that the 7-th input token does not lead to any output.*



CSDF actors like the one on the left of Figure 4 can be formalized similarly, taking into account that they periodically cycle through firings with different quanta and firing durations.

An *SDF graph* represents the composition of multiple SDF actors, as in the examples shown in Figure 1. Edges in SDF graphs are often annotated with initial tokens representing the fact that the initial state of some queues is non-empty. To model this, we introduce an explicit actor:

Example 17. *The* initial token actor with $k \in \mathbb{N}$ tokens *is the actor $I_k = (\{p\}, \{q\}, R_{I_k})$ such that $(x,y) \in R_{I_k}$ iff, for $n \in \mathbb{N}$:*

$$y(q)(n) = 0 \text{ if } n < k, \text{ and } y(q)(n) = x(p)(n-k) \text{ otherwise.}$$

That is, I_k outputs k initial tokens at time 0, and then behaves as the identity function. I_k satisfies all monotonicity and continuity properties and is temporally causal (not strictly).

An SDF graph cannot always be reduced to an equivalent SDF actor. Indeed, in general, the serial or parallel composition of two SDF actors is not an SDF actor: see [41] for examples and details. In Section 9.1.3, we show how the $(\max, +)$ representation of SDF graphs can be used for composition.

We will next show how to check refinement between two SDF actors, and more generally between two SDF graphs.

9.1.1 Checking refinement on SDF actors

Let A_1 and A_2 be two SDF actors. We want to check whether $A_1 \sqsubseteq A_2$. Clearly, A_1 and A_2 must have the same sets of input and output ports, say P and Q. Suppose A_1 and A_2 have quanta functions $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$, and firing durations d_1 and d_2, respectively.

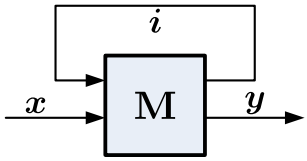


Figure 7: A $(\max, +)$ representation of a Synchronous Dataflow Graph

Proposition 12. *$A_1 \sqsubseteq A_2$ iff $d_1 \leq d_2$ and $\forall p \in P, q \in Q, n \leq \boldsymbol{r}_1(p) \cdot \boldsymbol{r}_2(p) : \boldsymbol{r}_1(q) \cdot (n \div \boldsymbol{r}_1(p)) \geq \boldsymbol{r}_2(q) \cdot (n \div \boldsymbol{r}_2(p))$.*

To show this we first prove the following lemma.

Lemma 8. *For positive integers q_1, q_2, p_1, p_2: $\forall n \in \mathbb{N} : q_1(n \div p_1) \geq q_2(n \div p_2) \implies \forall k \in \mathbb{N} : p_1(k \div q_1) + p_1 \leq p_2(k \div q_2) + p_2$*

As a corollary, we have a much simpler sufficient condition.

Corollary 1. *$A_1 \sqsubseteq A_2$ if $d_1 \leq d_2$, and $\forall p \in P : \boldsymbol{r}_1(p) \leq \boldsymbol{r}_2(p)$ and $\forall q \in Q : \boldsymbol{r}_1(q) \geq \boldsymbol{r}_2(q)$.*

9.1.2 Representing SDF graphs by $(\max, +)$ matrices

In order to check refinement it is useful to derive a representation of an SDF graph by a $(\max, +)$ recursive equation. The timing semantics of SDF is captured by a $(\max, +)$ linear equation.

For an SDF graph A with external input and output ports P and Q, $\boldsymbol{r}_A : P \cup Q \to \mathbb{N}$ denotes the *repetition vector* of the graph, which assigns to every external port the relative rates at which tokens are consumed and produced and specifies the pattern in which token dependencies are repeated [28]. For example, consider the SDF graph APP shown in Figure 1. APP has a single external port, output port p. The repetition vector assigns to p the number $\boldsymbol{r}_{\mathrm{APP}}(p) = 12 \cdot 441 = 5292$, which corresponds to the total number of tokens produced at p by 12 firings of SRC. The 12 firings of SRC, together with 5 firings of DEC, form an iteration of the SDF graph, in which the total numbers of tokens produced and consumed at every link are equal (e.g., $5 \cdot 1152 = 12 \cdot 480$ in this example).

Given an input event trace $x \in Tr(P)$ (possibly containing finite event sequences), we transform it into an infinite sequence $\boldsymbol{v}_0, \boldsymbol{v}_1, ...$, of $(\max, +)$ vectors. They are ordinary vectors, but used with $(\max, +)$ matrix-vector algebra instead of the usual linear algebra. The double indexing with port p in P and event index n with $0 \leq n < \boldsymbol{r}(p)$ within an iteration is used instead of the simple vector index to simplify the notation.

$$\boldsymbol{v}_k(p, n) = x(p)(k \cdot \boldsymbol{r}(p) + n) \text{ with } p \in P,\ k \in \mathbb{N},\ 0 \leq n < \boldsymbol{r}(p)$$

That is, $\boldsymbol{v}_k(p)$ represents the timestamps of the k-th repetition of $\boldsymbol{r}(p)$ events occurring at port p. The size of the vector $\boldsymbol{v}_k$ is equal to the sum of the entries of the input ports in the repetition vector $\boldsymbol{r}$. For convenience of dealing with finite sequences, we include ∞ into $(\max, +)$ algebra such that for any $t \in \mathbb{R}$, $t + \infty = \infty + t = \infty$. Recall that ∞ represents absence of events in our semantic framework. We also include $-\infty$ which is the absorbing element of max-plus addition: for any $t \in \mathbb{R} \cup \{\infty, -\infty\}$, $t - \infty = -\infty + t = -\infty$. We use $-\infty$ below to represent absence of a dependency in the SDF graph.

Conversely we can transform an infinite sequence $\boldsymbol{w}_k$ of output $(\max, +)$ vectors back into an output event trace:

$$y(q)(n) = \boldsymbol{w}_{n \div \boldsymbol{r}(q)}(q, n \mod \boldsymbol{r}(q))$$

where $\mod$ is the remainder of integer division.

With this vector encoding of the event traces, the input-output relation of any SDF graph can be characterized by a recursive $(\max, +)$ linear equation of the following form.

$$\begin{bmatrix} \boldsymbol{i}_{k+1} \\ \boldsymbol{w}_k \end{bmatrix} = \mathbf{M}_A \begin{bmatrix} \boldsymbol{i}_k \\ \boldsymbol{v}_k \end{bmatrix}$$

The vectors $\boldsymbol{i}_k$ capture the internal state of the SDF graph. The size of the vector $\boldsymbol{i}_k$ is equal to the number of initial tokens in the SDF graph. Typically $\boldsymbol{i}_0 = \mathbf{0}$. The vectors $\boldsymbol{v}_k$ and $\boldsymbol{w}_k$ capture the inputs and outputs, respectively.

It is convenient to split the matrix $\mathbf{M}$ into four quadrants representing the mutual dependencies between inputs, outputs and internal state.

$$\begin{bmatrix} \boldsymbol{i}_{k+1} \\ \boldsymbol{w}_k \end{bmatrix} = \begin{bmatrix} \mathbf{M}^A & \mathbf{M}^B \\ \mathbf{M}^C & \mathbf{M}^D \end{bmatrix} \begin{bmatrix} \boldsymbol{i}_k \\ \boldsymbol{v}_k \end{bmatrix} \tag{1}$$

Example 18. *The initial token actor with n tokens, I_n, is represented by a $(n+1) \times (n+1)$ matrix. It consumes and produces one token per iteration. Since the first token consumed is the $n+1$-st token to be output, I_n has a memory of n tokens, captured by a size n internal vector $\boldsymbol{i}_k$ and a size n initial vector $\boldsymbol{i}_0 = \mathbf{0}$. The vector shifts up in every iteration and the time stamp of the last token consumed is added at the bottom, i.e., the memory is formed by vector $\boldsymbol{i}_k$ of size n, for which $\boldsymbol{i}_{k+1}(m) = \boldsymbol{i}_k(m+1)$ for $0 \leq m < n-1$, and $\boldsymbol{i}_{k+1}(n) = \boldsymbol{v}_k$. The output is $\boldsymbol{w}_k = \boldsymbol{i}_k(0)$.*

$$\mathbf{I}_n = \left[\begin{array}{ccccc|c} -\infty & 0 & -\infty & -\infty & \cdots & -\infty \\ -\infty & -\infty & 0 & -\infty & \cdots & -\infty \\ -\infty & -\infty & -\infty & 0 & \cdots & -\infty \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ -\infty & -\infty & -\infty & -\infty & \cdots & 0 \\ \hline 0 & -\infty & -\infty & -\infty & \cdots & -\infty \end{array}\right]$$

For instance, for $n = 1$, we have

$$\mathbf{I}_1 = \left[\begin{array}{c|c} -\infty & 0 \\ \hline 0 & -\infty \end{array}\right]$$

which corresponds to the following system of $(\max, +)$ equations:

$$\begin{aligned} \boldsymbol{i}_{k+1} &= \boldsymbol{v}_k \\ \boldsymbol{w}_k &= \boldsymbol{i}_k = \boldsymbol{v}_{k-1} \end{aligned}$$

A homogeneous SDF actor with firing duration d is represented by a 1×1 matrix $\mathbf{A} = [d]$ and an empty initial vector, because it is memoryless and the output event is directly determined from the input event.

Example 19. *Consider the SDF graphs of Figure 9. The behavior of the top graph in Figure 9 is captured by the following $(\max, +)$ equation:*

$$\begin{bmatrix} i_{1,k+1} \\ i_{2,k+1} \\ y_{2k} \\ y_{2k+1} \end{bmatrix} = \left[\begin{array}{cc|cc} 7 & -\infty & 7 & 7 \\ 7 & -\infty & 7 & 7 \\ \hline -\infty & 3 & -\infty & -\infty \\ -\infty & 3 & -\infty & -\infty \end{array}\right] \begin{bmatrix} i_{1,k} \\ i_{2,k} \\ x_{2k} \\ x_{2k+1} \end{bmatrix}$$

corresponding to the following system of $(\max, +)$ equations:

$$\begin{aligned} i_{1,k+1} = i_{2,k+1} &= \max\{i_{1,k}, x_{2k}, x_{2k+1}\} + 7 \\ y_{2k} = y_{2k+1} &= i_{2,k} + 3 \end{aligned}$$

Hence, the matrix representation of the top graph of Figure 9 is:

$$\left[\begin{array}{cc|cc} 7 & -\infty & 7 & 7 \\ 7 & -\infty & 7 & 7 \\ \hline -\infty & 3 & -\infty & -\infty \\ -\infty & 3 & -\infty & -\infty \end{array}\right]$$

The lower SDF graph of Figure 9, which has an additional edge which may model a finite buffer capacity, is represented by the matrix:

$$\left[\begin{array}{ccc|cc} 7 & -\infty & 7 & 7 & 7 \\ 7 & -\infty & 7 & 7 & 7 \\ -\infty & 3 & -\infty & -\infty & -\infty \\ \hline -\infty & 3 & -\infty & -\infty & -\infty \\ -\infty & 3 & -\infty & -\infty & -\infty \end{array}\right]$$

9.1.3 Computing compositions of SDF graphs

We can compute the parallel composition of SDF graphs based on their matrix representation. Let A_1 and A_2 be represented by matrices

$$\mathbf{A}_1 = \begin{bmatrix} \mathbf{M}_1^A & \mathbf{M}_1^B \\ \mathbf{M}_1^C & \mathbf{M}_1^D \end{bmatrix}, \quad \mathbf{A}_2 = \begin{bmatrix} \mathbf{M}_2^A & \mathbf{M}_2^B \\ \mathbf{M}_2^C & \mathbf{M}_2^D \end{bmatrix}$$

and initial vectors $\boldsymbol{i}_1$ and $\boldsymbol{i}_2$. Then their parallel composition $A_1||A_2$ is represented by the matrix

$$\left[\begin{array}{cc|cc} \mathbf{M}_1^A & -\infty & \mathbf{M}_1^B & -\infty \\ -\infty & \mathbf{M}_2^A & -\infty & \mathbf{M}_2^B \\ \hline \mathbf{M}_1^C & -\infty & \mathbf{M}_1^D & -\infty \\ -\infty & \mathbf{M}_2^C & -\infty & \mathbf{M}_2^D \end{array}\right]$$

and has an initial vector given by the concatenation of $\boldsymbol{i}_1$ and $\boldsymbol{i}_2$.

The serial composition $A_1 \theta A_2$ is represented by the matrix

$$\left[\begin{array}{cc|c} \mathbf{M}_1^A & -\infty & \mathbf{M}_1^B \\ \mathbf{M}_2^B\mathbf{M}_1^C & \mathbf{M}_2^A & \mathbf{M}_2^B\mathbf{M}_1^D \\ \hline \mathbf{M}_2^D\mathbf{M}_1^C & \mathbf{M}_2^C & \mathbf{M}_2^D\mathbf{M}_1^D \end{array}\right]$$

Also in this case, the initial vectors are concatenated.

Note that the $(\max, +)$ representation abstracts from the port names and θ merely defines a permutation of the vector indices. For the above matrix, it is assumed that the corresponding permutation is the identity function. For other θ, the result is obtained by appropriate permutation of the rows and columns of the submatrices $\mathbf{A}_1$ and $\mathbf{A}_2$ respectively.

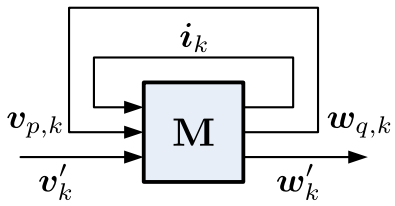


Figure 8: Feedback on $(\max, +)$ representations of SDF graphs.

Feedback in SDF graphs is achieved by connecting outputs to inputs with identical quanta in one graph iteration and without direct dependencies. We illustrate the feedback construction in Figure 8. To create feedback from input port p to output port q, we split the inputs $\boldsymbol{v}$ and outputs $\boldsymbol{w}$ up in two parts, $\boldsymbol{v}_p$ and $\boldsymbol{v}'$ and $\boldsymbol{w}_q$ and $\boldsymbol{w}'$ respectively where $\boldsymbol{v}_p$ contains the elements of vector $\boldsymbol{v}$ with events of port p and $\boldsymbol{w}_q$ contains the elements of vector $\boldsymbol{w}$ with events of port q. We feed $\boldsymbol{w}_q$ back to $\boldsymbol{v}_p$. We assume that those inputs and outputs that are used for the feedback are aligned to the same rows in the matrix. We can always align them

by appropriate permutation of the rows and columns of the matrix. This gives us a matrix equation of the following form:

$$\left[\begin{array}{c} \boldsymbol{i}_{k+1} \\ \boldsymbol{w}_{q,k} \\ \boldsymbol{w}'_K \end{array}\right] = \left[\begin{array}{c|cc} \mathbf{M}^A & \mathbf{M}^B_1 & \mathbf{M}^B_2 \\ \hline \mathbf{M}^C_1 & \mathbf{M}^D_{1,1} & \mathbf{M}^D_{1,2} \\ \mathbf{M}^C_2 & \mathbf{M}^D_{2,1} & \mathbf{M}^D_{2,2} \end{array}\right] \left[\begin{array}{c} \boldsymbol{i}_k \\ \boldsymbol{v}_{p,k} \\ \boldsymbol{v}'_k \end{array}\right]$$

The center block, $\mathbf{M}^D_{1,1}$, in the matrix must contain only $-\infty$ to ensure there are no direct dependencies from p to q. Now we can solve the equation and eliminate $\boldsymbol{v}_p$ from the input vector using the constraint $\boldsymbol{v}_{p,k} = \boldsymbol{w}_{q,k}$ (for brevity and clarity we use the $(\max, +)$ notation $\oplus$ of the maximum operator):

$$\begin{aligned}
\left[\begin{array}{c} \boldsymbol{i}_{k+1} \\ \boldsymbol{w}_{q,k} \\ \boldsymbol{w}'_k \end{array}\right] &= \left[\begin{array}{c|cc} \mathbf{M}^A & \mathbf{M}^B_1 & \mathbf{M}^B_2 \\ \hline \mathbf{M}^C_1 & -\infty & \mathbf{M}^D_{1,2} \\ \mathbf{M}^C_2 & \mathbf{M}^D_{2,1} & \mathbf{M}^D_{2,2} \end{array}\right] \left[\begin{array}{c} \boldsymbol{i}_k \\ \boldsymbol{v}_{p,k} \\ \boldsymbol{v}'_k \end{array}\right] \\
&= \left[\begin{array}{c|cc} \mathbf{M}^A & \mathbf{M}^B_1 & \mathbf{M}^B_2 \\ \hline \mathbf{M}^C_1 & -\infty & \mathbf{M}^D_{1,2} \\ \mathbf{M}^C_2 & \mathbf{M}^D_{2,1} & \mathbf{M}^D_{2,2} \end{array}\right] \left[\begin{array}{c} \boldsymbol{i}_k \\ \mathbf{M}^C_1 \boldsymbol{i}_k \oplus \mathbf{M}^D_{1,2} \boldsymbol{v}'_k \\ \boldsymbol{v}'_k \end{array}\right] \\
&= \left[\begin{array}{c|c} \mathbf{M}^A \oplus \mathbf{M}^B_1 \mathbf{M}^C_1 & \mathbf{M}^B_2 \oplus \mathbf{M}^B_1 \mathbf{M}^D_{1,2} \\ \hline \mathbf{M}^C_1 & \mathbf{M}^D_{1,2} \\ \mathbf{M}^C_2 \oplus \mathbf{M}^D_{2,1} \mathbf{M}^C_1 & \mathbf{M}^D_{2,2} \oplus \mathbf{M}^D_{2,1} \mathbf{M}^D_{1,2} \end{array}\right] \left[\begin{array}{c} \boldsymbol{i}_k \\ \boldsymbol{v}'_k \end{array}\right]
\end{aligned}$$

9.1.4 Checking refinement on SDF graphs

From the recursive Equation (1), we can derive an explicit function from the input vectors to the output vectors:

$$\begin{aligned}
\boldsymbol{w}_k &= \mathbf{M}^C \boldsymbol{i}_k \oplus \mathbf{M}^D \boldsymbol{v}_k \\
&= \mathbf{M}^C (\mathbf{M}^A \boldsymbol{i}_{k-1} \oplus \mathbf{M}^B \boldsymbol{v}_{k-1}) \oplus \mathbf{M}^D \boldsymbol{v}_k \\
&= \mathbf{M}^C \mathbf{M}^A \boldsymbol{i}_{k-1} \oplus \mathbf{M}^C \mathbf{M}^B \boldsymbol{v}_{k-1} \oplus \mathbf{M}^D \boldsymbol{v}_k \\
&= \mathbf{M}^C \mathbf{M}^A (\mathbf{M}^A \boldsymbol{i}_{k-2} \oplus \mathbf{M}^B \boldsymbol{v}_{k-2}) \oplus \mathbf{M}^C \mathbf{M}_B \boldsymbol{v}_{k-1} \oplus \mathbf{M}^D \boldsymbol{v}_k \\
&= \mathbf{M}^C \mathbf{M}^{A^2} \boldsymbol{i}_{k-2} \oplus \mathbf{M}^C \mathbf{M}^A \mathbf{M}^B \boldsymbol{v}_{k-2} \oplus \mathbf{M}^C \mathbf{M}^B \boldsymbol{v}_{k-1} \oplus \mathbf{M}^D \boldsymbol{v}_k \\
&= \dots \\
&= \mathbf{M}^C \mathbf{M}^{A^k} \boldsymbol{i}_0 \oplus \mathbf{M}^D \boldsymbol{v}_k \oplus \mathbf{M}^C \bigoplus_{n=0}^{k-1} \mathbf{M}^{A^n} \mathbf{M}^B \boldsymbol{v}_{k-1-n}
\end{aligned} \tag{2}$$

Note that for an SDF graph without open inputs, this reduces to:

$$\boldsymbol{w}_k = \mathbf{M}^C \mathbf{M}^{A^k} \boldsymbol{i}_0 \tag{3}$$

For equal sized matrixes $\mathbf{M}_1$ and $\mathbf{M}_2$ with K rows and L columns, we write $\mathbf{M}_1 \leq \mathbf{M}_2$ iff for all $0 \leq k < K$ and $0 \leq l < L$, $\mathbf{M}_1(k,l) \leq \mathbf{M}_2(k,l)$.

We next show how to check refinement between two SDF graphs. We assume that the two graphs have consistent repetition vectors, i.e., vectors that have a common multiple and the same ratios between all input and output port pairs.[1] Two such graphs generally have submatrices $\mathbf{M}^D$ of different size. In order to compare the graphs we need matrices for which $\mathbf{M}^D$ is of equal size. To achieve this, we can aggregate the appropriate number of iterations of the graph into a single vector, according to the common hyper period of the graphs. As an example, we show how to combine two iterations into one: combining

$$\left[\begin{array}{c} \boldsymbol{i}_{2k+1} \\ \boldsymbol{w}_{2k} \end{array}\right] = \left[\begin{array}{c|c} \mathbf{M}^A & \mathbf{M}^B \\ \hline \mathbf{M}^C & \mathbf{M}^D \end{array}\right] \left[\begin{array}{c} \boldsymbol{i}_{2k} \\ \boldsymbol{v}_{2k} \end{array}\right] \quad \text{and} \quad \left[\begin{array}{c} \boldsymbol{i}_{2k+2} \\ \boldsymbol{w}_{2k+1} \end{array}\right] = \left[\begin{array}{c|c} \mathbf{M}^A & \mathbf{M}^B \\ \hline \mathbf{M}^C & \mathbf{M}^D \end{array}\right] \left[\begin{array}{c} \boldsymbol{i}_{2k+1} \\ \boldsymbol{v}_{2k+1} \end{array}\right]$$

[1] Notice that for atomic SDF actors we can check refinement also in the case where the repetition vectors (in this case, quanta functions) are not consistent, as stated in Proposition 12.

gives:

$$\begin{bmatrix} \boldsymbol{i}_{2k+2} \\ \boldsymbol{w}_{2k} \\ \boldsymbol{w}_{2k+1} \end{bmatrix} = \left[\begin{array}{c|cc} \mathbf{M}^{A^2} & \mathbf{M}^A\mathbf{M}^B & \mathbf{M}^B \\ \hline \mathbf{M}^C & \mathbf{M}^D & -\infty \\ \mathbf{M}^C\mathbf{M}^A & \mathbf{M}^C\mathbf{M}^B & \mathbf{M}^D \end{array}\right] \begin{bmatrix} \boldsymbol{i}_{2k} \\ \boldsymbol{v}_{2k} \\ \boldsymbol{v}_{2k+1} \end{bmatrix}$$

Note that the size of the internal state vector remains the same.

Lemma 9. *Let* $\mathbf{M}$ *and* $\mathbf{N}$ *be two matrix representations of SDF graphs* A *and* B *with the same input and output ports respectively and with the same port iteration quanta.* $B \sqsubseteq A$ *if and only if (i)* $\mathbf{N}^D \leq \mathbf{M}^D$*, (ii)* $\mathbf{N}^C{\mathbf{N}^A}^k\mathbf{N}^B \leq \mathbf{M}^C{\mathbf{M}^A}^k\mathbf{M}^B$ *for all* $k \geq 0$ *and (iii)* $\mathbf{N}^C{\mathbf{N}^A}^k\boldsymbol{i}_{\mathbf{N},0} \leq \mathbf{M}^C{\mathbf{M}^A}^n\boldsymbol{i}_{\mathbf{M},0}$ *for all* n*.*

Proof. Note that even if $\mathbf{M}$ and $\mathbf{N}$ have differently sized internal states, the sizes of the matrices we are comparing are identical. The 'if' direction is trivial. For the 'only if' direction, consider the following. If $\mathbf{N}^D \not\leq \mathbf{M}^D$, let $\mathbf{N}^D(k,l) > \mathbf{M}^D(k,l)$. Let k correspond to event number m on output port q and l correspond to event number n on input port p. If we consider input event trace x where $x(r)(i) = 0$ for all $r \neq p$ and all i and $x(p)(i) = 0$ for all $i < n$ and $x(p)(i) = t$ for all $i \geq n$ where t is a constant. $y_A(q)(m) = (\mathbf{M}^C\boldsymbol{i}_{\mathbf{M},0})(k) \oplus \max_{0\leq i<\boldsymbol{r}(p)} \mathbf{M}^D(k, s+i) + x(p)(i)$, where s is the starting index of the tokens of port p in the vector, so that $s + n = l$. $y_B(q)(m) = (\mathbf{N}^C\boldsymbol{i}_{\mathbf{N},0})(k) \oplus \max_{0\leq i<\boldsymbol{r}(p)} \mathbf{N}^D(k, s+i) + x(p)(i)$. For sufficiently large t, $y_A(q)(m) = \max_{0\leq i<\boldsymbol{r}(p)} \mathbf{M}^D(k, s+i) + x(p)(i) = \max_{n\leq i<\boldsymbol{r}(p)} \mathbf{M}^D(k, s+i) + t$ and $y_B(q)(m) = \max_{0\leq i<\boldsymbol{r}(p)} \mathbf{N}^D(k, s+i) + x(p)(i) = \max_{n\leq i<\boldsymbol{r}(p)} \mathbf{N}^D(k, s+i) + t$. For SDF graphs, $y_A(q)(m) = \mathbf{M}^D(k, s+n) + t$ and $y_B(q)(m) = \mathbf{N}^D(k, s+n) + t$. Hence, $y_A(q)(m) < y_B(q)(m)$ and because SDF actors are deterministic, B does not refine A. The proof for the other terms is similar, but even more tedious. □

The conditions of Lemma 9 can be checked based on the fact that sequences of the form $\mathbf{M}^k$ in $(\max, +)$ algebra are known to become ultimately periodic, i.e., there exist some K and N such that for all $k \geq K$, $\mathbf{M}^k = \mathbf{M}^{k-N} + N\lambda_{\mathbf{M}}$ where $\lambda_{\mathbf{M}}$ is the eigenvalue of the matrix $\mathbf{M}$.

Proposition 13. *The refinement relation between SDF graphs with consistent quanta is decidable.*

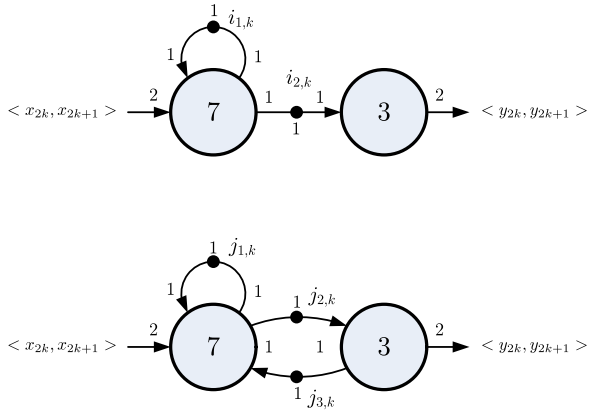


Figure 9: Refinement of an SDF graph to bound the size of a buffer.

Example 20. *Figure 9 shows two SDF graphs. Both graphs have one external input port and one external output port. Notation* $< x_{2k}, x_{2k+1} >$ *is used for the subsequence of the time stamps of two successive tokens consumed at the external input of the graph, and similarly for* $< y_{2k}, y_{2k+1} >$ *at the output. Notations '7' and '3' denote the firing durations of the corresponding actors. The bottom graph is identical to the top graph except that it has an additional, back-pressure edge.*

Recall from Example 19 that the top graph can be represented by the following $(\max, +)$ *equation:*

$$\begin{bmatrix} i_{1,k+1} \\ i_{2,k+1} \\ y_{2k} \\ y_{2k+1} \end{bmatrix} = \left[\begin{array}{cc|cc} 7 & -\infty & 7 & 7 \\ 7 & -\infty & 7 & 7 \\ \hline -\infty & 3 & -\infty & -\infty \\ -\infty & 3 & -\infty & -\infty \end{array}\right] \begin{bmatrix} i_{1,k} \\ i_{2,k} \\ x_{2k} \\ x_{2k+1} \end{bmatrix}$$

It follows from the above that for all k, $i_{1,k} = i_{2,k}$. *Therefore, the equation can be simplified to:*

$$\begin{bmatrix} i_{12,k+1} \\ y_{2k} \\ y_{2k+1} \end{bmatrix} = \left[\begin{array}{c|cc} 7 & 7 & 7 \\ \hline 3 & -\infty & -\infty \\ 3 & -\infty & -\infty \end{array}\right] \begin{bmatrix} i_{12,k} \\ x_{2k} \\ x_{2k+1} \end{bmatrix}$$

The lower SDF graph of Figure 9 can be represented by the following equation:

$$\begin{bmatrix} j_{12,k+1} \\ j_{3,k+1} \\ y_{2k} \\ y_{2k+1} \end{bmatrix} = \left[\begin{array}{cc|cc} 7 & 7 & 7 & 7 \\ 3 & -\infty & -\infty & -\infty \\ \hline 3 & -\infty & -\infty & -\infty \\ 3 & -\infty & -\infty & -\infty \end{array}\right] \begin{bmatrix} j_{12,k} \\ j_{3,k} \\ x_{2k} \\ x_{2k+1} \end{bmatrix}$$

We have

$$\mathbf{M} = \left[\begin{array}{c|cc} 7 & 7 & 7 \\ \hline 3 & -\infty & -\infty \\ 3 & -\infty & -\infty \end{array}\right] \text{ and } \mathbf{N} = \left[\begin{array}{cc|cc} 7 & 7 & 7 & 7 \\ 3 & -\infty & -\infty & -\infty \\ \hline 3 & -\infty & -\infty & -\infty \\ 3 & -\infty & -\infty & -\infty \end{array}\right]$$

We wish to apply Lemma 9. We can immediately see that $\mathbf{M}^D = \mathbf{N}^D$. *We next compute:*

$$\begin{aligned} \mathbf{M}^C \mathbf{M}^{A^k} \boldsymbol{i}_0 &= \mathbf{M}^C \begin{bmatrix} 7 \end{bmatrix}^k \boldsymbol{i}_0 \\ &= \mathbf{M}^C \begin{bmatrix} 7k \end{bmatrix} \boldsymbol{i}_0 \\ &= \begin{bmatrix} 3 \\ 3 \end{bmatrix} \begin{bmatrix} 7k \end{bmatrix} \boldsymbol{i}_0 \\ &= \begin{bmatrix} 7k+3 \\ 7k+3 \end{bmatrix} \begin{bmatrix} 0 \end{bmatrix} \\ &= \begin{bmatrix} 7k+3 \\ 7k+3 \end{bmatrix} \end{aligned}$$

and similarly for $\mathbf{N}$*:*

$$\begin{aligned} \mathbf{N}^C \mathbf{N}^{A^k} \boldsymbol{j}_0 &= \mathbf{N}^C \begin{bmatrix} 7 & 7 \\ 3 & -\infty \end{bmatrix}^k \boldsymbol{j}_0 \\ &= \mathbf{N}^C \begin{bmatrix} 7k & 7k \\ 7k-4 & 7k-4 \end{bmatrix} \boldsymbol{j}_0 \text{ (assuming for the moment } k \geq 2) \\ &= \begin{bmatrix} 3 & -\infty \\ 3 & -\infty \end{bmatrix} \begin{bmatrix} 7k & 7k \\ 7k-4 & 7k-4 \end{bmatrix} \boldsymbol{j}_0 \\ &= \begin{bmatrix} 7k+3 & 7k+3 \\ 7k+3 & 7k+3 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 7k+3 \\ 7k+3 \end{bmatrix} \end{aligned}$$

It is easy to check separately that the final result also holds for $k = 0, 1$, *thus it holds for all* $k \in \mathbb{N}$, *which shows* $\mathbf{M}^C \mathbf{M}^{A^k} \boldsymbol{i}_0 = \mathbf{N}^C \mathbf{N}^{A^k} \boldsymbol{j}_0$ *for all* $k \in \mathbb{N}$.

Finally, we compute:

$$\begin{aligned}\mathbf{M}^C\mathbf{M}^{A^k}\mathbf{M}^B &= \begin{bmatrix} 7k+3 \\ 7k+3 \end{bmatrix} \mathbf{M}^B \\ &= \begin{bmatrix} 7k+3 \\ 7k+3 \end{bmatrix} \begin{bmatrix} 7 & 7 \end{bmatrix} \\ &= \begin{bmatrix} 7k+10 & 7k+10 \\ 7k+10 & 7k+10 \end{bmatrix}\end{aligned}$$

and similarly for $\mathbf{N}$*:*

$$\begin{aligned}\mathbf{N}^C\mathbf{N}^{A^k}\mathbf{N}^B &= \begin{bmatrix} 7k+3 & 7k+3 \\ 7k+3 & 7k+3 \end{bmatrix} \mathbf{N}^B \\ &= \begin{bmatrix} 7k+3 & 7k+3 \\ 7k+3 & 7k+3 \end{bmatrix} \begin{bmatrix} 7 & 7 \\ -\infty & -\infty \end{bmatrix} \\ &= \begin{bmatrix} 7k+10 & 7k+10 \\ 7k+10 & 7k+10 \end{bmatrix}\end{aligned}$$

therefore $\mathbf{M}^C\mathbf{M}^{A^k}\mathbf{M}^B = \mathbf{N}^C\mathbf{N}^{A^k}\mathbf{N}^B$ *for all* $k \in \mathbb{N}$*. Thus, for the two SDF graphs the corresponding terms used in the conditions of Lemma 9 are equal, and hence the two SDF graphs refine each other and are in fact equivalent. Note that in this example the behavior of the terms becomes periodic with* k *right from the start.*

9.1.5 Computing throughput bounds on SDF graphs

There are well known methods for computing throughput of an SDF graph. We will show here that these methods can be also used to compute our notions of throughput bounds. SDF throughput is typically computed for closed graphs without any inputs, as sources are usually modeled as part of the graph. Common (equivalent) techniques for calculating throughput are based on Maximum Cycle Ratio analysis (in timed synchronous dataflow literature sometimes incorrectly called Maximum Cycle Mean analysis) of an equivalent homogeneous SDF graph [36] on an explicit state-space exploration of the operational semantics or on a $(\max, +)$ formulation of the semantics.

We have seen in Equation 3 that for an SDF graph without open inputs, the following equation calculates the output trace.

$$\boldsymbol{w}_k = \mathbf{M}^C\mathbf{M}^{A^k}\boldsymbol{i}_0$$

It is known [4] that the sequence of vectors $\boldsymbol{w}_k$ becomes eventually periodic and that the average rate at which all the entries in the vector grow, are determined by the eigenvector of the matrix $\mathbf{M}^A$. If the events on output port q are governed by eigenvalue λ, then the throughput on that port is equal to $\boldsymbol{r}(q)/\lambda$.

We next examine SDF graphs which do have open inputs. Clearly, now the throughput on output sequences depends on the throughput with which inputs are offered on the input ports. However, internal dependencies within the SDF graph may result in the output throughput not being proportional to the throughput at the inputs, but being constrained by the internal throughput limitation of the SDF graph.

Recall from Equation 2, that the general formula for the output sequence of vectors is as follows.

$$\boldsymbol{w}_k = \mathbf{M}^C\mathbf{M}^{A^k}\boldsymbol{i}_0 \oplus \mathbf{M}^D\boldsymbol{v}_k \oplus \mathbf{M}^C \bigoplus_{n=0}^{k-1} \mathbf{M}^{A^n}\mathbf{M}^B\boldsymbol{v}_{k-1-n}$$

Splitting the equation in terms of vectors into separate output sequences reveals that the outputs are defined as the maximum of a set of event sequences. Let N be the size of the vectors $\boldsymbol{v}_k$ and M the size of the vectors $\boldsymbol{w}_k$.

Introduce the infinite sequences $w_m(k)$, $v_n(k)$, $i_m(k)$ and $h_{m,n}(k)$ for $0 \leq m < M$ and $0 \leq n < N$, defined as follows.

$$\begin{aligned} w_m(k) &:= \boldsymbol{w}_k(m) \\ v_n(k) &:= \boldsymbol{v}_k(n) \\ i_m(k) &:= \boldsymbol{i}_k(m) = \left(\mathbf{M}^C \mathbf{M}^{A^k} \boldsymbol{i}_0\right)(m) \\ h_{m,n}(k) &:= h_{m,n}(k) = \begin{cases} \mathbf{M}^D(m,n) & \text{if} \quad k = 0 \\ (\mathbf{M}^C \mathbf{M}^{A^{k-1}} \mathbf{M}^B)(m,n) & \text{if} \quad k > 0 \end{cases} \end{aligned}$$

Then we can rewrite Equation 2 as follows. (Using $\mathbf{M}(m,*)$ to denote the m-th row of matrix $\mathbf{M}$.)

$$\begin{aligned} w_m(k) &= \left(\mathbf{M}^C \mathbf{M}^{A^k} \boldsymbol{i}_0\right)(m) \oplus \left(\mathbf{M}^D\right)(m,*)\boldsymbol{v}_k \oplus \left(\mathbf{M}^C \bigoplus_{l=0}^{k-1} \mathbf{M}^{A^l} \mathbf{M}^B \boldsymbol{v}_{k-1-l}\right)(m) \\ &= i_m(k) \oplus \bigoplus_{n=0}^{N-1} \mathbf{M}^D(m,n)\boldsymbol{v}_k(n) \oplus \left(\mathbf{M}^C \bigoplus_{l=0}^{k-1} \bigoplus_{n=0}^{N-1} \left(\mathbf{M}^{A^l} \mathbf{M}^B\right)(m,n)\boldsymbol{v}_{k-1-l}(n)\right) \\ &= i_m(k) \oplus \bigoplus_{n=0}^{N-1} \left(\mathbf{M}^D(m,n)v_n(k) \oplus \left(\bigoplus_{l=0}^{k-1} \mathbf{M}^C \mathbf{M}^{A^l} \mathbf{M}^B(m,n)v_n(k-1-l)\right)\right) \\ &= i_m(k) \oplus \bigoplus_{n=0}^{N-1} \left(h_{m,n}(0)v_n(k) \oplus \left(\bigoplus_{l=1}^{k} h_{m,n}(l)v_n(k-l)\right)\right) \end{aligned}$$

In the last line, we recognize $\oplus$ (max) and the $(\max, +)$ convolution operators on event sequences, defined as follows.

Definition 21. *Let τ_1 and τ_2 be two event sequences. The sequence $\max(\tau_1, \tau_2)$ (also written $\tau_1 \oplus \tau_2$) is the event sequence such that $\max(\tau_1, \tau_2)(n) = \max(\tau_1(n), \tau_2(n))$. The $(\max, +)$-convolution $\tau_1 \otimes \tau_2$ is the event sequence such that $\tau_1 \otimes \tau_2(n) = \max_{0 \leq k \leq n} \tau_1(k) + \tau_2(n-k)$.*

Using these operator, the equation becomes:

$$w_m = i_m \oplus \bigoplus_{n=0}^{N-1} (h_{m,n} \otimes v_n) \tag{4}$$

The throughput of the output sequences w_m is determined by a combination of max operations and $(\max, +)$ convolutions of the internal sequences i_m, the *impulse response* sequences $h_{m,n}$ and the input sequences v_n. Let us investigate how our throughput definition behaves for these operations.

Lemma 10. $T^{lb}(\max(\tau_1, \tau_2)) = \min(T^{lb}(\tau_1), T^{lb}(\tau_2))$.

Proof. (i) Let $T < T^{lb}(\tau_1)$ and $T < T^{lb}(\tau_2)$. There exist K_1 and K_2 such that for all $n > \max(K_1, K_2)$, $n > \tau_1(n) \cdot T$ and $n > \tau_2(n) \cdot T$ and thus $n > \max(\tau_1, \tau_2)(n) \cdot T$. And thus we may conclude that $T < T^{lb}(\max(\tau_1, \tau_2))$. (ii) If $T > T^{lb}(\tau_1)$, then $T > T^{lb}(\max(\tau_1, \tau_2))$, because $\max(\tau_1, \tau_2)(n) \geq \tau_1(n)$ for all n. Similarly for $T > T^{lb}(\tau_2)$. From (i) and (ii) together it follows that $T^{lb}(\max(\tau_1, \tau_2)) = \min(T^{lb}(\tau_1), T^{lb}(\tau_2))$. □

Lemma 11. $T^{lb}(\tau_1 \otimes \tau_2) = \min(T^{lb}(\tau_1), T^{lb}(\tau_2))$.

Proof. (i) Let $T < T^{lb}(\tau_1)$ and $T < T^{lb}(\tau_2)$ and let T' be such that $T < T' < T^{lb}(\tau_1)$ and $T < T' < T^{lb}(\tau_2)$. There exist K_1 such that for all $n > K_1$, $n > \tau_1(n)T'$. Similarly, there exists K_2 such that for all $n > K_2$, $n > \tau_2(n)T'$. This means that there exist Δ_1 and Δ_2 such that $n + \Delta_1 > \tau_1(n)T'$ and $n + \Delta_2 > \tau_2(n)T'$ for

all $n \geq 0$. Then for any n and $k \leq n$, $n + \Delta_1 + \Delta_2 = k + \Delta_1 + (n-k) + \Delta_2 > \tau_1(k)T' + \tau_2(n-k)T'$ and if we let $\Delta = \Delta_1 + \Delta_2$, $n + \Delta > (\tau_1 \otimes \tau_2)(n)T'$. Now,

$$\begin{aligned} n &> (\tau_1 \otimes \tau_2)(n)T' - \Delta \\ nT &> (\tau_1 \otimes \tau_2)(n)T'T - \Delta T \\ nT' &> (\tau_1 \otimes \tau_2)(n)T'T - \Delta T + n(T' - T) \\ n &> (\tau_1 \otimes \tau_2)(n)T + \frac{n(T' - T) - \Delta T}{T'} \\ n &> (\tau_1 \otimes \tau_2)(n)T \quad \text{if} \quad n > \frac{\Delta T}{T' - T} \end{aligned}$$

From this result we may conclude that $T < T^{lb}(\tau_1 \otimes \tau_2)$.

(ii) Let w.l.o.g. $T > T^{lb}(\tau_1)$. Then $T > T^{lb}(\tau_1 \otimes \tau_2)$, because $\tau_1 \otimes \tau_2(n) \geq \tau_1(n)$ for all n.

From (i) and (ii) together it follows that $T^{lb}(\tau_1 \otimes \tau_2) = \min(T^{lb}(\tau_1), T^{lb}(\tau_2))$. □

We observe that the throughput of the sequences i_n and $h_{m,n}$ are determined by the eigenvalues of the matrix $\mathbf{M}^A$ and are equal to $1/\lambda$ if λ is the unique eigenvalue of $\mathbf{M}^A$ [4]. On a strongly connected SDF graph, $\mathbf{M}^A$ has a single eigenvalue and the throughput of each of these signals is the same.

Proposition 14. *Let A be a strongly connected SDF graph with input ports P and output ports Q represented by the quadrant $(\max, +)$ matrices $\mathbf{M}^A$, $\mathbf{M}^B$, $\mathbf{M}^C$ and $\mathbf{M}^D$. Let $\boldsymbol{r} : P \cup Q \to \mathbb{N}$ be the port iteration quanta of A. Let x be an input trace of A. Then $\mathbf{M}^A$ has a unique eigenvalue, λ, and*

$$T^{lb}(A, x, q) = \boldsymbol{r}(q) \cdot \min(\frac{1}{\lambda}, \min_{p \in P} \frac{T^{lb}(x(p))}{\boldsymbol{r}(p)}).$$

Proof. Follows directly from the characterization by Equation 4 and Lemmas 10 and 11. □

Example 21. *Recall that the matrix representation of the bottom graph of Figure 9 is*

$$\left[\begin{array}{ccc|cc} 7 & -\infty & 7 & 7 & 7 \\ 7 & -\infty & 7 & 7 & 7 \\ -\infty & 3 & -\infty & -\infty & -\infty \\ \hline -\infty & 3 & -\infty & -\infty & -\infty \\ -\infty & 3 & -\infty & -\infty & -\infty \end{array}\right]$$

It follows from the solution to the eigenvalue equation for the top-left block matrix that $\lambda = 7$:

$$\begin{bmatrix} 7 & -\infty & 7 \\ 7 & -\infty & 7 \\ -\infty & 3 & -\infty \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -4 \end{bmatrix} = 7 + \begin{bmatrix} 0 \\ 0 \\ -4 \end{bmatrix}$$

With $\boldsymbol{r}(x) = 2$ and $\boldsymbol{r}(y) = 2$, the throughput lower bound on y becomes according to Proposition 14:

$$T^{lb}(y) = 2 \cdot \min\left(\frac{1}{7}, \frac{T^{lb}(x)}{2}\right) = \min\left(\frac{2}{7}, T^{lb}(x)\right).$$

9.1.6 Computing latency on SDF Graphs

Various methods for computing latency exist in the SDF literature [19, 31], differing in particular in how the related input-output events are specified. Sometimes they are left implicit and derived from the data dependencies defined by the SDF graph. Sometimes they are specified explicitly. It is natural to specify the related input-output events in patterns which repeat with the periodic behavior of SDF iterations. In

particular, an IOES can be captured by a set $\hat{\mathcal{E}} \in 2^{P\times\mathbb{N}} \times 2^{Q\times\mathbb{N}}$. Then, for an SDF graph with repetition vector $\boldsymbol{r}$, the actual IOES associated with $\hat{\mathcal{E}} = (E_P, E_Q)$ is

$$\begin{aligned} \mathcal{E} &= \{(E_P^k, E_Q^k) \mid k \geq 0\}, \qquad \text{where} \\ E_P^k &= \{(p, n + k\boldsymbol{r}(p)) \mid (p,n) \in E_P\} \\ E_Q^k &= \{(q, n + k\boldsymbol{r}(q)) \mid (q,n) \in E_Q\} \end{aligned}$$

Latency can be computed by an exploration of the operational behavior of the SDF graph. This behavior is deterministic and ultimately periodic, where a period spans one or more iterations of the graph. The latency can be determined by exploring the behavior until it becomes periodic and observing the latency of the events. The latency values after that are identical because of the periodic behavior.

9.1.7 Computing buffer capacities on SDF graphs

We have seen in Proposition 14 that outputs of an SDF graph may have a limited throughput, independent of the throughput on its input event sequences. This also applies to the outputs modeling the consumption of data from its inputs; it can process its input data with a limited speed. As a consequence, if the input is offered with a higher throughput, then the required capacity of the input buffer is unbounded, i.e., ∞.

As a first check, therefore, it is necessary to determine the throughput of the input event sequence(s) and the internal throughput of the consuming SDF graph. Focusing on a particular input port p, we can determine the throughput T^p of the event sequence produced on p versus the throughput $T^{p'}$ on the consumption port p'. If $T^p > T^{p'}$ then the required capacity is unbounded, otherwise they must be equal, because it is not possible to consume more than is produced. In case the throughput is equal, the maximal buffer occupation can be determined from an exploration of the state-space of the operational semantics of SDF, as is the underlying method for the exact SDF buffer size analysis of [37].

9.1.8 Time non-deterministic SDF

The formal semantics of SDF typically (as well as in our case) assumes constant firing durations. Often these durations are (implicitly) understood to be worst-case firing durations of time non-deterministic actors in the final implementation. Existing works, e.g., [46, 37], claim that event sequences derived with constant firing durations are conservative (pessimistic) estimates of the final implementation with varying firing durations, and similarly that performance metrics of the model with constant firing durations are conservative estimates of these metrics in the final implementation. We address this relation formally within our framework, and confirm this claim. Given the same port quanta, actors with a constant firing duration refine and are refined by actors with a variable firing duration, as they have the same upper bound on their firing duration. This implies that for instance max-plus algebra can be used to derive conservative event traces, throughput, latency, and buffer requirements of actors in the final implementation.

9.2 Service Curves

In this section we study actors on $\mathcal{T} = \mathbb{R}^{\geq 0}$ defined by *service curves* and convolutions. Our study is motivated by NC and RTC, however, we adopt a simpler setting (omitting resource modeling), merely aiming at illustrating the main concepts and at drawing links. Moreover, while NC typically presents its results for so-called fluid models (continuous data over continuous time) we focus on discrete behaviors over continuous time.

In NC and RTC, event sequences are represented as *arrival functions*, i.e., left-continuous functions $\alpha : \mathbb{R}^{\geq 0} \to \mathbb{N}$, where $\alpha(t)$ represents the cumulative total of events that have occurred up to (and including) time t. We can go from a (non-zeno) event sequence τ to its arrival function $\overline{\tau}$ defined by $\overline{\tau}(t) = \sup\{n+1 \mid \tau(n) \leq t\}$ (as introduced in Section 8). By definition, $\sup \emptyset = 0$, so that $\overline{\tau}(t) = 0$ for $t < \tau(0)$. Note that $\overline{\tau}$ is indeed left-continuous. Conversely, an arrival function α defines an event sequence $\overline{\alpha}(n) = \inf\{t \mid n+1 \leq \alpha(t)\}$. By definition, $\inf \emptyset = \infty$, so that if $n + 1 > \alpha(t)$ for all t, then $\tau(n) = \infty$ and the n-th event never occurs.

Lemma 12. *Let τ be a non-zeno event sequence and let α be an arrival function. Then $\overline{\overline{\tau}} = \tau$ and $\overline{\overline{\alpha}} = \alpha$.*

Proof.

$$\begin{aligned}
\overline{\overline{\tau}}(n) &= \inf\{t \mid n+1 \leq \overline{\tau}(t)\} \\
&= \inf\{t \mid n+1 \leq \sup\{m+1 \mid \tau(m) \leq t\}\} \\
&= \inf\{t \mid n \leq \sup\{m \mid \tau(m) \leq t\}\} \\
&= \inf\{t \mid \exists m : n \leq m : \tau(m) \leq t\} \\
&= \inf\{t \mid \tau(n) \leq t\} \\
&= \tau(n)
\end{aligned}$$

$$\begin{aligned}
\overline{\overline{\alpha}}(t) &= \sup\{n+1 \mid \overline{\alpha}(n) \leq t\} \\
&= \sup\{n+1 \mid \inf\{u \mid n+1 \leq \alpha(u)\} \leq t\} \\
&= \sup\{n \mid \inf\{u \mid n \leq \alpha(u)\} \leq t\} \\
&= \sup\{n \mid \exists u : n \leq \alpha(u) : u \leq t\} \\
&= \sup\{n \mid n \leq \alpha(t)\} \\
&= \alpha(t)
\end{aligned}$$

□

Note that event sequences bear strong similarities to the *pseudo-inverse functions* of NC [8]. The duality between the two descriptions is also known in the $(\max, +)$ work, for instance between *dater* and *counter* descriptions [4].

Classes of arrival functions are usually characterized by means of arrival *curves*, i.e., functions $F : \mathbb{R}^{\geq 0} \to \mathbb{N}$ specifying a bound $F(d)$ on the number of discrete events arriving in any interval of length d. For instance, α satisfies lower bound arrival curve F if for all $t, d \in \mathbb{R}^{\geq 0}$, $\alpha(t+d) - \alpha(t) \geq F(d)$. Similar bounding functions could be defined for event sequences.

In NC and RTC, arrival curves are used as a representation of a set of arrival functions, namely those arrival functions whose arrival curves are bounded by the given curve. The strong result of NC and RTC is that just as services may operate on concrete arrival functions, they can operate on upper or lower bound arrival curves and yield upper or lower bound curves characterizing the set of concrete outputs.

Common operations on arrival functions are convolutions. The $(\min, +)$ convolution operation on arrival functions is defined as:

$$(\alpha_1 \otimes \alpha_2)(t) = \inf_{0 \leq s \leq t} \{\alpha_1(t-s) + \alpha_2(s)\}$$

In terms of the equivalent event sequence model, the corresponding operation is a $(\max, +)$ convolution (Definition 21):

$$(\tau_1 \otimes \tau_2)(n) = \max_{0 \leq k \leq n} \{\tau_1(n-k) + \tau_2(k)\}$$

To show that this is indeed the equivalent operation, we show that the conversion $\overline{\cdot}$ and the convolution operations commute. For this, we first prove the following lemma.

Lemma 13. *Let τ_1 and τ_2 be non-zeno event sequences and α_1 and α_2 the corresponding arrival functions. Then*

$$(\tau_1 \otimes \tau_2)(n) > t \Leftrightarrow (\alpha_1 \otimes \alpha_2)(t) < n+1.$$

Proof. ($\Rightarrow$) If $(\tau_1 \otimes \tau_2)(n) > t$, then by definition of the convolution, there exists l such that $\tau_1(n-l)+\tau_2(l) > t$. Let l be such, then there exists some s, $0 \leq s \leq t$ such that $\tau_1(n-l) > t-s$ and $\tau_2(l) > s$. By Lemma 6, $\alpha_1(t-s) \leq n-l$ and $\alpha_2(s) \leq l$. Then $\alpha_1(t-s)+\alpha_2(s) \leq n < n+1$ and thus $(\alpha_1 \otimes \alpha_2)(t) < n+1$. ($\Leftarrow$) If $(\alpha_1 \otimes \alpha_2)(t) < n+1$, then by definition of the convolution, there exists s such that $\alpha_1(t-s)+\alpha_2(s) < n+1$. Let s be such, then there exists some l, $0 \leq l \leq n$ such that $\alpha_1(t-s) \leq n-l$ and $\alpha_2(s) \leq l$. By Lemma 6, $\tau_1(n-l) > t-s$ and $\tau_2(l) > s$. Then $\tau_1(n-l)+\tau_2(l) > t$ and thus $(\tau_1 \otimes \tau_2)(n) > t$. □

Lemma 14. *Let α_1 and α_2 be two arrival functions. Then*

$$\overline{\alpha_1 \otimes \alpha_2} = \overline{\alpha_1} \otimes \overline{\alpha_2}.$$

Proof.

$$\begin{aligned}
\overline{\alpha_1 \otimes \alpha_2}(n) &= \inf\{t \mid n+1 \leq (\alpha_1 \otimes \alpha_2)(t)\} \\
&= \inf\{t \mid t \geq (\overline{\alpha_1} \otimes \overline{\alpha_2})(n)\} \quad \{\text{by Lemma 13}\} \\
&= (\overline{\alpha_1} \otimes \overline{\alpha_2})(n)
\end{aligned}$$

□

The empty sequence ϵ is the zero-element of convolution: $\tau \otimes \epsilon = \epsilon \otimes \tau = \epsilon$. The zero sequence $\vec{0}$ is the unit-element of convolution: $\tau \otimes \vec{0} = \vec{0} \otimes \tau = \tau$.

Similarly, $(\min, +)$ deconvolution is defined as

$$\alpha_1 \oslash \alpha_2(t) = \sup_s \{\alpha_1(t+s) - \alpha_2(s)\}$$

and it similarly becomes in terms of event sequences:

$$\tau_1 \oslash \tau_2(n) = \inf_k \{\tau_1(n+k) - \tau_2(k)\}$$

Lemma 15. *Convolution is commutative and associative.*

Proof. (Similar to [8], Thm.3.1.5 for arrival functions.) Commutativity follows directly from the definition. For associativity, let τ_i, $i = 1, 2, 3$ be event sequences.

$$\begin{aligned}
((\tau_1 \otimes \tau_2) \otimes \tau_3)(n) &= \max_{0 \leq k \leq n} \{(\tau_1 \otimes \tau_2)(n-k) + \tau_3(k)\} \\
&= \max_{0 \leq k \leq n} \{\max_{0 \leq m \leq n-k} \{\tau_1(n-k-m) + \tau_2(m)\} + \tau_3(k)\} \\
&= \max_{0 \leq k \leq n, 0 \leq m \leq n-k} \{\tau_1(n-k-m) + \tau_2(m) + \tau_3(k)\} \\
&= \max_{0 \leq k \leq n, 0 \leq (l-k) \leq n-k} \{\tau_1(n-k-(l-k)) + \tau_2(l-k) + \tau_3(k)\} \\
&= \max_{0 \leq k \leq l \leq n} \{\tau_1(n-l) + \tau_2(l-k) + \tau_3(k)\} \\
&= \max_{0 \leq l \leq n} \{\tau_1(n-l) + \max_{0 \leq k \leq l} \{\tau_2(l-k) + \tau_3(k)\}\} \\
&= \max_{0 \leq l \leq n} \{\tau_1(n-l) + (\tau_2 \otimes \tau_3)(l)\} \\
&= (\tau_1 \otimes (\tau_2 \otimes \tau_3))(n)
\end{aligned}$$

□

Convolution of event sequences is monotone in the earlier relation.

Lemma 16. *Let τ_i, $i = 1, 2, 3$ be event sequences such that $\tau_1 \leq \tau_2$, then $\tau_1 \otimes \tau_3 \leq \tau_2 \otimes \tau_3$ (a corresponding property of arrival functions is called* isotonicity *in [8]).*

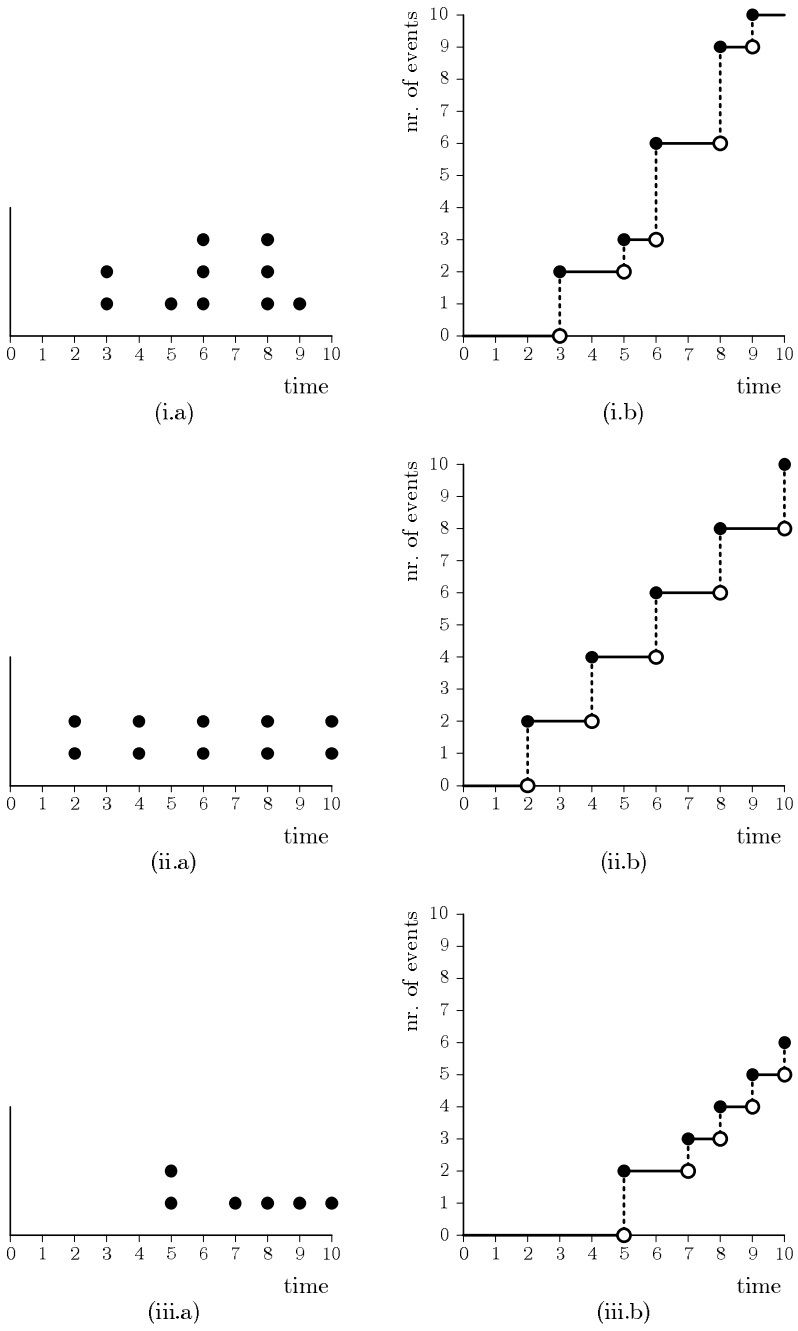


Figure 10: Event sequence (a) and arrival function (b) of input (i), actor service (ii) and their convolutions (iii).

Proof. $(\tau_1 \otimes \tau_3)(n) = \max_{0\le k\le n}\{\tau_1(n-k)+\tau_3(k)\} \le \max_{0\le k\le n}\{\tau_2(n-k)+\tau_3(k)\} = \tau_2 \otimes \tau_3$. It is easy to see that the lengths of both sequences are identical and equal to the length of the shortest τ_i. □

Note that it is not necessarily true in the opposite direction. For instance, $\tau_1 \otimes \epsilon \le \tau_2 \otimes \epsilon$, for any τ_1 and τ_2.

A *service curve* characterizes the amount of service (work) an actor can perform in a given interval of time, or conversely, the amount of time it takes to process a set of input events. Formally, a *service curve* is a function $\beta : \mathbb{N} \to \mathbb{R}^{\ge 0}$, $\beta(n)$ being the time to process n events. Processing n events should not take more time than n times processing a single event. This property of a service curve is called *subadditivity*, a straightforward adaptation of the definition of Boudec and Thiran [8]:

Definition 22. *A service curve β is* subadditive *if for any $n, m > 0$, $\beta(n+m-1) \le \beta(n-1)+\beta(m-1)$.*

Note that if β is subadditive then $\beta(n) < \infty$ for all n if $\beta(0) < \infty$.

Definition 23 (SC actors)**.** *A subadditive service curve β defines an input-complete, non-deterministic actor $\mathsf{C}^\beta = (\{p\}, \{q\}, R_{\mathsf{C}^\beta})$, called an* SC actor*, and such that $x\mathsf{C}^\beta y$ iff $y(q) \le x(p) \otimes \beta$.*

The intuition is as follows. $\beta(n)$ is an upper bound on the time it takes the server to process n data items assuming they are available for processing. An upper bound on the completion time for n output data items is given by the worst case combination of the arrival time of data item $n-k$ and the time it takes to process the remaining k items. Note that, in the expression $x(p) \otimes \beta$, even though $x(p)$ and β are functions with (almost) the same signature, they are objects with different meaning: $x(p)$ is an event sequence, whereas β is a service curve.

In order to effectively use service curves, a finite representation is needed. Often piece-wise linear functions or a finite set of discrete points are used (see http://www.mpa.ethz.ch/Rtctoolbox). With such representations in hand, problems on SC actors can be reduced to problems on service curves. For instance, $\mathsf{C}^{\beta_1} = \mathsf{C}^{\beta_2}$ iff $\beta_1 = \beta_2$.

Proposition 15. *Let β_1 and β_2 be two service curves, then $\beta_2 \le \beta_1$ iff $\mathsf{C}^{\beta_2} \sqsubseteq \mathsf{C}^{\beta_1}$.*

Proof. (only if) Both are input-complete, so $\text{in}_{\mathsf{C}^{\beta_1}} \subseteq \text{in}_{\mathsf{C}^{\beta_2}}$. Let $x\mathsf{C}^{\beta_2}y$, then $y(q) \le x(p) \otimes \beta_2$. It suffices to show that $y(q) \le x(p) \otimes \beta_1$ and thus that $x(p) \otimes \beta_2 \le x(p) \otimes \beta_1$, which follows directly from $(x(p) \otimes \beta_2)(n) = \max_{0\le k\le n}\{x(p)(n-k)+\beta_2(k)\} \le \max_{0\le k\le n}\{x(p)(n-k)+\beta_1(k)\} = (x(p) \otimes \beta_2)(n)$. □

Serial composition of SC actors amounts to convolution of their service curves. This is similar to NC and RTC where composition is also defined as convolution.

Proposition 16. *Let C^{β_1} be an SC actor with input port p_1 and output port q_1 and C^{β_2} an SC actor with input port p_2 and output port q_2, then with θ the obvious mapping, $\mathsf{C}^{\beta_1}\theta\mathsf{C}^{\beta_2} = \mathsf{C}^{\beta_1 \otimes \beta_2}$.*

Proof. We need to show that (i) if $x\mathsf{C}^{\beta_1}\theta\mathsf{C}^{\beta_2}y$, then $x\mathsf{C}^{\beta_1 \otimes \beta_2}y$ and (ii) vice versa. For (i), let $x\mathsf{C}^{\beta_1}\theta\mathsf{C}^{\beta_2}y$, then there exist y_1 and x_2 such that $x\mathsf{C}^{\beta_1}y_1$ and $x_2\mathsf{C}^{\beta_2}y$. Then $y_1(q_1) \le x(p) \otimes \beta_1$ and $y(q) \le x_2(p_2) \otimes \beta_2$ and $x_2(p_2) = y_1(q_1)$. Using Lemma 16 and associativity of convolution, $y(q) \le x_2(p_2) \otimes \beta_2 \le (x(p) \otimes \beta_1) \otimes \beta_2 = x(p) \otimes (\beta_1 \otimes \beta_2)$ and thus $x\mathsf{C}^{\beta_1 \otimes \beta_2}y$.

(ii). $x\mathsf{C}^{\beta_1 \otimes \beta_2}y$. $y(q) \le x(p) \otimes (\beta_1 \otimes \beta_2)$. Let y_1 be defined by $y_1(q_1) = x(p) \otimes \beta_1$. Then clearly $x\mathsf{C}^{\beta_1}y_1$. Moreover, $y(q) \le y_1(q_1) \otimes \beta_2$, because $y(q) \le x(p) \otimes \beta_1 \otimes \beta_2$ and thus with $x_2(p_2) = y_1(q_1)$, $x_2\mathsf{C}^{\beta_2}y$. □

9.3 Discrete-Time Automata

A natural representation of actors is automata. Automata, in contrast with SDF actors, do not have $\sqsubseteq$-monotonicity and input-closure built-in, and such properties have to be explicitly verified when necessary. There are many automata variants, over finite or infinite words, with various acceptance conditions, finite or infinite-state,[2] and so on. We are not going to propose a single automaton-based model for actors. Instead

[2] We do require finite representations, but a finite representation (e.g., a counter machine) can still represent an infinite-state system.

we will discuss some general ideas as well as some cases for which we have algorithms. We limit our discussion to *discrete-time* automata (DTA) in the sense that time is counted by discrete transitions. DTA generate actors over a discrete time domain, $\mathcal{T} = \mathbb{N}$. One could also consider timed automata [2] where $\mathcal{T} = \mathbb{R}^{\geq 0}$.

One possible model is an automaton whose transitions are labeled with subsets of $P \cup Q$, the set of input or output ports. An example is shown in Figure 11. The state drawn with two circles is the *accepting* state. In this *implicit-tick model* each transition corresponds to one time unit. If the transition is labeled by some set of ports $V \subseteq P \cup Q$, then an event at each port in V occurs at the corresponding instant in $\mathbb{N}$. V may be empty, as in the self-loop transition of the automaton shown in the figure. In this case, no events occur at that time instant.[3]

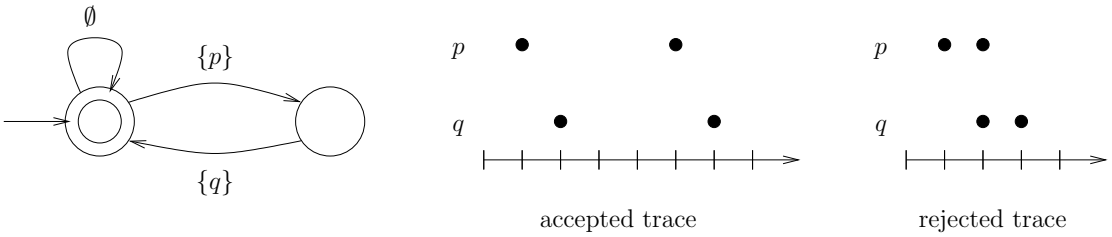


Figure 11: Finite-state automaton labeled with subsets of $\{p, q\}$ (left) and two possible event traces (middle and right). The automaton accepts the leftmost trace but not the rightmost one.

The implicit-tick model cannot capture event traces where more than one event occurs simultaneously at the same port (and therefore an implicit-tick actor cannot be input-complete). One possibility is to dissociate the elapse of time from transitions, by introducing a special label, $\mathtt{t}$, denoting one time unit. Then, we can use automata whose transitions are labeled with single ports or $\mathtt{t}$, that is, whose alphabet is $P \cup Q \cup \{\mathtt{t}\}$. We call this the *explicit-tick model*. An example is shown in Figure 12. This automaton generates strictly more event traces than the one in Figure 11.

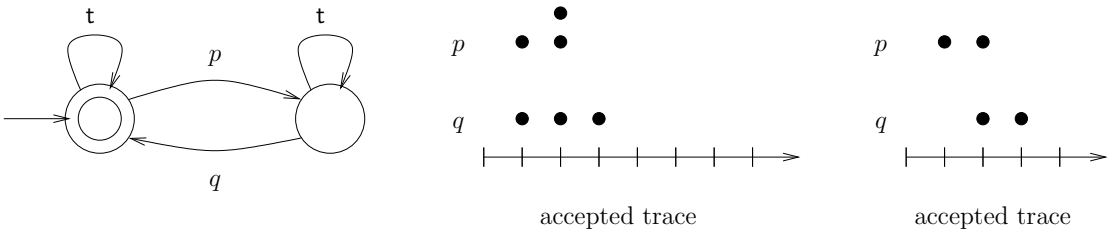


Figure 12: Finite-state automaton labeled with elements of $\{p, q, \mathtt{t}\}$ (left) and two possible event traces accepted by the automaton (middle and right).

A DTA M defines an actor $A(M) = (P, Q, R_{A(M)})$ as follows. Every (finite or infinite) accepting run of M generates a (finite or infinite) word w in the language of M, denoted $\mathcal{L}(M)$. Every word w can be mapped to a unique event trace pair $Tr(w) \in Tr(P) \times Tr(Q)$, as illustrated in the figures above. Then, $R_{A(M)}$ is the set of all event trace pairs generated by words in $\mathcal{L}(M)$, i.e., the set $\{Tr(w) \mid w \in \mathcal{L}(M)\}$, also denoted $Tr(\mathcal{L}(M))$.

An *ITA* (resp. *ETA*) is an implicit-tick (resp. explicit-tick) automaton on finite words. An *ITBA* (resp. *ETBA*) is an implicit-tick (resp. explicit-tick) Büchi automaton. Implicit-tick DTA are (strictly) less

[3] Note that automata in the implicit-tick model are essentially *transducers* [6] with input alphabet 2^P and output alphabet 2^Q. Every transition in such a transducer is labeled by a pair (P', Q') with $P' \subseteq P$ and $Q' \subseteq Q$. Often such pairs are denoted P'/Q', a notation that the advantage of being explicit about inputs and outputs. In the automaton shown in Figure 11, for instance, the labels $\{p\}$ and $\{q\}$ would be written as $\{p\}/\emptyset$ and $\emptyset/\{q\}$, respectively. Label $\emptyset$ would be written as $\emptyset/\emptyset$.

expressive than explicit-tick DTA and finite-word DTA are strictly less expressive than corresponding Büchi versions:

Proposition 17. *Every ITA (resp. ITBA) M can be transformed into an ETA (resp. ETBA) M' such that $A(M) = A(M')$. Every ITA (resp. ETA) M can be transformed into an ITBA (resp. ETBA) M' such that $A(M) = A(M')$.*

Proof. For the first claim, irrespectively of whether M is an ITA or ITBA, a transition $s \xrightarrow{P' \cup Q'} s'$ in M, with $P' = \{p_1, p_2, ..., p_n\} \subseteq P$ and $Q' = \{q_1, q_2, ..., q_m\} \subseteq Q$, can be transformed into a sequence of transitions

$$s \xrightarrow{p_1} r_1 \xrightarrow{p_2} \cdots \xrightarrow{p_n} r_n \xrightarrow{q_1} r_{n+1} \xrightarrow{q_2} \cdots \xrightarrow{q_m} r_{n+m} \xrightarrow{\mathtt{t}} s'$$

in M', where $r_1, ..., r_{n+m}$ are new, non-accepting states (created for this transition).

For the second claim, if M is an ITA, we add a new state s_{new} with a self-loop $s_{new} \xrightarrow{\emptyset} s_{new}$. From any accepting state s of M, we add a transition $s \xrightarrow{\emptyset} s_{new}$. We turn s into non-accepting and s_{new} into accepting. The obtained ITBA M' has a single accepting state, s_{new}. This transformation ensures that $A(M) = A(M')$ because $Tr(w) = Tr(w \cdot \emptyset^\omega)$ for any finite word w.

For an ETA, the transformation is essentially the same, except that the new transitions are labeled with $\mathtt{t}$ instead of $\emptyset$. □

Two distinct words w and w' of an ITA M can result in the same event trace, for instance, if $w' = w \cdot \emptyset^n$, for different $n \geq 1$. To avoid technical complications related to this, we will assume that M is *tick-closed*, that is, for any $w \in \mathcal{L}(M)$, $w \cdot \emptyset^* \subseteq \mathcal{L}(M)$. An ETA M is tick-closed iff for any $w \in \mathcal{L}(M)$, $w \cdot \mathtt{t}^* \subseteq \mathcal{L}(M)$. The ITA of Figure 11 and ETA of Figure 12 are tick-closed. Any ITA (ETA) M can be transformed to a tick-closed ITA (ETA) M' so that $A(M) = A(M')$. In the rest of this section we assume all ITA or ETA to be tick-closed. We also assume that automata are finite-state, all their states are reachable from the initial state, and there is no state that cannot reach an accepting state by a non-empty path.

Given actors represented by discrete-time automata, we are interested in answering various questions.

9.3.1 Checking equality

Given M and M', is $A(M) = A(M')$? For ITA, there is a bijection between infinite words and event traces, that is, $\forall w, w' \in (2^{P \cup Q})^\omega : w \neq w' \iff Tr(w) \neq Tr(w')$. (Note that $Tr(w)$ may be finite, even though w is infinite, if w ends in $\emptyset^\omega$.) The same bijection does not hold for finite words as explained above. Nevertheless, because ITA are assumed to be tick-closed, we can show:

Proposition 18. *For two ITA (ITBA) M_1 and M_2, we have $A(M_1) = A(M_2)$ iff $\mathcal{L}(M_1) = \mathcal{L}(M_2)$.*

For ETA, we cannot reduce actor equivalence to language equivalence, even when the automata are tick-closed, since we can obtain the same event trace from different words by commuting transitions happening between two successive ticks. To solve this problem we use results from *trace language theory* [18]. Consider an ETA M with input and output ports P and Q, respectively. $\Sigma = P \cup Q \cup \{\mathtt{t}\}$ is a finite alphabet. We define $I = (P \cup Q) \times (P \cup Q)$ to be an irreflexive and symmetric binary relation on Σ, called an *independence relation*. If $(a, b) \in I$ then the letters a and b are independent, meaning they can be swapped in a word without changing the trace. I defines an equivalence relation $\sim$ on Σ^*, such that two words $w, u \in \Sigma^*$ are equivalent iff w can be obtained from u by repeatedly commuting adjacent independent letters. The equivalence class of word w w.r.t. $\sim$ is $[w]$. Equivalence classes $[w]$, are called *traces* in trace theory (not to be confused with our event traces). A set T of such traces is called a *trace language*. If $T = [L] = \{[w] \mid w \in L\}$ for some regular language $L \subseteq \Sigma^*$, then T is called a *rational* trace language. A trace language defined by an ETA is a rational trace language.

Observe that, for any two words $w, u \in \Sigma^*$, if $w \sim u$ then $Tr(w) = Tr(u)$. The converse does not generally hold because of trailing $\mathtt{t}$ letters at the end of the word, which do not impact the corresponding event trace. If both w and u end with the same number of $\mathtt{t}$'s, however, then $Tr(w) = Tr(u)$ implies $w \sim u$. Based on this and tick-closure, we can state the following:

Lemma 17. *For two ETA M_1 and M_2, we have $A(M_1) = A(M_2)$ iff $[\mathcal{L}(M_1)] = [\mathcal{L}(M_2)]$.*

The problem to check $[\mathcal{L}(M_1)] = [\mathcal{L}(M_2)]$ is known as the equivalence problem for rational trace languages. It is known that this problem is decidable iff the independence relation I is transitive (see Ch.5 of [18]). Fortunately, this is true in our case. Therefore, checking equality of ETA actors is decidable.

9.3.2 Checking determinism and input-completeness

Given M, is $A(M)$ deterministic? An automaton M is deterministic if for any state s of M, and for any label a, there is at most one outgoing transition from s labeled a. Determinism of M does not imply determinism of $A(M)$, as Figure 13 illustrates: in the figure, q_1 and q_2 are assumed to be distinct output ports. $A(M)$ is non-deterministic iff there are two words $w, w' \in \mathcal{L}(M)$, with $(x, y) = Tr(w)$ and

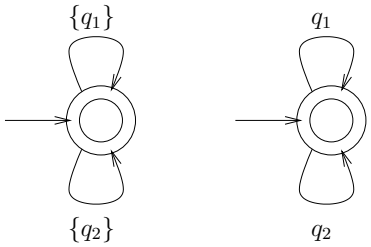


Figure 13: Deterministic finite-state automata generating non-deterministic actors.

$(x', y') = Tr(w')$, such that $x = x'$ but $y \neq y'$.

Proposition 19. *For any ITA or ITBA M it is decidable whether $A(M)$ is deterministic.*

Proof. Suppose M is a (not necessarily deterministic) ITBA. We compute the synchronous product of M with itself, where the two copies synchronize on the input ports. That is, a transition $\xrightarrow{P_1 \cup Q_1}$ in one copy must synchronize with a transition $\xrightarrow{P_2 \cup Q_2}$ in the other copy such that $P_1 = P_2$. Q_1 and Q_2 may be different, but when they are different, a boolean flag is set from false to true, and remains at true forever after. We claim that $A(M)$ is deterministic iff there exists an infinite run in the product such that: (1) the flag is true after some point on; and (2) the run is accepting for both copies of M. Finding such a run can be done using standard model-checking techniques. □

Given M, is $A(M)$ input-complete? Finite-word automata cannot generate input-complete actors because they cannot generate infinite event traces. Implicit-tick automata cannot generate input-complete actors either, since they do not allow multiple simultaneous events at the same port.

For an ETA or ETBA M, a simple sufficient condition for $A(M)$ to be input-complete is that all states of M are accepting, and every state s has a transition labeled a for every $a \in P \cup Q \cup \{\mathtt{t}\}$.

9.3.3 Computing compositions and hiding

Given M with output ports Q, and given $Q' \subseteq Q$, compute M' so that $A(M') = A(M) \backslash Q'$. If M is an ITA or an ITBA, then M' can be obtained by replacing each transition $\xrightarrow{P' \cup Q''}$ of M with a transition $\xrightarrow{P' \cup Q'' \backslash Q'}$.

If M is an ETA, then M' can be obtained by replacing each transition $\xrightarrow{q}$ of M such that $q \in Q'$ with a "silent" ("epsilon") transition, and then removing the silent transitions by determinizing the resulting automaton. If M is an ETBA then a similar procedure can be applied, however, since Büchi automata are not determinizable in general, the resulting automaton may contain silent transitions.

Given M_1 and M_2, compute M so that $A(M) = A(M_1)||A(M_2)$. M can be computed as a product of M_1 and M_2, so that $\mathcal{L}(M)$ contains exactly those words w such that $Tr(w) = Tr(w_1) \cup Tr(w_2)$, for $w_i \in \mathcal{L}(M_i)$ and $i = 1, 2$. If M_1 and M_2 belong to the implicit-tick model, M is a synchronous product, so that a pair of transitions $\xrightarrow{P_1 \cup Q_1}$ of M_1 and $\xrightarrow{P_2 \cup Q_2}$ of M_2 yields a transition $\xrightarrow{P_1 \cup P_2 \cup Q_1 \cup Q_2}$ in the product. If M_1 and M_2 belong to the explicit-tick model, then M can be computed as the product of M_1 and M_2, where only transitions labeled with $\mathtt{t}$ synchronize, while transitions labeled with ports interleave.

Given M_1 and M_2, and a bijection θ from the output ports of M_1 to the input ports of M_2, compute M so that $A(M) = A(M_1)\theta A(M_2)$. Feeding the output of M_1 (after relabeling) into the input of M_2 can be achieved by a product of both automata synchronizing on the corresponding ports. The main challenge in computing the serial composition is to ensure that the constraint in_θ is satisfied (see Definition 9). We assume M_1 and M_2 are ITA. We construct the composite automaton M as the synchronous product of M_1, M_2 and a third automaton M_{in} capturing the constraint in_θ. We construct M_{in} as an alternating automaton that can then be converted into a non-deterministic automaton using standard techniques.

The constraint in_θ ensures that the composite actor accepts only those valid input traces of M_1 for which no matter what output trace M_1 produces, this is a valid input trace to M_2 after relabeling by θ. Since we are only interested in the inputs accepted by M_2, we construct automaton M_2' from M_2 by hiding the output events and determinizing the result. M_2' has input ports P_2 and no output ports. Let $M_2' = (S_2, s_{2,0}, \delta_2, F_2)$, i.e., M_2' has states S_2, initial state $s_{2,0}$, transition function $\delta_2 : S_2 \times 2^{P_2} \to S_2$, and accepting states F_2. We assume that δ_2 is total, i.e., defined for every $s_2 \in S_2$ and every $P' \subseteq P_2$ (we can always add a "dummy" rejecting state to achieve this). Let $M_1 = (S_1, s_{1,0}, \delta_1, F_1)$, with input ports P_1 and output ports Q_1. δ_1 is not necessarily total (recall that every state of M_1 must have a path to an accepting state). For $P' \subseteq P_1$, let $\mathcal{Q}(s, P')$ denote the set $\{Q' \subseteq Q_1 \mid \delta_1(s, P' \cup Q') \text{ is defined}\}$, i.e., the possible outputs for input events P'.

M_{in} has states $S = S_1 \times S_2$, accepting states $F = F_1 \times F_2$, and initial state $(s_{1,0}, s_{2,0})$. Its alphabet is 2^{P_1}. Its transition function encodes the condition that for state (s_1, s_2) and input P', any output Q' produced by M_1 is also accepted as an input by M_2' (after relabeling by θ), and any remainder of the output produced by M_1 is also recursively accepted by M_2'. This is captured by the conjunction below, which is trivially accepting if P' is not a valid input for M_1, i.e., if $\mathcal{Q}(s_1, P') = \emptyset$. Formally, for $P' \subseteq P_1$:

$$\delta((s_1, s_2), P') = \bigwedge_{Q' \in \mathcal{Q}(s_1, P')} (\delta_1(s_1, P' \cup Q'), \delta_2(s_2, \theta(Q')))$$

Lemma 18. *Let M_{in} be defined as above for the serial composite $M_1\theta M_2$. Then $x \in Tr(\mathcal{L}(M_{in}))$ iff $x \in \text{in}_{A(M_1)}$ and for every y such that $xA(M_1)y$, $\theta(y) \in \text{in}_{A(M_2)}$ (i.e., iff $x \in \text{in}_\theta$).*

Proof. (sketch) (only if) Let $x \in Tr(\mathcal{L}(M_{in}))$ and let ρ be the run witnessing the acceptance of x: ρ is a tree because M_{in} is an alternating automaton. At any state reached after some prefix of the input word, the number of branches in the tree is at least one by construction of the automaton. Hence, there exists at least one linear path on the automaton M_{in} leading from the initial state to an accepting state. Projected onto M_1, this is an accepting run for input x and therefore, $x \in \text{in}_{A(M_1)}$. Now let y be such that $xA(M_1)y$. Then there exists a run of M_1 with x and y as its inputs and outputs respectively. Tree ρ has a unique path corresponding to input x and output y. Projecting that same path of the tree onto a run of M_2' it follows that $\theta(y) \in \text{in}_{M_2}$.

(if) Let x be an input trace, accepted by M_1, such that for every y for which $xA(M_1)y$, $\theta(y) \in \text{in}_{A(M_2)}$. Then an accepting run on M_{in} can be constructed as follows. At any state (s_1, s_2) reached in the run being constructed, input x determines the next input symbol and the conjunction in the transition function requires that for every possible output Q' of M_1, from s_1, there exists a transition with input $\theta(Q')$ in M_2'. Such a next state exists by the assumption that for every output y of M_1, $\theta(y)$ is an accepted input to M_2 and since M_2 is deterministic, the word $\theta(y)$ must be accepted from the state s_2 and the corresponding transition must exist. The final set of states, reached after consuming the entire input word, must be accepting because, projected on M_1, they represent a valid input output pair x, y and, projected on M_2', $\theta(y)$ is an accepted input of M_2. □

Proposition 20. *For finite-state and deterministic ITA M_1 and M_2 and bijection θ, the above described algorithm computes ITA M such that $A(M) = A(M_1)\theta A(M_2)$.*

Given M, input port p and output port q, compute M' such that $A(M') = A(M)(p = q)$. If M is an ITA or an ITBA, then M' can be easily obtained by removing from M all transitions $\xrightarrow{P' \cup Q'}$ except those that satisfy $p \in P' \iff q \in Q'$.

9.3.4 Checking actor refinement on ITA

Given M_1 and M_2, is $A(M_1) \sqsubseteq A(M_2)$? We show that checking actor refinement on ITA can be reduced to checking language containment with respect to an appropriate closure.

Given automaton M, we construct an (initially infinite state) automaton $M_{\infty\sqsubseteq}$ that recognizes the refinement closure of M, i.e., it accepts all words of M, but also all words that correspond to traces that refine the traces of M. We define $M_{\infty\sqsubseteq}$ for a single output port q, but the construction can be generalized to multiple output ports. Figure 14 shows an example. We add a counter n to count the surplus of q events. Whenever later in a word M requires a q event, we allow this event to be absent and decrease the counter. The following gives a precise definition of this idea, parameterized with a bound k on the counter.

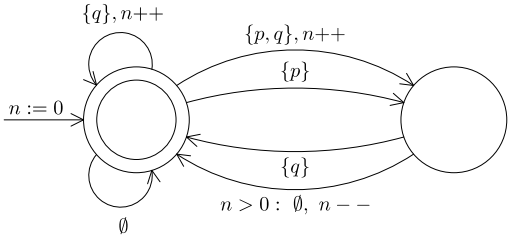


Figure 14: Refinement closure of the automaton of Figure 11.

Definition 24 (Refinement closures)**.** *Let $M = (S, s_0, E, F)$ be an ITA with a single output port q, states S, initial state $s_0 \in S$, accepting states $F \subseteq S$, and transitions E. For $k \in \mathbb{N}$, the* k-bounded refinement closure *of M is the automaton $M_{k\sqsubseteq} = (S_{k\sqsubseteq}, s_{k\sqsubseteq,0}, E_{k\sqsubseteq}, F_{k\sqsubseteq})$ such that $S_{k\sqsubseteq} = \{(s, n) \mid s \in S, 0 \le n \le k\}$, $s_{k\sqsubseteq,0} = (s_0, 0)$, and $F_{k\sqsubseteq} = \{(s, n) \in S_{k\sqsubseteq} \mid s \in F\}$. For every transition $(s_1, \sigma, s_2) \in E$, we have the following transitions in $E_{k\sqsubseteq}$:*

$$\begin{array}{cc} \big((s_1, n), \sigma, (s_2, n)\big) & \text{if} \quad 0 \le n \le k \\ \big((s_1, n), \sigma \cup \{q\}, (s_2, n+1)\big) & \text{if} \quad 0 \le n < k, q \notin \sigma \\ \big((s_1, n), \sigma \cup \{q\}, (s_2, n)\big) & \text{if} \quad n = k, q \notin \sigma \\ \big((s_1, n), \sigma \backslash \{q\}, (s_2, n-1)\big) & \text{if} \quad 0 < n \le k, q \in \sigma \end{array}$$

The (unbounded) refinement closure *of M is the automaton $M_{\infty\sqsubseteq}$ defined in a similar way, but where counter n is unbounded.*

Lemma 19. *Let M_1 and M_2 be ITA with the same input ports P and the same, single output port q. Then $A(M_1) \sqsubseteq A(M_2)$ iff for every $w \in \mathcal{L}(M_1)$ such that $Tr(w) = (x, y)$ and $x \in \text{in}_{A(M_2)}$, $w \in \mathcal{L}(M_{2,\infty\sqsubseteq})$.*

Proof. (only if) If w is a word of M_1 with inputs $Tr(w) = (x, y)$ and x is also accepted by M_2, then there is a word w' of M_2 which is refined by w. Let ρ be the run of M_2 that accepts w'. Then there is a run for w in the closure $M_{2,\infty\sqsubseteq}$, by taking the corresponding transitions of ρ, while increasing/decreasing the counter n if necessary to match the presence/absence of output events q. The counter cannot become negative, because $w \sqsubseteq w'$.

(if) Let w be accepted by $M_{2,\infty\sqsubseteq}$ by a run ρ. We can project ρ onto M_2 and we find that the corresponding word w' in M_2 is refined by w. □

Unfortunately, $M_{2,\infty\sqsubseteq}$ is not a finite-state automaton, in fact, $\mathcal{L}(M_{2,\infty\sqsubseteq})$ is not always regular. Let $\mathcal{L}(M) = (\{p,q\}\cdot\emptyset)^*$, where p is the only input port and q the only output port. Then $\mathcal{L}(M_{\infty\sqsubseteq})$ contains all words of the form $(\{p,q\}\cdot\{q\})^n\cdot\{p\}^n$, for any $n\in\mathbb{N}$. Based on this, we can show that $\mathcal{L}(M_{\infty\sqsubseteq})$ is not regular. Despite this difficulty, we can make use of the finite memory property of M_1 and M_2 to find an upper bound on the size of the refinement closure, which proves that checking refinement for ITA is decidable:

Proposition 21. *Let M_1 and M_2 be deterministic ITA with the same input ports P and the same, single output port q. $A(M_1)\sqsubseteq A(M_2)$ iff for every $w\in\mathcal{L}(M_1)$ such that $Tr(w)=(x,y)$ with $x\in \text{in}_{A(M_2)}$, $w\in\mathcal{L}(M_{2,N\sqsubseteq})$, where $N=|S_1|\times|S_2|$.*

Proof. Let $M_i=(S_i,s_{i,0},E_i,F_i)$. The 'if' part follows from Lemma 19 and the fact $\mathcal{L}(M_{2,N\sqsubseteq})\subseteq\mathcal{L}(M_{2,\infty\sqsubseteq})$.

For the 'only if' part, let w be a word of M_1, accepted by run ρ_1 with $Tr(w)=(x,y)$ such that x is also accepted by M_2. Then, by Lemma 19, w is accepted by $M_{2,\infty\sqsubseteq}$, by accepting run ρ_∞. Assume towards a contradiction that w is not accepted by $M_{2,N\sqsubseteq}$. M_2 is deterministic, therefore so is $M_{2,N\sqsubseteq}$. Let c and c_∞ be the counters of $M_{2,N\sqsubseteq}$ and $M_{2,\infty\sqsubseteq}$, respectively. Since w is not accepted by $M_{2,\infty\sqsubseteq}$, the unique run ρ_N of $M_{2,N\sqsubseteq}$ generated by w eventually "deadlocks" (i.e., is not able to complete w) with $c=0$. Moreover, because w is accepted by $M_{2,\infty\sqsubseteq}$, c must have previously reached N in ρ_N. Let $w=w_1\cdot w_2\cdot w_3$, such that at the beginning of w_2, $c=N$ in ρ_N, $c_\infty>N$ in ρ_∞, and the end of w_2 is the point where $M_{2,N\sqsubseteq}$ deadlocks. Since $c=N$ at the beginning of w_2 and $c=0$ at its end, the length of w_2 is at least N. Observing the runs ρ_1 and ρ_∞ side-by-side, since $N=|S_1|\times|S_2|$, there must exist a recurring pair of states s_1 in ρ_1 and (s_∞,n_1), (s_∞,n_2) in ρ_∞ with $n_2<n_1$. On this fragment of w_2, M_1 and M_2 both complete a cycle and the net number of output events in M_1 is smaller than in M_2. The fact that this cycle can be repeated arbitrarily many times, after which an accepting state can be reached, contradicts the hypothesis $A(M_1)\sqsubseteq A(M_2)$. □

9.3.5 Computing throughput on ITBA

Given ITBA M with sets of input and output ports P and Q, respectively, and given an output port $q\in Q$ and an input trace $x\in Tr(P)$, we want to compute $T^{lb}(A(M),x,q)$ and $T^{ub}(A(M),x,q)$. To do this, we need a finite representation for x. A natural choice is to represent x as a deterministic ITBA M_x that only refers to ports in P. We require that M_x generates a single trace x. These assumptions imply that M_x has the form of a "lasso".

First, we compute a product of M and M_x such that the two automata synchronize on inputs. We remove from the product all strongly connected components (SCCs) that contain no accepting state of M and denote the result by M'.

We assign a weight to each transition $\xrightarrow{P'/Q'}$ of M': weight 1 if $q\in Q'$ and weight 0 otherwise. With these weights M' can be viewed as a weighted directed graph. We run Karp's algorithm [26] to compute the minimum cycle mean and maximum cycle mean of M', denoted $\text{MCM}_{\min}$ and $\text{MCM}_{\max}$, respectively. $\text{MCM}_{\min}$ (resp., $\text{MCM}_{\max}$) is the minimum (resp., maximum) over all simple cycles κ in M'' of the ratio $\frac{w(\kappa)}{|\kappa|}$, where $w(\kappa)$ is the sum of weights of all transitions in κ and $|\kappa|$ is the length of κ (i.e., the number of transitions in κ).

Proposition 22. *$T^{lb}(A(M),x,q)=MCM_{\min}$ and $T^{ub}(A(M),x,q)=MCM_{\max}$.*

Proof. (sketch) The tricky part is that Karp's algorithm considers all cycles, including non-accepting (in the Büchi sense) cycles. However, all cycles are guaranteed to belong to an accepting SCC (otherwise the SCC is removed by construction of M'). Then, from any cycle it is possible to reach an accepting state and then return to the cycle. This "detour" may increase the throughput by some amount ε, however, ε can be made arbitrarily small by taking the detour very infrequently (but infinitely often, to be accepting). □

9.3.6 Computing latency on IT(B)A

Given automaton M with sets of input and output ports P and Q, and given an IOES $\mathcal{E}$, we would like, first, to check that $\mathcal{E}$ is valid for $A(M)$, and second, to compute the latency $D^{\mathcal{E}}(A(M))$. To do this, we need

a finite representation for $\mathcal{E}$. Many choices exist for this. We will consider a simple one, where $\mathcal{E}$ is defined by a single tuple $(P', Q', k, \ell) \in 2^P \times 2^Q \times \mathbb{N} \times \mathbb{N}$, so that

$$\mathcal{E} = \{(\{(p, k) \mid p \in P'\}, \{(q, \ell) \mid q \in Q'\})\}.$$

This measures the latency between the moment where k tokens arrive at each of the ports in P' until the moment where ℓ tokens arrive at each of the ports in Q'. We assume that M is an ITA or ITBA.

$\mathcal{E}$ is invalid for $A(M)$ iff there exist traces x, y such that (1) $(x, y) \in A(M)$; (2) for all $p \in P'$, $x(p)(k) \neq \infty$; and (3) there exists $q \in Q'$ such that $y(q)(\ell) = \infty$. This is equivalent to the following property: there exists an accepting run ρ of M and a port $q \in Q'$, such that ρ contains: (a) at least k occurrences of each $p \in P'$; and (b) strictly less than ℓ occurrences of q. This property can be encoded and checked using standard model-checking techniques.

We turn to the problem of computing $D^{\mathcal{E}}(A(M))$. First note that we can check, for a given $d \in \mathbb{N}$, whether $D^{\mathcal{E}}(A(M)) > d$. This question can be formulated as a model-checking question as above, namely, finding an accepting run ρ of M such that: (a) ρ contains at least k occurrences of each $p \in P'$; (b) the latest of these occurrences happens at the j-th transition of ρ; and (c) there exists $q \in Q'$ such that ρ contains strictly less than ℓ occurrences of q until its $(j + d)$-th transition. It then suffices to find the minimum d for which $D^{\mathcal{E}}(A(M)) > d$ is false. Such a minimum is guaranteed to exist, because M is finite state and $\mathcal{E}$ is assumed to be valid.

9.3.7 Computing buffer capacities on IT(B)A

We are given ITA or ITBA M, input port $p \in P$, and input trace x captured as a lasso IT(B)A M_x as explained in Section 9.3.5. We want to compute $\mathsf{B}(A(M), x, p)$. M must have an output consumption port p' corresponding to p, as explained in Section 8.

We first compute the ITBA M' representing the serial composition of M_x and M. It is straightforward to show that $\mathsf{B}(A(M), x, p) = \infty$ iff M' has a cycle in which there are more productions than consumptions, i.e., more transitions containing p than transitions containing p'. After checking that this is not the case, we know that every cycle must have an equal number of p's and p''s: otherwise, the number of p''s is larger, which means that there exist behaviors where consumptions at p exceed productions. We add to M' a counter that counts the number of p's observed, minus the number of p''s. The counter cannot become negative and the counter is bounded because on every cycle of the original automaton the number of p events and p' events are equal. Then $\mathsf{B}(A(M), x, p)$ is equal to the largest value the counter may reach in M'.

10 Discussion and Future Work

We have proposed an interface theory for timed actors with a refinement relation based on the earlier-is-better principle, suitable for worst-case performance analysis. Our framework is compositional and unifies existing formalisms, allowing different types of models to be used in the same design process. For example, automata models could refine SDF models.

The earlier-is-better principle may not seem directly applicable in scenarios where outputs should be produced not too late but not too early either. One way to handle these systems is by using external components that buffer the outputs in case they arrive early, and reproduce them at the right time. This is, for instance, the role of the DAC component discussed in Section 2. If such a buffer is finite, there is a bound to how much earlier the output can be produced. Finite buffers can be captured by back-pressure dependencies like those in Figure 1.

An alternative approach is to combine the earlier-is-better refinement with the corresponding *later-is-better* version, obtained by replacing $y \sqsubseteq y'$ by $y' \sqsubseteq y$ in Condition (2) of Definition 14. Separate specifications could then be used, expressing upper and lower bounds on timing behavior, and refined using the earlier- or later-is-better relation, respectively. Examining in detail this hybrid approach is part of future work.

Other directions for future work include examining the algorithmic complexity of the various computational problems introduced above and coming up with practically useful algorithms; implementing the

algorithms and performing experiments; integrating new representations under our framework; and having a thorough comparative study of the different representations, for instance, in terms of expressiveness and complexity.

Acknowledgments

We would like to thank Prof. Edward Lee and all members of the dataflow study group at UC Berkeley for their valuable feedback. This work was performed in part while Marc Geilen was visiting UC Berkeley in Spring 2010 as a Mackay Fellow.

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC) and the following companies: Bosch, National Instruments, Thales, and Toyota.

References

[1] G. Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, 1990.

[2] R. Alur and D. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126:183–235, 1994.

[3] R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *CONCUR*, 1998.

[4] F. Baccelli, G.J. Olsder, J.-P. Quadrat, and G. Cohen. *Synchronization and linearity. An algebra for discrete event systems.* Wiley, 1992.

[5] R-J. Back and J. Wright. *Refinement Calculus.* Springer, 1998.

[6] J. Berstel. *Transductions and context-free languages.* Teubner, 1979.

[7] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static dataflow. *IEEE Tran. on Signal Processing*, 44(2), 1996.

[8] J.-Y. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet.* Springer, 2001.

[9] G. Buttazzo. *Hard Real-time computing systems.* Kluwer, 1997.

[10] S. Cattani and M. Kwiatkowska. A refinement-based process algebra for timed automata. *Form. Asp. of Computing*, 17(2):138–159, 2005.

[11] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[12] B.A. Davey and H.A. Priestley. *Introduction to lattices and order.* Cambridge University Press, 2002.

[13] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In *HSCC*, pages 91–100, 2010.

[14] L. de Alfaro and T. Henzinger. Interface automata. In *Foundations of Software Engineering (FSE).* ACM Press, 2001.

[15] L. de Alfaro, T. A. Henzinger, and M. I. A. Stoelinga. Timed interfaces. In *EMSOFT*, pages 108–122, 2002.

[16] L. de Alfaro, R. Majumdar, V. Raman, and M. Stoelinga. Game refinement relations and metrics. *L. Meth. in Comp. Sc.*, 4(3), 2008.

[17] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer-Verlag, 1974.

[18] V. Diekert and G. Rozenberg (eds.). *The Book of Traces*. World Scientific, 1995.

[19] A.H. Ghamarian, S. Stuijk, T. Basten, M.C.W. Geilen, and B.D. Theelen. Latency minimization for synchronous data flow graphs. *Digital Systems Design, Euromicro Symposium on*, pages 189–196, 2007.

[20] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *IET CDT*, 3(5), 2009.

[21] T.A. Henzinger and S. Matic. An interface algebra for real-time components. In *RTAS*, pages 253 – 266, 2006.

[22] H. Hermanns, U. Herzog, and J-P. Katoen. Process algebra for performance evaluation. *Theor. Comput. Sci.*, 274(1-2):43–87, 2002.

[23] Thomas T. Hildebrandt, Prakash Panangaden, and Glynn Winskel. A relational model of non-deterministic dataflow. *Mathematical Structures in Computer Science*, 14(5):613–649, 2004.

[24] B. Jonsson and W. Yi. Testing preorders for probabilistic processes can be characterized by simulations. *Theor. Comput. Sci.*, 282(1):33–51, 2002.

[25] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, 1974.

[26] R.M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Math.*, 23(3):309–311, 1978.

[27] M. Krichen and S. Tripakis. Conformance Testing for Real-Time Systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[28] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[29] E. A. Lee, X. Liu, and S. Neuendorffer. Classes and inheritance in actor-oriented design. *ACM TECS*, 8(4):1–26, 2009.

[30] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 17(12):1217–1229, 1998.

[31] O. M. Moreira and M. J. G. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP J. on Adv. in Signal Proc.*, 2007.

[32] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[33] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[34] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. *Nordic J. of Computing*, 2(2):250–273, 1995.

[35] J. Sifakis. Use of petri nets for performance evaluation. In *Measuring, Modelling and Evaluating Computer Systems*, pages 75–93, 1977.

[36] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, Inc., Boca Raton, FL, USA, 2009.

[37] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Trans. Comput.*, 57(10):1331–1345, 2008.

[38] S. Tasiran, R. Alur, R. P. Kurshan, and R. K. Brayton. Verifying abstractions of timed systems. In *CONCUR*, pages 546 – 562, 1996.

[39] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *ISCAS*, pages 101–104, 2000.

[40] L. Thiele, E. Wandeler, and N. Stoimenov. Real-time interfaces for composing real-time systems. In *EMSOFT*, pages 34–43, 2006.

[41] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee. Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs. UC Berkeley tech. report available at http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-52.html, 2010.

[42] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee. A theory of synchronous relational interfaces. UC Berkeley tech. report available at http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-45.html, 2010.

[43] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. Di Natale. Implementing Synchronous Models on Loosely Time-Triggered Architectures. *IEEE Transactions on Computers*, 57(10):1300–1314, 2008.

[44] G. Weiss and R. Alur. Automata based interfaces for control and scheduling. In *HSCC*, pages 601–613. Springer, 2007.

[45] M. Wiggers, M. Bekooij, P. Jansen, and G. Smit. Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure. In *RTAS*, pages 281–292, 2007.

[46] M. Wiggers, M. Bekooij, and G. Smit. Monotonicity and run-time scheduling. In *EMSOFT*, pages 177–186, 2009.