Timing is all you need

Susanne Graf, Bengt Jonsson, Behnam Khodabandeloo, Chengzi Huang, Nikolaus Huber, Philipp Rümmer, Wang Yi

Timing is All You Need

Susanne Graf[1,2], Bengt Jonsson[1], Behnam Khodabandeloo[1], Chengzi Huang[1], Nikolaus Huber[1], Philipp Rümmer[1,3], and Wang Yi[1]

[1] Uppsala University, Sweden
[2] Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, France
[3] University of Regensburg, Germany

Abstract. Deterministic models play a crucial role in computer system development, enabling the simulation and verification of system behaviors before Model-Driven Development (MDD) tools transform and compile these models into final implementations. Ensuring determinism is essential to guarantee that the behaviors of the implemented system maintain the properties analyzed in the models. This paper investigates the semantics of deterministic models for data-flow networks, where systems consist of components that compute functions on streams. While Kahn Process Networks (KPN) serve as a well-established semantic theory for time-insensitive deterministic systems, it proves inadequate for systems with time dependent components. To address this limitation, we use the concept of timed streams and develop a fixed-point theory tailored for time-sensitive systems in the style of KPN. This theory serves as the foundation for the MDD tool-chain, known as MIMOS, currently under development in Uppsala.

1 Introduction

Model-Driven Development (MDD) is a software-engineering paradigm, which applies models to raise the level of abstraction at which software is developed and maintained. The use of models can serve many purposes, including to simplify development by abstracting from complexity, to support modularization through a component-based approach, to allow formal analysis supported by a formal semantics of models, to support reliability guarantees, documentation, and others. There are numerous concrete frameworks for MDD in various application domains. In the spectrum of MDD frameworks, one can discern a tradeoff between generality and effectiveness for various tasks such as analysis and code-generation. At one end of the spectrum is the Model-Driven Architecture by OMG, centering around the UML language; its aspiration for generality makes it difficult to provide effective automated support for tasks such as formal verification. At the other end are various domain-specific frameworks that have been developed for a certain application domain, thereby allowing the definition of a precise formal semantics, support efficient code generation, etc. In this spectrum, an interesting framework is the jABC (Java Application Building Center) framework by Tiziana Margaria and Bernhard Steffen [21], which has been developed over the last 30 years. Its original predecessors have been used in a domain-specific fashion to design telecommunication services, decision support systems, and test automation environments. In jABC, models of services and applications are constructed by composing reusable building

blocks (components) into graph structures, which are digested by supporting tools for animation, prototyping, verification, code generation, and so on. Since its presentation in [21], the versatility of this approach has been demonstrated by applications of jABC and its successors (e.g., XMDD [12]) in a vast number of different applications; a recent example is agriculture [2].

For embedded systems, MDD is particularly suitable: one reason being that modeling facilitates verification and validation of systems that interact with their physical environment. A wide range of MDD frameworks focus on embedded systems design in the synchronous design paradigm, Simulink/Stateflow, and Scade. Simulation is maybe the most important verification technique for V&V of Embedded system designs: therefore deterministic execution semantics is important. Since timing is important for embedded systems, time determinism is also important. This holds true even in the presence of concurrency. For instance, the execution semantics of Simulink/Stateflow is completely time deterministic, even though Stateflow has concurrency-like constructs.

Notably, most of the existing approaches are to perform mathematical simulation based on the synchronous hypothesis where software components implement mathematical functions and the computations of functions take zero-time. The recent work on MIMOS [25] attempts to break the synchronous assumption which is not suitable for performance-demanding and predictable applications on complex platforms such as heterogeneous multi-core processors. The goal of MIMOS is to develop an asynchronous design paradigm for building embedded systems, that remain functional- and timing-deterministic as in the synchronous paradigm and in addition, performance- and resource-ware, and also composable, enabling dynamic software updates after deployment.

This paper considers the problem of providing a time deterministic execution semantics to modeling languages for embedded systems. We focus only on data-flow languages despite of the synchronous or asynchronous paradigm. There is a variety of such languages: Lustre, Esterel, Signal, Simulink/Stateflow and also MIMOS, that are all in this category. The focus of Data-flow languages on data computations makes it easy to model algorithms in signal processing, control, etc. Different languages can be equipped with models of timing in many different ways, where the timing model can be tailored to the specific desiderata of an application domain.

The problem of providing a suitable semantic model for data-flow languages has been given some attention in the research community. The starting point for (almost) all such models is the model by Kahn [4], which provides an elegant model of dataflow networks where nodes are used to implement functions on streams, and links to buffer input and output data among functions. Kahn embedded his model in an elegant CPO structure, allowing us to derive the semantics of cyclic networks from semantics of its nodes.

Kahns semantics works for functionally deterministic networks. It cannot be applied out-of-the box to functionally non-deterministic models. One central stumbling block is the non-deterministic merge node. A number of suggestions for generalizing Kahn's model have appeared. One class of model adds explicit timing to data items in streams, resulting in various suggestions for timed streams. It might seem that moving to timed streams trivially results in a deterministic semantics in the domain of timed streams,

under the assumption that nodes themselves are "time deterministic". Intuitively, this means that if the exact timing of all input is given, then the behavior of a time deterministic node is completely determined, and results in a uniquely defined output. This impression is true, but only up to some limit, and under some mildly restrictive assumptions. For instance, Yates [24] produces such a model under the assumption that there is a minimal time quantity, which bounds the delay between reading input and producing output for any node. Such an assumption may be true in the physical world, assuming a minimal size of components and speed-of-light-like arguments. But some languages have a conceptual semantics not conforming to this. For instance, Simulink allows components with instantaneous output, and even allows zero-delay-cycles under some restrictions.

In this paper, we look at some of these border phenomena, and discuss how various proposals for timed stream models can or cannot cope with them. We also propose a new variation of such a time stream model, and discuss its merits and shortcomings. The main guiding principles as proposed in the work on MIMOS, include:

- Main desiderata: functional determinism and time determinism.
- The model is asynchronous to allow flexibility for dynamic software updates.
- The move from synchronous to asynchronous paradigm enables pipeline-parallelism, "faster is generally better" instead of "you must respect a static schedule".

2 Informal Motivation

The non-timed model of Kahn Networks, and the way in which it guarantees determinism is well-known [4], However, the determinism of such Networks is destroyed as soon as nodes can have time dependent behavior, where the time dependency translates into functional non-determinism, e.g. merging two streams. In an untimed setting, we could try to define a merge function recursively by something like:

$$\begin{aligned} f^{try}_{merge}(\varepsilon,\varepsilon) &= \varepsilon \\ f^{try}_{merge}(a \bullet s_1, s_2) &= a \bullet f^{try}_{merge}(s_1, s_2) \\ f^{try}_{merge}(s_1, b \bullet s_2) &= b \bullet f^{try}_{merge}(s_1, s_2) \end{aligned}$$

However, this definition does not give a unique value for $f^{try}_{merge}(\langle a\rangle, \langle b\rangle)$. We can stay in an untimed setting by making the merge unfair, e.g., defining $f^{try}_{merge}(\langle a\rangle, \langle b\rangle) = \langle ab\rangle$. But by monotonicity, we must then define $f^{try}_{merge}(\varepsilon, \langle b\rangle) = \varepsilon$, and in fact $f^{try}_{merge}(\varepsilon, s_2) = \varepsilon$ for any stream s_2. Intuitively, this function waits for input in its first input stream, regardless of what input appears in its second input stream. In order to define a reasonably fair version of the merge, one then resorts into a timed model. For such a model, there are some alternatives:

- One alternative is to model streams as timed signals over the non-negative reals, i.e., as functions $\mathbb{T} \to \mathbb{D}$ for some time domain $\mathbb{T}$ (e.g., the non-negative reals) and data domain $\mathbb{D}$. This has been done for example in [24]. This model seems fine for modeling physical processes that operate in continuous time, and possibly also hardware components at the physical level of electric signal, but not for modeling controllers or other components at the software level, which we would like to model as observing their input only at specific time points.

– Another alternative is the approach of MIMOS [26, 25] for modeling of software components such as controllers as well as complex software architecture. MIMOS allows input reading and output writing of software components only at specific time points, namely the start and end of their computation periods. This results in a model of streams as (finite or infinite) sequences of timed tokens. Each timed token is a pair $\langle d,t\rangle$, consisting of a data value $d \in \mathbb{D}$ and a time point $t \in \mathbb{T}$, sometimes referred to as a *time stamp*. Time stamping input and output is also the approach chosen in the framework of Lingua Franca [11].

  The non-negative time domain allows the definition of a CPO on timed streams, in a similar way as the original model by Kahn and McQueen. It seems reasonable to require that the time stamps in a stream are (not necessarily strictly) increasing. As we will see, this type of model still allows some important design decisions that may be important for being able to model a wide class of systems.

We take the second approach and base our model on the idea of viewing streams as sequences of timed tokens. We would like such a model to be able to capture as general a class of phenomena as possible, maybe including zero-delay-cycles, timeouts, and the like. We assume nodes to be fully time deterministic. This means that they follow a completely deterministic program/code. This program may use timers, and may also have access to a precise clock for wall-clock time. In each of its behaviors, a node reads inputs at precise time points. These time points may be different from behavior to behavior, but should be uniquely determined by the sequence of inputs received upto then.

In order to allow a general class of policies for how to receive inputs buffered in e.g. FIFO or unordered sets etc., we may want, when considering the input at a specified time t, to allow nodes to oversee the complete state of input ports, implying that they can see how many timed tokens have appeared up to time t including to detect absence of input, and also in which sequence they have been produced. For instance, a node may decide at time t to read all timed tokens available at time t, or all tokens up to a maximal number. In addition, we may likely want to include the possibility to read or send a sequence of tokens with the same time stamp but with a causality or priority order.

An example of a type of component that we may want to model, is a so-called *register*. One way to model a register is as a source of streams of timed tokens, where the time stamps are determined autonomously by the register. One could also envisage a model of registers, that are "polled", which can be modeled as having a stream of input tokens that are requests for register values, and which respond to each input by a register value after some delay (which can also be 0).

After these preliminary elaborations, let us try to define a general computation model, which can capture the semantics of a very general class of time deterministic nodes, including registers. In such a model, nodes have input and output ports. Each port can be an output port of at most one node, but can be an input port of several nodes. Each port sees a stream of timed tokens produced by the node which has this port as an output port. If the port is an input of the system, the stream is an arbitrary timed stream.

Let us now return to the model of streams of timed tokens. It turns out that out-of-the-box, it cannot model the full generality of time deterministic systems. Let us show this by the following example.

Example 1. Consider a node which outputs a token at time 2 if no input has appeared up to time 1, whereas it does not output anything at all if input appears at or before time 1. If we try to model this as a function f on timed streams, then this function would have to satisfy the following equations:

$$\begin{aligned} f(\varepsilon) &= \langle b, 2\rangle \\ f(\langle a, 1\rangle) &= \varepsilon \end{aligned}$$

To be used in the denotational semantics setting outlined above, the function f should be *continuous* (and therefore, in particular, *monotonic*), which means that prefixes of input streams should be mapped to prefixes of output streams. The function f defined here is obviously not monotonic. On the other hand, there is no question that the node is time deterministic in an intuitive sense. □

One way to understand this example is to acknowledge that streams of timed tokens by themselves do not show one important "input" to a node, which is time. We assume that nodes have access to the current (wall-clock) time, and this "input" can also influence the behavior of a node. In Example 1, the passage of time triggers a timeout, which in its turn causes output tokens to be emitted. From this understanding, we can extend the "streams of timed tokens" model by also letting the passage of time be an input to a network, and to each of its nodes. We can see (at least) two ways in which this can be done:

- We can let time be an additional "input stream": each network has an additional time input, which is read by all its nodes. In Example 1, the function f would output $\langle b, 2\rangle$ if its input is empty *and* its time output is more than 1. We can make f monotonic be requiring that any extension of the empty input can only add tokens with time stamps at least the corresponding time input, i.e., (strictly) larger than 1.
- We can let time be an additional "token" in each input stream. Such a token would be significant only to mark how far time has progressed for the receiver of the stream, implying that all time tokens except the last can be ignored. We thus represent this token by equipping each input stream with a "length" value Δ, indicating how far time has progressed in the reception of the stream. In Example 1, the function f would output $\langle b, 2\rangle$ for inputs $\langle \varepsilon, \Delta\rangle$ such that $\Delta > 1$. To recover monotonicity, $\langle \varepsilon, \Delta\rangle$ cannot be extended by timed tokens with time stamps smaller than Δ.

In §4, we work out this model of timed streams and illustrate some trade-offs using examples. The guiding principle is that the resulting model should retain the machinery of the original Kahn untimed model of streams of data items. The preservation of this principle imposes some constraints on the mathematical machinery, which we aim to illustrate.

3 Related Work

For modeling timed computation, in the literature, there are two primary approaches. The first one uses ordered sets, continuous (or monotonic) functions, and Tarski's fixed-point theorem. The second utilizes metric spaces, contraction mappings, and Banach's contraction principle.

The first category of models proposed in [23, 8, 9, 1, 26, 25] is essentially using complete partial orders and least fixed-points as models of timed systems. In [23], Yates and Gao tackle the fixed-point problem for systems with components constrained by bounded reaction times (referred to as "Δ-causal"). They transform this problem into one involving a suitably constructed Scott-continuous function, thus extending the Kahn principle to networks of real-time processes. In [8, 9], Liu et al. simplify the use of complete partial orders with a prefix order for timed systems. They introduce a special value to mark the absence of events and define signals on lower sets of the tag set to ensure time progression in the semantics of systems. Strictly causal functions, which extend signal domains monotonically, are shown to have unique fixed-points where time diverges if they are also Scott-continuous. This method relaxes the bounded-reaction-time constraint, allowing components with locally bounded reaction times. However, it does not fully address systems with more complex, varying reaction times, like those exhibiting non-trivial Zeno behaviors. The approach in [1] models components as Scott-continuous functions based on the prefix relation on signals. They note that in many practical scenarios, simplifying assumptions could be made: (1) the tag set is a total order, and (2) the values of a signal can be indexed in non-decreasing order. This allows signals to be viewed as streams, reducing the theory to the standard Kahn semantics. Moreover, heterogeneity (managing different tag sets) and distribution are addressed within the generalized Kahn semantics. In [26, 25], streams are modeled as sequences of timed tokens, where each token is described by a value and a time stamp. They represent software components as functions that read timed input streams and generate outputs at specific time points, subsequently presenting a fixed-point semantics for this model in the Kahn style.

The second category of models is proposed in other works, e.g. [5, 6, 7, 19, 20, 24, 16, 17, 3, 10, 13, 14, 18]. These models are based on metric spaces, contraction mappings, and Banach's contraction principle. Reed and Roscoe are the first to apply this framework in a real-time extension of CSP [19, 20]. Yates extends this approach to create the first extensional model of timed computation with a real-time extension of Kahn's process networks [24]. Müller and Scholz introduce another such extension in [16], adopting metric spaces of dense signals rather than timed streams. A uniform framework combining these models is presented in [5, 6, 7]. Building on the generalization of bounded reaction times, Portmann et al. [18] introduce a causality function to capture a more general form of strict causality. However, existing models [5, 6, 7, 19, 20, 24, 16, 18] require a positive lower bound on the reaction time of system components, which ensures that the functions are contraction mappings and allows the use of Banach's fixed-point theorem. This requirement prevents the modeling of non-trivial Zeno phenomena and rigid time divergence. Additionally, these approaches typically use an unbounded subset of real numbers as the tag set, excluding other options like super dense time, thus limiting their applicability to a broader range of systems.

Naundorf [17] removes the bounded-reaction-time constraint and allows for arbitrary tag sets, defining strictly causal functions with a non-constructive proof for unique fixed-points. His approach is valid under a totally ordered tag set but remains incomplete (see Example 3.9 in [14]). An interesting generalization of Naundorf's theorem is proven in [3], aiming to remove references to generalized distances, though the proof remains non-constructive. Naundorf's proposal is also rephrased in [10], using a generalized distance function to identify strictly causal functions as strictly contracting ones, facilitating access to the fixed-point theory of generalized ultra metric spaces. Matsikoudis and Lee [13, 14] further develop this concept by presenting a constructive fixed-point theorem for strictly contracting functions. They show that this theorem arises from a more intuitive concept of strict causality, where outputs are affected only by inputs that occur strictly before them, provided that the input ordering is well-founded.

4 Formal Definition of (Timed) Streams and Continuous Functions

Complete Partial Orders. We start with some of the basic definitions required in our setting. A *chain-Complete Partial Order* (CPO) is a pair $\langle A, \sqsubseteq \rangle$, where A is a set and $\sqsubseteq$ is a partial order on A such that any increasing chain $a_1 \sqsubseteq a_2 \sqsubseteq a_3 \sqsubseteq \cdots$ in A has a least upper bound, denoted $\lim_{n\to\infty} a_n$. A *continuous function* $f : A \to A$ on a CPO $\langle A, \sqsubseteq \rangle$ is a monotonic function such that $f(\lim_{n\to\infty} a_n) = \lim_{n\to\infty} f(a_n)$ for any increasing chain $a_1 \sqsubseteq a_2 \sqsubseteq \cdots$ in A. By Kleene's fixed-point theorem, every continuous function f on a CPO $\langle A, \sqsubseteq \rangle$ with a least element $\bot \in A$ has a least fixed-point, which can be constructed as $\lim_{n\to\infty} f^n(\bot)$. More generally, if $\langle A, \sqsubseteq_A \rangle$ and $\langle B, \sqsubseteq_B \rangle$ are CPOs, then we call a function $f : A \to B$ *continuous* if f is monotonic and $f(\lim_{n\to\infty} a_n) = \lim_{n\to\infty} f(a_n)$ for any increasing chain $a_1 \sqsubseteq_A a_2 \sqsubseteq_A \cdots$ in A (see, e.g., [15]).

In denotational semantics (e.g., [22]), the construction of a fixed-point can be seen as building the result of a computation through successive approximations. If f models a program or program fragment that uses input, part of which comes from previously computed output, successive approximations $f^n(\bot)$ can be interpreted as the output obtained after n applications of f.

Streams. Assume a data domain $\mathbb{D}$. Let $\langle \mathbb{D}^\infty, \sqsubseteq \rangle$ be the set of finite and infinite sequences of elements (i.e., "streams") in $\mathbb{D}$ with $\sqsubseteq$ being the prefix ordering on sequences. Then $\langle \mathbb{D}^\infty, \sqsubseteq \rangle$ is a CPO. When describing the behavior of networks, we associate streams with ports. For a set of ports P, this association is represented by a *valuation* $V : P \to \mathbb{D}^\infty$. The set of valuations $P \to \mathbb{D}^\infty$ is turned into a CPO, as follows: For $V : P \to \mathbb{D}^\infty$ and $V' : P \to \mathbb{D}^\infty$, we define $V \sqsubseteq V'$ if $V(p) \sqsubseteq V'(p)$ for all p in P. The limit $\lim_{n\to\infty} V_n$ of an increasing chain $V_1 \sqsubseteq V_2 \sqsubseteq \cdots$ in $P \to \mathbb{D}^\infty$ is the function $V_{lim} : P \to \mathbb{D}^\infty$ defined by $V_{lim}(p) = \lim_{n\to\infty} V_n(p)$ for all p in P.

Timed Streams. Let $\mathbb{T}$ be a totally ordered time domain with ordering relation $\leq$. For the time being, let $\mathbb{T}$ be the set $\mathbb{R}^{\geq 0}$ of real numbers (later, we might also let it be the set $\mathbb{N}^{\geq 0}$ of natural numbers). A *timed stream* is a finite or infinite stream of pairs in $\mathbb{D} \times \mathbb{T}$, in which the time stamps are non-decreasing. There are two possible choices for the non-decreasingness restriction: (i) A *strict timed stream* is a timed stream in which

the time stamps are strictly increasing. (ii) A *non-strict timed stream* is a timed stream in which the time stamps are non-decreasing. We consider for now non-strict timed streams, so that no restrictions are imposed that are stronger than necessary. Let TS be the set of streams in $(\mathbb{D}\times\mathbb{T})^\infty$ in which successive time stamps are non-decreasing. The set TS with prefix ordering is a CPO, in the same way as $\langle\mathbb{D}^\infty,\sqsubseteq\rangle$ is a CPO. For a set P of ports, the domain $\langle(P\to TS),\sqsubseteq\rangle$ is a CPO, which we denote by TS_P.

Functions over Timed Streams. Let us consider a node N with input ports I and output ports O. Let us try to represent its semantics as a function $f_N : TS_I \to TS_O$, We could then try to model, for instance, a (timed) merge node, which merges input streams on ports i_1 and i_2 to an output stream on port o, as a function $merge : TS_{\{i_1,i_2\}} \to TS_{\{o\}}$, defined through $merge(V_{\{i_1,i_2\}})(o) = f_{merge}(V(i_1),V(i_2))$, where f_{merge} satisfies the equations

$$\begin{aligned} f_{merge}(\varepsilon,\varepsilon) &= \varepsilon \\ f_{merge}(\langle a,t_1\rangle,\varepsilon) &= \langle a,t_1\rangle \\ f_{merge}(\varepsilon,\langle b,t_2\rangle) &= \langle b,t_2\rangle\ . \end{aligned}$$

However, if we want f_{merge} to be monotonic, we quickly run into a problem when defining $f_{merge}(\langle a,t_1\rangle,\langle b,t_2\rangle)$, since by monotonicity, the result of this merge must extend both $\langle a,t_1\rangle$ and $\langle b,t_2\rangle$. This is clearly impossible in general. A similar problem is revealed by Example 1. Looking closer at these examples, the problem seems to be that timed streams on their own do not show one important "input" to a node, which is passage of time.

We therefore have to assume that nodes have access to the current (wall-clock) time, and this "input" can also influence the behavior of a node. In a model with just timed streams, this input is invisible, even if it can trigger behavior of a node, either by making clear that it is "safe" to forward data from one of the inputs in the case of f_{merge} (since earlier input can no longer appear), or by triggering a timeout which induces subsequent output (as in Example 1). As a remedy to the problem, let us model the "passage of time" input simply as a time point t in $\mathbb{T}$. The idea of this "passage of time" input is to represent how far time has progressed in the execution of a node or network. By assuming the existence of a maximal element $\infty\in\mathbb{T}$, we obtain a CPO $\mathbb{T}$ with the ordering $\sqsubseteq$ being the standard relation $\leq$.

When extending the modeling framework with a "passage of time" input, we can see (at least) two ways in which this can be done.

1. We can let time be an additional "input stream": each network has an additional time input, which is read by all its nodes. In Example 1, the function f would output $\langle b,2\rangle$ if its input is empty *and* its time input is more than 1. We can make f monotonic be requiring that any extension of the empty input can only add tokens with time stamps at least the corresponding time input, i.e., (strictly) larger than 1.
2. We can let time be an additional "token" in each input stream. Such a token would mark how far time has progressed in that particular stream. We note that it is safe to ignore all time tokens except the last one. We thus represent this token by equipping each input stream with a "length" value, indicating how far time has progressed in the reception or output of the stream. In Example 1, the function f would output

$\langle b, 2\rangle$ for inputs $\langle \varepsilon, \Delta\rangle$ such that $\Delta > 1$. To recover monotonicity, an extended timed stream $\langle s, \Delta\rangle$ cannot be extended by timed tokens with time stamps smaller than Δ.

We work out alternative 1 in §4.1, and alternative 2 in §4.2. We use the timed merge as an illustrating function, since it exhibits several of the problems and features of the two solutions.

4.1 Timed Streams with Global Time

In this section, we work out in more detail alternative 1, in which each node and network have access to an additional time input, modeled simply as a value in $\mathbb{T}$, representing the "current time". We assume that nodes have access to the current (wall-clock) time, and this "input" can also influence the behavior of a node. As discussed above, we turn $\mathbb{T}$ into a CPO by adding ∞ as a maximal element. Let us preliminarily represent a node or network N with input ports I and output ports O as a function $f_N : TS_I \times \mathbb{T} \to TS_O$.

As a first illustration, we model the node in Example 1 as a function $f_{ex1} : TS_{\{i\}} \times \mathbb{T} \to TS_{\{o\}}$ through $f_{ex1}(V_{\{i\}}, t)(o) = f_t(V(i), t)$, where f_t is defined by

$$\begin{array}{ll} f_t(s,t) = \varepsilon & \text{if } t \leq 1 \\ f_t(s,t) = \langle b, 2\rangle & \text{if } \mathit{mint}(s) > 1 \text{ and } t > 1 \\ f_t(s,t) = \varepsilon & \text{if } \mathit{mint}(s) \leq 1 \text{ and } t > 1 \end{array}$$

Here and in the following, for a non-empty timed stream s, we define $\mathit{mint}(s)$ as the smallest time stamp in s, and define $\mathit{mint}(\varepsilon) = \infty$. In the following, we often abuse notation, and do not distinguish between the function representing the node (which maps valuations and time points to valuations, which is f_{ex1} in the example) and the same function defined directly on timed streams (f_t) in the example.

From this illustration, we make some observations.

- We cannot define $f_t(\varepsilon, 1)$ as $\langle a, 2\rangle$: continuity w.r.t. the time input dictates $f_t(\varepsilon, 1) = \varepsilon$. Thus, a timeout which depends on the absence of input can trigger subsequent output only when the time input has progressed past the timeout value. This is natural, since the absence of input at a time point t can only be determined after the time point t has passed.
- We cannot turn $TS_{\{i\}} \times \mathbb{T}$ into a CPO simply by component-wise extension of the orders on $TS_{\{i\}}$ and $\mathbb{T}$. Namely, if we would do so, then we would have:

$$(\varepsilon, 2) \sqsubseteq (\langle a, 1\rangle, 2) \qquad \text{but} \qquad f_t(\varepsilon, 2) \not\sqsubseteq f_t(\langle a, 1\rangle, 2) \quad .$$

  The problem here is that in allowing $(\varepsilon, 2) \sqsubseteq (\langle a, 1\rangle, 2)$ we allow timed tokens in an input stream to be extended with time tokens that appeared earlier than the "current time" (2 in this case). We must therefore be careful in the definition of our CPO, and only allow the input timed streams to be extended by tokens with time stamps at least being the time value. One could lament that this detracts slightly from the elegance of the original Kahn model, but it seems to be an unavoidable consequence of the fact that time is a global parameter.

– The time input can be thought of as the "current wall-clock time" of the node. If t is the current time input, then the output is the result of what the node commits to outputting at time t (which can include future outputs), given that the input streams up to t (i.e., ignoring larger time stamps) are as in the timed stream inputs.

After these observations, let us provide the formal definitions.

Definition 1 (Domain of Timed Streams with Global Time). *Let P be a set of ports. Let the domain TS_P^G consist of the set $TS_P \times \mathbb{T}$ with the ordering $\sqsubseteq$ defined by $(V,t) \sqsubseteq (V',t')$ if*

– *$t \leq t'$, and*
– *for each $p \in P$ there is a (possibly empty) timed stream s'' such that $V'(p) = V(p) \bullet s''$, and such that s'' contains no time stamp smaller than t.*

Using this definition, we model the behavior of a network with input ports I and output ports O by a continuous function from TS_I^G to TS_O. The least function from TS_I^G to TS_O is the function $\bot : TS_I^G \to TS_Q$ defined by $\bot(V_I,t)(o) = \varepsilon$ for each o in O.

For the node in Example 1, the function f_t becomes continuous.

We now use these definitions to model a timed merge node. The function $f^g_{merge} : TS^G_{\{i_1,i_2\}} \to TS_{\{o\}}$ (using the previously mentioned abuse of notation) can be defined by recursion :

$$
\begin{array}{lll}
f^g_{merge}(\langle a,t_1\rangle \bullet s_1 \ , \ \langle b,t_2\rangle \bullet s_2 \ , \ t) & = \langle a,t_1\rangle \bullet f^g_{merge}(s_1 \ , \ \langle b,t_2\rangle \bullet s_2 \ , \ t) & \text{if } t_1 \leq t_2 \\
f^g_{merge}(\langle a,t_1\rangle \bullet s_1 \ , \ \langle b,t_2\rangle \bullet s_2 \ , \ t) & = \langle b,t_2\rangle \bullet f^g_{merge}(\langle a,t_1\rangle \bullet s_1 \ , \ s_2 \ , \ t) & \text{if } t_2 < t_1 \\
f^g_{merge}(\langle a,t_1\rangle \bullet s_1 \ , \ \varepsilon \ , \ t) & = \langle a,t_1\rangle \bullet f^g_{merge}(s_1 \ , \ \varepsilon \ , \ t) & \text{if } t_1 \leq t \\
f^g_{merge}(\varepsilon \ , \ \langle b,t_2\rangle \bullet s_2 \ , \ t) & = \langle b,t_2\rangle \bullet f^g_{merge}(\varepsilon \ , \ s_2 \ , \ t) & \text{if } t_2 < t \\
f^g_{merge}(s_1 \ , \ s_2 \ , \ t) & = \varepsilon & \text{otherwise.}
\end{array}
$$

This makes the function f^g_{merge} continuous if allowing s_1 and s_2 above to be infinite timed streams. We observe that in order to make f^g_{merge} a function, one needs to define a priority when two inputs with the same time stamp appear. In the above definition, f^g_{merge} gives priority to the first input. The input i_2 can only be forwarded when time has progressed so that one can be sure that no more inputs arrive on i_1. As an illustration, $f^g_{merge}(\langle a,1\rangle \ , \langle b,1\rangle \ , \ 1) = \langle a,1\rangle$; the second output cannot be forwarded at time 1, since it is still possible for input to arrive on the first input port at time 1.

The unfairness of f^g_{merge} gives rise to an unfairness in the limit, namely if an infinite stream of tokens appear at one time instant: $f^g_{merge}((\langle a,1\rangle)^\omega, (\langle b,1\rangle)^\omega, 2) = (\langle a,1\rangle)^\omega$, and even $f^g_{merge}((\langle a,1\rangle)^\omega, \langle b,1\rangle, 2) = (\langle a,1\rangle)^\omega$. This effect presupposes that no port can carry more than an ω-infinite long stream of tokens[4]. We can avoid such unfairness at

[4] We could maybe experiment with a solution like $f_{merge}((\langle a,1\rangle)^\omega, \langle b,1\rangle, 2) = (\langle a,1\rangle)^\omega \langle b,1\rangle$, allowing streams that are longer than ω, but this would be beyond the scope of this paper.

each time point, by letting the merge switch priorities after each received token:

$$
\begin{array}{lll}
f^g_{merge}(\langle a,t_1\rangle \bullet s_1\ ,\ \langle b,t_2\rangle \bullet s_2\ ,\ t) = \langle a,t_1\rangle \bullet g^g_{merge}(s_1\ ,\ \langle b,t_2\rangle \bullet s_2\ ,\ t) & \text{if } t_1 \le t_2 \\
f^g_{merge}(\langle a,t_1\rangle \bullet s_1\ ,\ \langle b,t_2\rangle \bullet s_2\ ,\ t) = \langle b,t_2\rangle \bullet g^g_{merge}(\langle a,t_1\rangle \bullet s_1\ ,\ s_2\ ,\ t) & \text{if } t_2 < t_1 \\
f^g_{merge}(\langle a,t_1\rangle \bullet s_1\ ,\ \varepsilon\ ,\ t) = \langle a,t_1\rangle \bullet g^g_{merge}(s_1\ ,\ \varepsilon\ ,\ t) & \text{if } t_1 \le t \\
f^g_{merge}(\varepsilon\ ,\ \langle b,t_2\rangle \bullet s_2\ ,\ t) = \langle b,t_2\rangle \bullet g^g_{merge}(\varepsilon\ ,\ s_2\ ,\ t) & \text{if } t_2 < t \\
f^g_{merge}(s_1\ ,\ s_2\ ,\ t) = \varepsilon & \text{otherwise.} \\
g^g_{merge}(\langle a,t_1\rangle \bullet s_1\ ,\ \langle b,t_2\rangle \bullet s_2\ ,\ t) = \langle a,t_1\rangle \bullet f^g_{merge}(s_1\ ,\ \langle b,t_2\rangle \bullet s_2\ ,\ t) & \text{if } t_1 < t_2 \\
g^g_{merge}(\langle a,t_1\rangle \bullet s_1\ ,\ \langle b,t_2\rangle \bullet s_2\ ,\ t) = \langle b,t_2\rangle \bullet f^g_{merge}(\langle a,t_1\rangle \bullet s_1\ ,\ s_2\ ,\ t) & \text{if } t_2 \le t_1 \\
g^g_{merge}(\langle a,t_1\rangle \bullet s_1\ ,\ \varepsilon\ ,\ t) = \langle a,t_1\rangle \bullet f^g_{merge}(s_1\ ,\ \varepsilon\ ,\ t) & \text{if } t_1 < t \\
g^g_{merge}(\varepsilon\ ,\ \langle b,t_2\rangle \bullet s_2\ ,\ t) = \langle b,t_2\rangle \bullet f^g_{merge}(\varepsilon\ ,\ s_2\ ,\ t) & \text{if } t_2 \le t \\
g^g_{merge}(s_1\ ,\ s_2\ ,\ t) = \varepsilon & \text{otherwise.}
\end{array}
$$

Note here that the priority is defined for the cases that the time stamps of the input streams are equal.

Deriving the Model of Networks from Models of Nodes Let us now define the mathematical machinery for deriving the model of a network from the models of its nodes. Assume a network consisting of nodes $N_1,\ldots,N_k$. Each node N_i has input ports I_i and output ports O_i. Each port can be an output port of at most one node, i.e., $O_i \cap O_j = \emptyset$ for $i \neq j$. The composition of $N_1,\ldots,N_k$ is a network N with output ports $O = \cup_{i=1}^k O_i$ and input ports $I = (\cup_{i=1}^k I_i) \setminus O$. Each node N_i is modeled by a continuous function $f_i : TS^G_{I_i} \to TS_{O_i}$. From the disjointness of the sets O_i, we can decompose the functions f_i into one function $f_{o_j} : TS^G_{I_i} \to TS_{\{o_j\}}$ for each output port $o_j \in O$. For each output port $o_j \in O$ this produces an equation of form

$$V(o_j) = f_{o_j}(V(i_1),\ldots V(i_m),t) \tag{1}$$

From these equations, we would like to obtain a function $f_N : TS^G_I \to TS_O$, which represents the behavior of the entire network. The problem is that the equations (1) are recursive in that the output ports may appear on both sides of the equations. Moreover, for a given t, we cannot directly derive a least solution to (1) by fixed-point iteration, since the ordering $\sqsubseteq$ on $TS^G_{\{o_j\}}$ does not allow us to extend $V(o_j)$ by time stamps smaller than t, thereby preventing a step-by-step construction of $V(o_j)$. To illustrate this, assume that a port o_1 is both an input and output port of a node N_1 and is connected back to itself when forming the network. Suppose further that the node first outputs a token a at time 1, and thereafter (including at time 1) copies input to output with no delay. The resulting network then produces an infinite timed stream $(\langle a,1\rangle)^\omega$ on o_1. This stream should be constructed step-by-step by iteration, but for time 2 (say), in the equation $V(o_1) = f_{o_1}(V(o_1),2)$ we have $f_{o_1}(s,2) = \langle a,1\rangle \bullet s$. If we try to produce $(\langle a,1\rangle)^\omega$ by the usual fixed-point iteration, as the limit of the sequence $f_{o_1}(\varepsilon,2), f_{o_1}(\langle a,1\rangle,2), f_{o_1}(\langle a,1\rangle \bullet \langle a,1\rangle,2),\ldots$, then we run into the problem that $(\varepsilon,2) \not\sqsubseteq (\langle a,1\rangle,2)$ etc. To resolve this issue we need to perform the fixed-point iteration at time 1, thereby exploiting that $(\varepsilon,1) \sqsubseteq (\langle a,1\rangle,1) \sqsubseteq (\langle a,1\rangle \bullet \langle a,1\rangle,1) \sqsubseteq \cdots$.

This means that before solving the system (1) for time t, we need solve it for all time points $t' < t$, and use the supremum of these solutions as a starting approximation at time t. Let us collect this into a definition.

Definition 2. *Let a network be formed as above yielding equations (1). In order to derive the function defining network behavior, we reformulate (1) by replacing V by a time dependent valuation $V^* : O \times \mathbb{T} \to TS$ which satisfies*

$$V^*(o_j,t) = f_{o_j}(V^*(i_1,t),\ldots V^*(i_m,t),t) \tag{2}$$

for all time points $t \in \mathbb{T}$. Let then V^* be the valuation such that for each $t \in \mathbb{T}$, the valuation $\lambda o_j.V^*(o_j,t)$ is the smallest valuation (in the domain TS_O) such that for each $o_j \in O$ we have $V^*(o_j,t') \sqsubseteq V^*(o_j,t)$ for all $t' < t$. The function defining network behavior is then given by $f_N(V_I,t)(o_j) = V^*(o_j,t)$.

Returning to the example before this definition, we first construct the solution $V^*(o_1,1) = (\langle a,1\rangle)^\omega$. Since $(\langle a,1\rangle)^\omega$ is a maximal element, it follows that $V^*(o_1,t) = (\langle a,1\rangle)^\omega$ for all $t \geq 1$.

4.2 Timed Streams with Individual End Times

In this section, we define a semantic domain in which time has a more local character. We extend the domain of timed streams by adding, to each timed stream, an *end time*, which is a non-negative real number representing the time up to which the timed stream has been observed. The notion of stream end times leads to a somewhat richer model of node behavior than the global time model. Since end times are associated both with input and output streams, durations enable nodes to communicate not only data items with time stamps, but also at which point in time the next output can occur, and they give us more freedom in modeling networks. Maybe more importantly, streams with end times make it possible to preserve the decentralized nature of execution in Kahn networks: nodes communicate exclusively through channels, and as long as communication semantics is preserved the different parts of a network can execute completely independently. End times correspond to an implementation of timed systems in which nodes, when not producing any data, still output some form of empty data frames ("stuttering") to communicate the absence of output to subsequent nodes.

As before, we assume that the time domain $\mathbb{T}$ contains a maximal element $\infty \in \mathbb{T}$, and thus forms a CPO.

Definition 3. *The set TSE of* timed streams with end times *consists of pairs $\langle s,\Delta\rangle$ of a timed stream s and an end time Δ that is at least as large as all time stamps in s:*

$$TSE = \{\langle s,\Delta\rangle \in TS \times \mathbb{T} \mid \Delta \geq maxt(s)\}\ .$$

Here, we define $maxt(s)$ as the largest time stamp occurring in s, with $maxt(s) = \infty$ if there is no largest time stamp and $maxt(\varepsilon) = 0$. We can turn the set TSE into a CPO in a similar way as in the global time setting: timed streams with end times can be extended to longer streams by adding further elements, but only if the time stamps of the new elements are at least as large as the previous end time:

Definition 4. *For $\langle s,\Delta\rangle, \langle s',\Delta'\rangle \in TSE$, we define $\langle s,\Delta\rangle \sqsubseteq \langle s',\Delta'\rangle$ if and only if $\Delta \leq \Delta'$ and $s' = s \bullet s''$ for some stream s'' with $mint(s'') \geq \Delta$.*

We can observe that the limit of an increasing chain $\langle s_0, \Delta_0\rangle \sqsubseteq \langle s_1, \Delta_1\rangle \sqsubseteq \cdots$ is the timed stream with end time $\langle \lim_{n\to\infty} s_n, \lim_{n\to\infty} \Delta_n\rangle$. A node or network N with input ports I and output ports O is represented as a function $f_N : TSE_I \to TSE_O$, where $TSE_P = P \to TSE$ is the CPO that associates a timed stream with end time with each element of P, with point-wise extension of the ordering relation.

Modeling timeouts. For illustration, let us return to Example 1. We can model the timeout node using the following function f_t. As mentioned in the beginning of the section, our model of the timeout node is able to explicitly state at which points in time the node can produce data. Since the earliest output can occur at time point 2, for input stream duration $\Delta \leq 1$ the function can be defined as $\langle \varepsilon, 2\rangle$. This output can then be extended to either $\langle\langle b, 2\rangle, \infty\rangle$ or $\langle \varepsilon, \infty\rangle$, depending on whether input data is observed at some time $t \leq 1$ or not:

$$\begin{array}{ll} f_t(\langle s, \Delta\rangle) = \langle \varepsilon, 2\rangle & \text{if } \Delta \leq 1 \\ f_t(\langle s, \Delta\rangle) = \langle\langle b, 2\rangle, \infty\rangle & \text{if } \Delta > 1 \text{ and } \mathit{mint}(s) > 1 \\ f_t(\langle s, \Delta\rangle) = \langle \varepsilon, \infty\rangle & \text{if } \Delta > 1 \text{ and } \mathit{mint}(s) \leq 1 \end{array} \tag{3}$$

We can verify that this function is indeed a continuous function in our CPO.

Deriving Models of Networks from Models of Nodes. Like in §4.1, we discuss how the functions associated with network nodes give rise to behavior of a network as a whole. For this, assume again a network consisting of nodes $N_1, \ldots, N_k$, where each node N_i has input ports I_i and output ports O_i. Each port can be an output port of at most one node, i.e., $O_i \cap O_j = \emptyset$ for $i \neq j$. The composition of $N_1, \ldots, N_k$ is a network N with output ports $O = \cup_{i=1}^{k} O_i$ and input ports $I = \cup_{i=1}^{k} I_i \setminus O$. Each node N_i is modeled by a continuous function $f_i : TSE_{I_i} \to TSE_{O_i}$.

To execute the network, each of the nodes has to process its inputs; since the output of some of the nodes is used as input of other nodes, and since there might be feedback loops, execution has to be repeated until a fixed-point is reached. For this, consider the CPO TSE_O that associates a timed stream with end time with each output port. Each element of TSE_O can be considered as a possible internal state of the network. To run the network to completion, we start from the state $\bot \in TSE_O$ that assigns the smallest element (empty stream with end time 0) to each output port, and then repeatedly update all output streams to the values computed by the nodes.

More formally, for some set P of ports, some element $a \in TSE_P$, and some subset $P' \subseteq P$, we write $a|_{P'} \in TSE_{P'}$ for the restriction of a to P'. We define the update function of the network as the function $F_{step}(in) : TSE_O \to TSE_O$ from internal states to internal states, taking the (fixed) inputs $in \in TSE_I$ of the network into account:

$$F_{step}(in)(out) \;=\; \bigcup_{i\in\{1,\ldots,k\}} f_i\big((in \cup out)|_{I_i}\big)$$

The function $F_{step}(in)$ is continuous, and its least fixed-point $F(in) = \lim_{n\to\infty} F_{step}(in)^n(\bot)$ corresponds to the outputs of the networks when run to completion.

From Denotation to Implementation. Since functions in our new model not only *receive,* but also *return* end times of streams, a discussion is necessary whether all continuous functions model nodes that could exist in the real world. Consider the following three functions for copying data from input to output:

$$f_{1/2}(\langle\langle a_1,t_1\rangle,\langle a_2,t_2\rangle,\langle a_3,t_3\rangle,\ldots,\Delta\rangle) = \langle\langle a_1,\tfrac{1}{2}t_1\rangle,\langle a_2,\tfrac{1}{2}t_2\rangle,\langle a_3,\tfrac{1}{2}t_3\rangle,\ldots,\tfrac{1}{2}\Delta\rangle$$
$$f_1(\langle\langle a_1,t_1\rangle,\langle a_2,t_2\rangle,\langle a_3,t_3\rangle,\ldots,\Delta\rangle) = \langle\langle a_1,t_1\rangle,\langle a_2,t_2\rangle,\langle a_3,t_3\rangle,\ldots,\Delta\rangle$$
$$f_2(\langle\langle a_1,t_1\rangle,\langle a_2,t_2\rangle,\langle a_3,t_3\rangle,\ldots,\Delta\rangle) = \langle\langle a_1,2t_1\rangle,\langle a_2,2t_2\rangle,\langle a_3,2t_3\rangle,\ldots,2\Delta\rangle$$

All three functions are continuous, but they differ in the time points and end time of the generated output streams. Function $f_{1/2}$ outputs every item received at time point t at the time $\frac{1}{2}t$, i.e., possibly at an earlier time than t; function f_1 keeps all time stamps, while function f_2 outputs at time $2t$, possibly at a later time than t. Although all three functions are meaningful in our semantic framework, function $f_{1/2}$ is an impossibility from an operational point of view: a node implementing this function would output (or forward) data before it has received the data. Assuming that time progresses equally fast on all streams of the network, no such node can exist. Functions f_1 and f_2, in contrast, could be implemented; a node for f_2 would require unbounded memory, however, since it has to store data incoming at time t until it can be output at time $2t$.

In order to talk about implementability, we introduce a general notion of causality. We first introduce the notion of cause, which is the minimal input producing a given output. Then, given a cause $\langle s',\Delta'\rangle$, we ask for a minimal extension $\langle s'',\Delta''\rangle$ of $\langle s',\Delta'\rangle$ such that $f(\langle s',\Delta'\rangle \bullet \langle s'',\Delta''\rangle)$ is an extension of $f(\langle s',\Delta'\rangle)$. For readability, the following definition only consider the case of unary functions, the extension to input tuples is straightforward.

Definition 5 (Notion of cause). *Consider a continuous function $f_N : TSE \to TSE$. We say that*

- $\langle s',\Delta'\rangle$ *is the* cause *for* $\langle s,\Delta\rangle$ *for* f_N *if* $\langle s',\Delta'\rangle$ *is the smallest stream such that* $f_N(\langle s',\Delta'\rangle) = \langle s,\Delta\rangle$. *That is, if for all* $\langle s'',\Delta''\rangle \sqsubset \langle s',\Delta'\rangle$, $f_N(\langle s'',\Delta''\rangle) \sqsubset \langle s,\Delta\rangle$ *(where* $\sqsubset$ *stands naturally for strictly smaller, that is, strictly smaller stream or end time).*
- *if* $\langle s',\Delta'\rangle$ *is the* cause *for* $\langle s,\Delta\rangle$ *for* f_N*, and* $\langle s,\Delta\rangle$ *is of the form* $\langle s^* \bullet x,\Delta\rangle$ *for some stream x (not necessarily a single item), then* $\langle s',\Delta'\rangle$ causes the extension $\langle s^*,\Delta^*\rangle \to \langle s^* \bullet x,\Delta\rangle$ *if for all* $\langle s'',\Delta''\rangle \sqsubset \langle s',\Delta'\rangle$, $f_N(\langle s'',\Delta''\rangle) \sqsubseteq \langle s^*,\Delta^*\rangle$. *Note that the extension x is minimal.*
- *if* $\langle s',\Delta'\rangle$ causes the extension $\langle s^*,\Delta^*\rangle \to \langle s^* \bullet x,\Delta\rangle$*, then the* cause for the extension $\langle s^*,\Delta^*\rangle \to \langle s^* \bullet x,\Delta\rangle$ *is* Δ' *if for any* $\Delta'' < \Delta'$, $f_N(\langle s',\Delta''\rangle) \sqsubseteq \langle s^*,\Delta^*\rangle$*, and there exists such a* Δ''. *That is, it is the time progress to* Δ' *(e.g. a timeout) that causes the extension* $\langle s^*,\Delta^*\rangle \to \langle s^* \bullet x,\Delta\rangle$.

We have now defined the cause for an extension of the output $\langle s^*,\Delta^*\rangle$ to a larger output $\langle s^* \bullet x,\Delta\rangle$. Now, we define define causality as a constraint on the time stamps of the output extension x, such that we can show (conjecture) that causal functions are implementable, possibly by infinitely powerful machines.

Definition 6 (Causality). *Consider a continuous function $f_N : TSE \to TSE$. Then, every $\langle s, \Delta'' \rangle$ in the image of f_N with $\langle \varepsilon, 0 \rangle \sqsubset \langle s, \Delta'' \rangle$ can be written in the form $\langle s^* \bullet x, \Delta'' \rangle$, such that there is a $\langle s', \Delta' \rangle$ which* causes the extension $\langle s^*, \Delta^* \rangle \to \langle s^* \bullet x, \Delta \rangle$ *for some $\Delta \leq \Delta''$. The function f_N is* causal *if*

- *whenever $\langle s', \Delta' \rangle$ is the cause for the extension $\langle s^*, \Delta^* \rangle \to \langle s^* \bullet x, \Delta \rangle$ and $\varepsilon \sqsubset x$, then, if $s' = \varepsilon$ then $mint(x) \geq 0$ else $mint(x) \geq maxt(s')$.*
- *if, in addition, Δ' is the cause for the extension $\langle s^*, \Delta^* \rangle \to \langle s^* \bullet x, \Delta \rangle$ (and $\varepsilon \sqsubset x$), then $mint(x) \geq \Delta'$. The time stamp of x must be at least as large as its cause, and here the cause is Δ'.*

This definition extends to tuples in a straightforward manner: in both cases the time of the effect must be greater than the maximal time of its cause.

Now, we can formulate our conjecture on the requirements for a function to be implementable.

Conjecture 1 (Timing consistency postulate). A continuous function $f_N : TSE_I \to TSE_O$ is implementable, if

(1) it is causal in the sense of Definition 6, and
(2) for every chain of input tuples with diverging end times, that is,

$$(\langle s_1^1, \Delta_1^1 \rangle, \ldots, \langle s_n^1, \Delta_n^1 \rangle) \sqsubseteq (\langle s_1^2, \Delta_1^2 \rangle, \ldots, \langle s_n^2, \Delta_n^2 \rangle) \sqsubseteq \cdots \quad \text{with} \quad \forall k.\ \lim_{j \to \infty} \Delta_k^j = \infty$$

also the end times of each output stream tend to infinity:

$$\text{if } f_N(\langle s_1^j, \Delta_1^j \rangle, \ldots, \langle s_n^j, \Delta_n^j \rangle) = (\langle r_1^j, \bar{\Delta}_1^j \rangle, \ldots, \langle r_m^j, \bar{\Delta}_m^j \rangle) \quad \text{then} \quad \forall k.\ \lim_{j \to \infty} \bar{\Delta}_k^j = \infty$$

(the function does not block time).

Monotonicity of f_N guarantees that the output produced for an extension of a previous input can only extend the previously computed output; it may depend on the previous output (its potentially unbounded memory). Causality (condition 1) implies timing consistency. If the interpretation of time stamps is the time at which its data item is written (or delivered), then it is guaranteed that every cause is written not later than its corresponding effect. We may still need an infinitely fast machine. Finally, condition (2) guarantees that for every input chain that does not block time (the end times of all input streams diverge), output approximations with diverging end times are produced. That is f_n does not block time progress (not for more than a finite number of steps).

Modeling timed merge. As a second example of using our model of timed streams with end times, we show how timed merge can be captured. The following definition corresponds to the "unfair" version of merge from §4.1, i.e., the function prefers data arriving at input 1 over the one from input 2. We use an auxiliary function g to model the interleaving of data items from the two input streams. In the first equation, defining the actual function f^{et}_{merge}, we need to set the end time of the output stream to the minimum

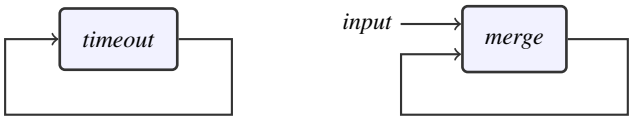

Fig. 1: Feedback loops with timeout and timed merge nodes

end time $\min\{\Delta_1,\Delta_2\}$ of the input streams, since additional data arriving through the input streams can lead to further output at time $\min\{\Delta_1,\Delta_2\}$ or later.

$$
\begin{array}{lll}
f^{et}_{merge}(\langle s_1,\Delta_1\rangle,\langle s_2,\Delta_2\rangle) & = \langle g(\langle s_1,\Delta_1\rangle,\langle s_2,\Delta_2\rangle),\min\{\Delta_1,\Delta_2\}\rangle & \\
g(\langle\langle a,t_1\rangle \bullet s_1,\Delta_1\rangle,\langle s_2,\Delta_2\rangle) & = \langle a,t_1\rangle \bullet g(\langle s_1,\Delta_1\rangle,\langle s_2,\Delta_2\rangle) & \text{if } t_1 \le \min(mint(s_2),\Delta_2) \\
g(\langle s_1,\Delta_1\rangle,\langle\langle b,t_2\rangle \bullet s_2,\Delta_2\rangle) & = \langle b,t_2\rangle \bullet g(\langle s_1,\Delta_1\rangle,\langle s_2,\Delta_2\rangle) & \text{if } t_2 < \min(mint(s_1),\Delta_1) \\
g(\langle\langle a,t_1\rangle \bullet s_1,\Delta_1\rangle,\langle\varepsilon,\Delta_2\rangle) & = \varepsilon & \text{if } \Delta_2 < t_1 \text{ lacks input from inp1} \\
g(\langle\varepsilon,\Delta_1\rangle,\langle\langle b,t_2\rangle \bullet s_2,\Delta_2\rangle) & = \varepsilon & \text{if } \Delta_1 \le t_2 \text{ lacks input from inp2} \\
g(\langle\varepsilon,\Delta_1\rangle,\langle\varepsilon,\Delta_2\rangle) & = \varepsilon &
\end{array}
$$

4.3 On Networks with Feedback Loops

As a litmus test for our semantic models, we consider the handling of nodes that do not consume any computation time (zero delay), and of networks that contain zero-delay feedback loops. A zero-delay loop is a feedback loop that can produce output without any time passing, i.e., output is produced at the same time point as input entering the loop. Although it can be argued that actual implementations of networks cannot contain zero-delay steps, such computations are an important abstraction when modeling systems; it is therefore desirable that denotational semantics are able to capture computation that is instantaneous.

Simple Feedback Networks

Timeout with feedback. We consider networks with feedback loops, shown in Fig. 1. The first example is a timeout node, which is a node that outputs data with delay: if no data has been received before or at time 1, then a data item is output at time 2. As a thought experiment, we investigate the effect of applying this timeout node in a direct feedback loop, shown on the left-hand side of Fig. 1.

In the global-time model, the network defines a continuous function from $\mathbb{T}$ to $TS_{\{o\}}$. We can compute the output as a fixed-point for each time point t in $\mathbb{T}$ from the definition in the beginning of §4.1. As a result, the port contains ε if $t \le 1$ and $\langle b,2\rangle$ if $t > 1$.

In the model of timed streams with end time, using definition (3), the output produced by the left network is the limit of the sequence

$$\langle\varepsilon,0\rangle \sqsubseteq \langle\varepsilon,2\rangle \sqsubseteq \langle\langle b,2\rangle,\infty\rangle$$

which coincides with the last timed stream $\langle\langle b,2\rangle,\infty\rangle$. Arguably, this corresponds to our intuition about timeout node run in a feedback loop. It should be noted that the

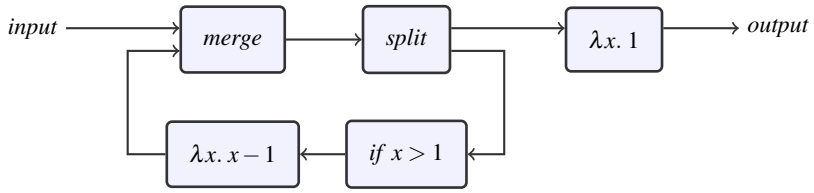

Fig. 2: Repetition network with zero-delay loop: for each timed data item $\langle k,t\rangle$ that is received, the network is supposed to output k times the item $\langle 1,t\rangle$.

"passing" of time in this example is the result of the timeout node specifying an output stream end time that is greater than the end time of the input stream; intuitively, the node itself is making time pass.

Merge with feedback. An example of zero-delay computation is the timed merge function, shown on the right-hand side of Fig. 1 and discussed for the global-time model in §4.1 and for the model of timed streams with end times in §4.2. Both of our models of timed merge (f^{g}_{merge} and f^{et}_{merge}) forward data received on the input streams without delay to the output stream. Both models moreover work in case of an infinite number of data items arriving at the same point in time. This situation can lead to an unfair selection of data, as discussed in §4.1, where also a possible fix to ensure fairness is provided.

In the global-time model, the first external input is copied into an infinite stream on the internal channel, but only after time has advanced from the time of that input, because of the priority given to the external input. This priority also lets a stream of external input that appears with the same time stamp on the first channel be repeated infinitely on the output channel. Denoting the whole network by $f_{mergefb}$, we have, for instance

$$\begin{array}{lll}
f_{mergefb}(\langle a,t_1\rangle \bullet s_1\ ,\ t) & = \varepsilon & \text{if } t_1 > t \\
f_{mergefb}(\langle a,t_1\rangle \bullet s_1\ ,\ t) & = \langle a,t_1\rangle & \text{if } t_1 = t < mint(s_1) \\
f_{mergefb}(\langle a,t_1\rangle \bullet s_1\ ,\ t) & = (\langle a,t_1\rangle)^{\omega} & \text{if } t_1 < \min(t, mint(s_1)) \\
f_{mergefb}(\langle a,t_1\rangle\langle b,t_1\rangle \bullet s_1\ ,\ t) & = (\langle a,t_1\rangle\langle b,t_1\rangle)^{\omega} & \text{if } t_1 < \min(t, mint(s_1))
\end{array}$$

In the model of timed streams with end times, the zero-delay feedback loop shown in Fig. 1 has a less obvious effect. Given input streams with end times Δ_1 and Δ_2, respectively, the function f^{et}_{merge} produces a stream with end time $\min\{\Delta_1,\Delta_2\}$ as output. This implies that the fixed-point of the feedback loop in Fig. 1 has end time 0, i.e., the feedback loop makes time stop at point 0. The node can still produce output at time point 0, provided that input occurs at time 0 on the first input, but the end time of the output stream never advances beyond 0, hence the same holds for the second output.

Repetition Network As a somewhat more complicated example, consider now the network with zero-delay feedback of Fig. 2. For each $\langle k,t\rangle$ received from the external input, $k-1$ items $\langle i,t\rangle$ are fed back to the second input of the merge node and for each element $\langle i,t\rangle$ received by the merge node an item $\langle 1,t\rangle$ is sent to the global output. The

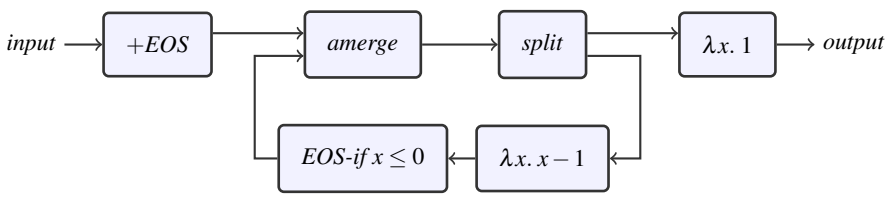

Fig. 3: Improved repetition network with zero-delay loop, using timed merge *amerge* with flow arbitration.

other nodes used in the network have semantics as follows: the nodes containing λ-expressions apply a function to each element of a stream ("*map*"); the *if*-nodes remove all data elements from a stream that do not satisfy the given predicate ("*filter*"); and the *split*-node copies all data that it receives to multiple output streams. Functions modeling those nodes can be defined easily in our framework.

Sadly, the network shown in Fig. 2 does not work, showcasing the limitations of timed merge. There are two issues:

1. The unfair timed merge has to prioritize one of its two input streams, which prevents the input data and the data flowing back through the feedback loop from being merged correctly. If priority is given to the input to the network, the feedback data is blocked (since further input could arrive at the same time); if priority is given to the feedback data, the input data is blocked. In particular, in the global time model, if priority is given to the input to the network, all elements $\langle k, t_1 \rangle$ present can be read at time t_1. But the feedback input is blocked at time t_1. If the external input progresses beyond t_1, one cannot extend the feedback input with elements at t_1. Giving priority to the feedback input has a similar effect.
2. When using the model of timed streams with end times, the feedback loop gets stuck at time 0; time never progresses beyond time 0.

In Fig. 3, we show one possible way to fix the repetition network. As we want to suppose asynchronous independent execution of the nodes, we need an explicit termination signal, at least from the second input, after which time may progress and the next item from the external input can be consumed. For this, the network in Fig. 3 uses a timed merge node *amerge* that requires an explicit signal to switch between the two input streams. For this, we assume a special data item *EOS* ("end-of-stride") that can occur on the streams. The intended semantics of the node *amerge* is as follows:

- Initially, *amerge* forwards data items from the first input to its output stream. Data items arriving on the second input are not forwarded and remain in the input channel.
- Once an *EOS* is received on the first input stream, the *amerge* switches to the second input, and now copies data from the second input to the output stream, including all items that were previously queued. No data is read from the first input channel.
- Once an *EOS* is received on the second input stream, the nodes switches back to the first input, etc.

To define a complete network, we introduce two further nodes: the node $+EOS$ copies data from the input to the output, but adds an *EOS* after each item; the node *EOS-if* replaces every data item for which the given predicate holds with *EOS*. The definition of the three new nodes in the global-time model and the model of timed streams with end time is straightforward.

5 Conclusion and Discussion

In this paper, we have studied the problem of defining deterministic execution semantics of asynchronous data-flow languages in the presence of time. By designing appropriate complete partial orders of timed streams, we are able to define the semantics in denotational style, staying close to the original definitions for Kahn process networks, yet are able to handle zero-delay feedback loops. The research has been inspired by the (ongoing) work on the MIMOS model [26] of computation for embedded systems.

There are several avenues of future work. We can extend our notion of *timed stream with end time* by distinguishing a *weak* and a *strong* notion of end time. This extension might be useful to model zero-delay loops (see §4.3) in a more direct fashion. With the *weak* interpretation of end time Δ, any extension of the timed stream can add items with time stamps $\geq \Delta$ (as we have considered so far in Definition 4), whereas with a *strong* interpretation of end time Δ, any extension of the timed stream can only add items with time stamps $> \Delta$. This allows us to write more natural definitions, for example of the merge function: we never need to suppose an end time beyond the time point of interest, but only need to distinguish between streams with weak and strong end points.

Another extension is an alternative notion of external time: it is defined by a sequence of time points at which a function may look at its input and extend the previously computed output. This variant can be expressed within the framework of external time defined here by requiring that every output produced has a time stamp not smaller than the time point at which it is read, meaning that our versions of merge cannot forward their initial time stamps in all cases. These external computation time points may be defined per node (then it comes very close to a model for MIMOS) or for an entire network.

Bibliography

[1] Caspi, P., Benveniste, A., Lublinerman, R., Tripakis, S.: Actors without directors: A Kahnian view of heterogeneous systems. In: Majumdar, R., Tabuada, P. (eds.) Hybrid Systems: Computation and Control. pp. 46–60. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)

[2] Guevara, I., Ryan, S., Singh, A., Brandon, C., Margaria, T.: Edge IoT prototyping using model-driven representations: A use case for smart agriculture. Sensors 24(2), 495 (2024), https://doi.org/10.3390/s24020495

[3] Hesselink, W.H.: A generalization of Naundorf's fixpoint theorem. Theor. Comput. Sci. 247(1-2), 291–296 (2000), https://doi.org/10.1016/S0304-3975(00)00202-4

[4] Kahn, G.: The semantics of a simple language for parallel programming. In: IFIP 74. pp. 471–475. North-Holland (1974)

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

[6] Lee, E.A.: Modeling concurrent real-time processes using discrete events. Annals of Software Engineering 7(1), 25–45 (1999)

[7] Liu, J., Lee, E.A.: On the causality of mixed-signal and hybrid models. In: Maler, O., Pnueli, A. (eds.) Hybrid Systems: Computation and Control. pp. 328–342. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)

[8] Liu, X.: Semantic foundation of the tagged signal model. University of California, Berkeley (2005)

[9] Liu, X., Lee, E.A.: CPO semantics of timed interactive actor networks. Theor. Comput. Sci. 409(1), 110–125 (2008), https://doi.org/10.1016/j.tcs.2008.08.044

[10] Liu, X., Matsikoudis, E., Lee, E.A.: Modeling timed concurrent systems. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006 – Concurrency Theory. pp. 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)

[11] Lohstroh, M., Menard, C., Soroush, B., Lee, E.A.: Toward a lingua franca for deterministic concurrent systems. ACM Transactions on Embedded Computing Systems 20(4) (2021)

[12] Margaria, T., Steffen, B.: Service-orientation: Conquering complexity with XMDD. In: Hinchey, M., Coyle, L. (eds.) Conquering Complexity, pp. 217–236. Springer (2012), https://doi.org/10.1007/978-1-4471-2297-5_10

[13] Matsikoudis, E., Lee, E.A.: On fixed points of strictly causal functions. In: Braberman, V.A., Fribourg, L. (eds.) Formal Modeling and Analysis of Timed Systems - 11th International Conference, FORMATS 2013, Buenos Aires, Argentina, August 29-31, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8053, pp. 183–197. Springer (2013), https://doi.org/10.1007/978-3-642-40229-6_13

[14] Matsikoudis, E., Lee, E.A.: The fixed-point theory of strictly causal functions. Theor. Comput. Sci. 574, 39–77 (2015), https://doi.org/10.1016/j.tcs.2015.01.036

[15] Matt, C., Maurer, U., Portmann, C., Renner, R., Tackmann, B.: Toward an algebraic theory of systems. Theoretical Computer Science 747, 1–25 (2018), https://www.sciencedirect.com/science/article/pii/S0304397518304092
[16] Müller, O., Scholz, P.: Functional specification of real-time and hybrid systems. In: Maler, O. (ed.) Hybrid and Real-Time Systems. pp. 273–285. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
[17] Naundorf, H.: Strictly causal functions have a unique fixed point. Theor. Comput. Sci. 238(1-2), 483–488 (2000), https://doi.org/10.1016/S0304-3975(99)00165-6
[18] Portmann, C., Matt, C., Maurer, U., Renner, R., Tackmann, B.: Causal boxes: Quantum information-processing systems closed under composition. IEEE Transactions on Information Theory 63(5), 3277–3305 (2017)
[19] Reed, G.M., Roscoe, A.W.: A timed model for communicating sequential processes. In: Kott, L. (ed.) Automata, Languages and Programming. pp. 314–323. Springer Berlin Heidelberg, Berlin, Heidelberg (1986)
[20] Reed, G.M., Roscoe, A.W.: Metric spaces as models for real-time concurrency. In: Main, M., Melton, A., Mislove, M., Schmidt, D. (eds.) Mathematical Foundations of Programming Language Semantics. pp. 331–343. Springer Berlin Heidelberg, Berlin, Heidelberg (1988)
[21] Steffen, B., Margaria, T., Nagel, R., Jörges, S., Kubczak, C.: Model-driven development with the jABC. In: Bin, E., Ziv, A., Ur, S. (eds.) Hardware and Software, Verification and Testing, Second International Haifa Verification Conference, HVC 2006, Haifa, Israel, October 23-26, 2006. Revised Selected Papers. Lecture Notes in Computer Science, vol. 4383, pp. 92–108. Springer (2006), https://doi.org/10.1007/978-3-540-70889-6_7
[22] Tennent, R.: The denotational semantics of programming languages. Comm. of the ACM 19, 437–453 (1976)
[23] Yates, R., Gao, G.: A Kahn principle for networks of non-monotonic real-time processes. In: Proc. PARLE. Lecture Notes in Computer Science, vol. 694 (1993)
[24] Yates, R.K.: Networks of real-time processes. In: Best, E. (ed.) CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings. Lecture Notes in Computer Science, vol. 715, pp. 384–397. Springer (1993), https://doi.org/10.1007/3-540-57208-2_27
[25] Yi, W., et al: MIMOS in a nutshell. (in preparation) (2024)
[26] Yi, W., Mohaqeqi, M., Graf, S.: MIMOS: A deterministic model for the design and update of real-time systems. In: ter Beek, M.H., Sirjani, M. (eds.) Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13271, pp. 17–34. Springer (2022), https://doi.org/10.1007/978-3-031-08143-9_2