MIMOS* : A Deterministic Model for the Design and Update of Real-Time Systems

Wang Yi†, Morteza Mohaqeqi† and Susane Graf‡
†*Uppsala University, Sweden*
Email: {wang.yi,morteza.mohaqeqi}@it.uu.se
‡*Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, Grenoble, France*
Email: susanne.graf@imag.fr

***Abstract*— Inspired by the pioneering work of Gilles Kahn on concurrent systems, we propose to model timed systems as a network of software components (implemented as real-time processes or tasks), each of which is specified to compute a collection of functions according to given timing constraints. We present a fixed-point semantics for this model which shows that each system function of such a network computes for a given set of (timed) input streams, a deterministic (timed) output stream. As a desired feature, such a network model can be modified by integrating new components for adding new system functions without changing the existing ones. Additionally, existing components may be replaced also by new ones fulfilling given requirements. Thanks to the deterministic semantics, a model-based approach is enabled for not only building systems but also updating them after deployment, allowing for efficient analysis techniques such as model-in-the-loop simulation to verify the complete behaviour of the updated system.**

1. Motivation

Today, a large part of the functionality of technical systems such as cars, airplanes, and medical devices is implemented by software, as an (embedded) real-time system. The current trend is that traditionally mostly closed and single-purpose systems become open platforms. They aim at the integration of an expanding number of software components over their life-time, e.g., in order to customize and enhance their functionality according to varying needs of individual users. To enable this, we must design and build systems that allow for updates after deployment. More importantly, it must be verified in field that the resulting systems not only preserve the original as well as the extended functionality, but also stay safe after updates. Clearly, such analysis and verification processes must be carried out in a model-based approach. This demands a deterministic model for real-time systems, that supports for automatic synthesis of software components, and also ensures that all properties verified based on a system model (modified) are also true of the system (updated accordingly).

. * MIMOS stands for Multi-Input Multi-Output Real-Time Systems.

Over the years, there have been a large number of timed models developed in the literature, notably the theory of timed automata, as well as various task models [26] developed for real-time systems. These models are either extremely expressive and highly non-deterministic, but cannot be analyzed efficiently, or restrictive in terms of expressive power, and cannot be used to design systems (e.g., system with complex synchronization structures) with desired functional behaviours.

In this paper, we propose to model timed systems as a network of real-time software components connected by communication channels in the style of Kahn Process Networks (KPN) [16], allowing asynchronous data exchange. We present a simple but expressive description language, called MIMOS, to formalize such abstract network models. We want to reduce the analysis complexity of functional and timing behaviours of our model-based approach to system design and update. Therefore, we have adopted the following principles for designing the model and its semantics:

Determinism. The key in a model-based approach is that the model of a system should be deterministic to ensure that the system behaves the same way as its model. In our approach, a model, as well as a system derived from it, can be viewed as a stream transformer. For a given set of input streams, the output streams defined by the model (system) must be unique. This means that, it specifies a set of functions over streams (we call them *system functions*) such that each one defines an output stream from a set of input streams. Second, the model (system) should be timing deterministic. This means that at any time point, if the inputs required for computation are available, the corresponding output should be delivered at a future time point after a fixed delay. Timing determinism can be relaxed to ensure only that the output may be delivered within a given time bound.

Separation of computation and communication. A system model should allow to specify the components of the system for computations and the communication channels for data exchange separately, as not only independent units in the architecture but also in the semantics. We assume non-blocking data exchange, implemented by either asynchronous FIFO channels for buffering system inputs and out-

puts, or registers for storing sampled time-dependent data. The separation allows the system components to be specified as *independent real-time tasks*, whose timing behaviours can be analyzed efficiently and locally. More importantly, as it is well-recognized in the theory of real-time scheduling [24], the underlying schedulability analysis for deployment will be greatly simplified compared with the case for dependent real-time tasks.

Updatability (avoidance of interference). The model of a system should allow for modifications by integrating new components for new system functions or replacing the existing components with refined ones, without changing the existing system functions determined by the original model. The separation of computation and communication by asynchronous data exchange avoids inter-component interference when new components are integrated. We require that new components may read but never write to the existing components via FIFOs or registers unless writing operations by the new components fulfill given requirements (specified using e.g. contracts [13]), which is essential for future updates to preserve the original system functionality. Even though protocols may be needed to coordinate data exchange among the components (e.g. to avoid race conditions in register reading and writing), the components may operate autonomously or independently from each other even when some of them stopped functioning correctly.

The rest of the paper is organized as follows: Section 2 summarizes our contributions and related work. Section 3 presents the MIMOS model, its informal semantics and the main theorem of this paper, stating the desired properties of MIMOS. Section 4 develops a fixed-point semantics for the model, establishing formal proofs for the main theorem. Section 5 presents open verification problems on the model, to be addressed in future work. Section 6 concludes the paper.

2. Contributions and Related Work

One of the main challenges in embedded real-time systems design is to ensure that the resulting system has deterministic input-output and predictable timing behavior (typically with deterministic input-to-output latency or known time bounds) even when multiple system functions are integrated and co-execute on a platform with limited resources. The deterministic semantics allows model-in-the-loop simulation using successful tools like Simulink/Stateflow to simulate and verify the complete system behavior. Over the past decades, numerous approaches to address this challenge have been devised by research communities in hardware, software, control, and communication. Several, including the *synchronous approach*, embodied by the languages Esterel, Lustre, and Signal [14], and the time-triggered paradigm promoted by Kopetz [18], ensure deterministic behavior by scheduling computation and communication among components at pre-determined time points. This results in highly reliable and predictable systems, but severely restricts the possibility to modify or update systems after deployment. The reason is that new components must fit exactly into the already determined time schedules, and components may perturb each others' timing via shared resources. In recent years, dynamic updates of real-time systems after deployment have attracted increasing interest. A model-based approach to the design and dynamic updates for cyber-physical systems is proposed in [27]. The work of [11] demonstrates that autonomous systems in operation can be updated through contract negotiation and run-time enforcement of contracts.

Contributions. We present a semantic model for real-time systems which on the one hand, ensures the deterministic input-output and predictable timing behaviors of a system, and on the other hand supports incremental updates after deployment without re-designing the whole system. In this model, a real-time system is described as a network of software components connected by communication channels. We provide a simple but expressive description language named MIMOSto formalize such networks where each component is designed to compute a collection of functions over data streams and the communication channels can be of two types: FIFO queues for buffering inputs and outputs, and registers for sampling time-dependent data from sources such as sensors or streams that are written and read at different rates. Components are further specified as real-time tasks to enforce that they read inputs, compute, and write outputs at time points satisfying certain time constraints. A fixed-point semantics is developed for the model, showing that it enjoys two desired properties: (1) such a network of real-time software components computes a set of functions, each one defined from a set of (timed) input streams to a unique (timed) output stream. (2) The network can be modified by integrating new components for adding new system functions or replacing existing components by refined ones (e.g. for better performance or security patches) without re-designing the whole system or changing the original system functions.

Related Work. An example of a time-triggered language developed for real-time systems is Giotto [15]. A Giotto program is a set of periodic tasks that communicate through ports. Giotto implements the synchronous semantics, preserving timing determinism and also value-determinism (but not determinism over sequences of values i.e., streams as in our model) by restricting to periodic tasks where reading from and writing to ports is fixed and performed at deterministic time points. It does not allow asynchronous communication via FIFO channels as MIMOS. This limits the possibility of updating a system in operation. A more recent work addressing the quasi-synchronous semantics of [8] is presented in [6]. The work also proposes to use multiple periodic tasks to implement the synchronous semantics on parallel and distributed architectures. It remains in the category of synchronous approaches to real-time programming without addressing issues related to dynamic updates. MIMOS can be viewed as a timed extension of Kahn Process Networks (KPN) [16]. In the literature, there have been various extensions to KPN. A special case of KPNs is dataflow process networks (DPN) [21]. A DPN is

a general dataflow model where each process is specified as repeated firings of a node. A node becomes enabled for execution according to a set of *firing rules*. However, no time constraints are specified in the firing rules. An implementation of KPN with bounded-size buffers is proposed in [10]. In this work, a composition approach preserving the Kahn semantics is presented for components whose production and consumption rate are the same in the long run. The work, however, is confined within the synchronous programming model. Related to the communication channels of KPN, a time-aware implementation of C, called *Timed C*, has been proposed in [23]. In Timed C, a program consists of a set of tasks communicating through two types of channels: *FIFO* and *Latest Value (LV)*. Analogous to KPN, reading from FIFO is blocking while writing is non-blocking. In contrast, reading and writing of LV channels are non-blocking. This communication model is similar to MIMOS. However, while Timed C is a general programming language without guaranteed determinism, we focus on both functional and timing determinism, and study these properties in a well-defined formal semantics. A standardized software architecture for automotive domain is developed by AUTOSAR [3]. Based on this, an application is organized as a collection of software components which perform data communication through a sender/receiver model. Data is processed by a receiver using a *queue* or a *last-is-best* policy. Our model can be thought of as a specialization of this approach which has a formal and deterministic semantics. Due to the known fact that AUTOSAR is only a reference model for automotive software architecture with various implementations and without a formal semantics, any formal proof is impossible.

3. The MIMOS Model

In this section, we present MIMOS based on Kahn Process Networks (KPN) [16]. A KPN is an abstract model of a parallel system consisting of a collection of processes connected by FIFO queues for data exchange. We view real-time systems as such a network where the computations as well as the respective input and outputs of the processes must meet given time constraints. Our model can be viewed as a timed version of KPN whose nodes are extended with timing constraints and edges with registers for sampling time-dependent inputs in addition to FIFO queues.

As KPN, MIMOS is essentially a simple description language to formalize system models. In this section, we present the main primitives and informal semantics of MIMOS. A formal fixed-point semantics is given in Section 4.

3.1. Preliminaries on Kahn Process Networks

Here, we recall the notion of KPN and its main properties. A KPN is a set of stand-alone processes, called nodes, which communicate through a set of *FIFO* channels. A node accesses channels through two operations: **read** and **write**.

Definition 1 (KPN). A Kahn Process Network $\mathcal{N}$ is a set of processes, called nodes, and a set of FIFO queues, called *channels*. Nodes behave according to the following rules.

- Each node computes a tuple of functions. For a set of input sequences, each of the functions defines a unique output sequence. A node may not access the state/data of the other nodes.
- Channels are of potentially unbounded capacity. At most one node is allowed to read from / write to each channel. However, a node may copy an output to multiple channels read by multiple readers.
- **read** from a channel is blocking, that is, the node is blocked until all data required for the next step is available, implying that a node cannot examine the emptiness of the channels; **write** to a channel is non-blocking.

A node of a KPN can be implemented by a set of local variables and a procedure, repeated indefinitely. The procedure may be specified in any conventional programming language, e.g., C.

Example 1. An example of a KPN program is shown in Listing 1. Nodes are defined by the `process` keyword. The procedure executed by a node is written in a `Repeat` block. The structure of this program is depicted in Fig. 1, where arrows represent FIFO channels.

```
process f(int out V) {
  Repeat {    write 1 on V;   }
}
process g(int in U; int threshold; int out V) {
  int count = 0;        // local variable
  Repeat {
    read(U);            // read from a channel
    count = count + 1;
    if count == threshold
      write 1 on V;     // write to a channel
      count = 0;
  }
}
int channel X, Y;
f(X) || g(X, 5, Y);     // concurrent execution
```

Listing 1. A sample KPN program.

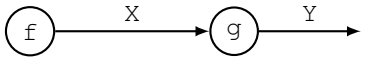

Figure 1. Structure of the program in Listing 1.

A KPN can be seen as a parallel program computing a set of functions from a set of input streams to a set of output streams defined by the least fixed point obtained by computing node functions in an arbitrary order [16]. Streams refer to complete histories of elements seen on some FIFO or output, and they are formally defined in Section 4.

The most important property of KPNs is their *determinism*. This holds under any sufficiently fair scheduler, i.e., schedulers which do not postpone a process indefinitely.

Theorem 1 (Functional Determinism of KPN [16]). Given a set of input streams, the set of output streams computed by a KPN is unique.

Theorem 1 indicates that implementation aspects, such as execution order, scheduling, and platform speed do not affect the functional behavior of a system implementing a KPN model.

3.2. Timed KPN (TKPN)

The order- and speed-independent functional determinism of KPN leads to a natural formalization of a timed version of KPN. Real-time systems are modelled as a KPN where each node is executed according to a real-time task model [26], specifying a release pattern which is a sequence of time points over the time line, and a deadline for each release.

Definition 2 (TKPN). The timed version denoted $\mathcal{N}_T$ of a KPN $\mathcal{N}$ is obtained by associating with each node **n** of $\mathcal{N}$ a release pattern and a positive integer, called *deadline*.

The release pattern of a TKPN node can be described using the well-established real-time task models [26], such as periodic tasks [22], generalized multiframe [5], DAG [4], or DRT [25] and timed automata [12] as long as they are deterministic.

Note that each node of a TKPN computes a tuple of functions, one for each output channel. If different functions have different time constraints, different deadlines may be assigned to the respective output channels.

Note also that in Definition 2, the internal structure and resource requirement for the nodes of a TKPN and the scheduling algorithm to be adopted in the implementation are left open. Only the time constraints (i.e. the release patterns and deadlines for the executions of nodes) are specified.

Informally, the operational behavior of a node in $\mathcal{N}_T$ is defined as follows. When the node is released, and if all needed inputs are available, it computes and delivers the resulting outputs, if any, within the given deadline. To achieve timing determinism, the inputs of a node are read at release time, outputs are delivered at the deadline. This read-execute-write approach is similar to the *implicit* communication model of AUTOSAR [1]. A formal semantics of TKPN is presented in Section 4.

Because a TKPN is also a KPN, and the execution rates assigned to nodes only restrict more explicitly the computation order of eligible nodes, the behaviour of a TKPN enjoys the desired functional determinism, which follows directly from Theorem 1. Furthermore, it enjoys also the timing determinism as declared later in Theorem 2.

3.3. MIMOS: TKPN with Further Extensions

In this section, we present our complete model. For this, we first extend TKPN with a new type of channel, called registers for sampling time-dependent data. Next, we augment the model with merge nodes for non-blocking reading of data from different sources.

TKPN with register-channels.. In real-time applications, inputs may be produced by the physical environment, and hence, the corresponding value may be time-dependent. In Cyber-Physical Systems, such inputs come typically from sensors sensing physical phenomena. The system usually does not need all data produced by a sensor but only the latest value. Additionally, the refresh rate of the sensor is not necessarily compliant with the execution rate of the node(s) reading the sensor. In this case, using a FIFO may lead to memory overflow or blocked computation (in case the FIFO is empty). In such situations, it is useful to have a communication channel which keeps only the most recently written value. We extend TKPN with such channels, called *register*.

The operations to access registers are syntactically the same as the ones to access FIFOs. We adopt the "last-is-best" semantics of [3]. **write** to a register over-writes the current value. **read** from a register is non-blocking. When both **read** and **write** occur at the same time, the current value is updated by **write** before it can be **read**.

Example 2. Listing 2 shows the program in Listing 1 extended with a register and time constraints. The program structure is illustrated by Fig. 2, where FIFO channels are represented by solid-line arrows, and registers by dashed arrows.

In this example, using a register instead of a FIFO to carry the *threshold* values has the advantage to (1) always use the most recent value available (the currently valid one), and (2) guarantee absence of buffer over- or underflow independently of the speed at which threshold values are produced and read.

```
process f(int out V) { ... }          // unchanged

process h(int out V) {
  Repeat {    write 6 on V;   }
}

process g(int in U; int in C; int out V) {
  int count = 0;
  int threshold;
  Repeat {
    read(U);              // reading (from FIFO)
    count = count + 1;
    threshold = read(C); // reading (from register)
    if count >= threshold then
      write 1 on V;
      count = 0;
  }
}
// Instantiating and connecting the components.
int channel FIFO X, Y;
int channel register Z = 5;   // initial value
f.timings = periodic(10, 10); // period=deadline=10
g.timings = periodic(10, 10);
h.timings = periodic(10, 10);
f(X) || h(Z) || g(X, Z, Y);
```

Listing 2. A sample program in Extended TKPN.

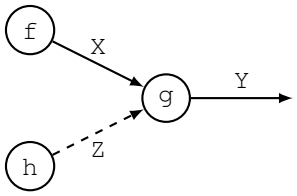

Figure 2. The structure of the program in Listing 2. Dashed arrow indicates a *register*.

TKPN with merge-nodes.. We further extend the model with a special type of nodes called *merge*. A merge node has at least two FIFO inputs. At each activation, the node reads all data available in its input FIFOs. The output, which is written to a FIFO, is all the data read from the first input FIFO, followed by the ones from the second input FIFO, and so on. That is, a prioritized timed merge where all data items arriving during a release period are considered to "have arrived at the same time".

The motivation for defining this type is that in reactive systems, there are (external) events, e.g., requests for the same services from different sources, to which the system must react. No input event is allowed to be missed. Suppose now that events come from distinct producers so that distinct FIFO's are needed. Assume that the node should do a computation whenever there is some data in any input FIFO. This is beyond the expressiveness of KPN. Our time-dependent merge-function solves this very typical problem, common in real-time applications.

Definition 3 (*MIMOS*: Extended TKPN). TKPN extended with registers and merge is TKPN where some channels can be a register instead of a FIFO and some nodes may be a merge-function. We call this extension of TKPN MIMOS.

The formal semantics for MIMOS developed in Section 4 shows that determinism is preserved if we see them as functions of the input data streams *and* their arrival times.

Theorem 2 (Functional and Timing Determinism of* MIMOS*). Given a set of input data streams and the arrival times of data items in the FIFO's (or registers), the set of output data streams computed and the time points at which the data items are inserted into the output FIFO's (or registers) by an extended TKPN is unique.

The result follows from Proposition 2 of Section 4.

3.4. Design and Update with MIMOS

A model-based approach can be sketched as follows[1]. First, to build a new system, a set of system functions to be implemented must be specified in terms of functional and timing requirements on their inputs and outputs as well as the respective end-to-end latency (see Section 5). A MIMOS model may be constructed, verified to satisfy the given requirements and compiled into code executable on the target platform to compute these functions. Prior to any update over the life cycle of the system, its original MIMOS model may be extended (i.e., updated) by connecting the outputs of the existing components (KPN nodes) to the new ones. Additionally, existing components may be replaced also by new ones fulfilling given requirements. Thanks to the independence of reading from/writing to channels, the added (or updated) system functions will not interfere with the existing ones. Thanks also to the deterministic semantics, it can be verified based on the updated model that the resulting system will satisfy the functional and timing requirements. Further, it must be verified that the platform is able to provide enough resources to meet the resource requirements of the new components by schedulability analysis and analysis of memory usage (see Section 5). If all verification steps are successful, the new components can be deployed (or installed). Otherwise, the update is rejected.

1. Addressing the different steps in details, including specification, modelling, verification and compilation is not in the scope of this paper.

4. Fixed-Point Semantics

Here, we present a formal semantics for MIMOS. We first recall definitions used in [16] to prove the order and time independent determinism of KPNs of Definition 1. We introduce a notion of timed stream to define the semantics of extended TKPNs of Definition 3 and to prove the main theorem of Section 3.3.

4.1. Preliminaries on the Semantics of KPN

First, we recall some basic notations from [16]. The function $\mathbf{F}$ associated with a node of a KPN is represented as a function from a set of input streams to a set of output streams. More precisely, $\mathbf{F}$ represents a tuple of functions, one for each output.

We now formally define streams and functions. We consider streams on a generic domain $\mathbb{D}$ which may be instantiated by any data domain. To ensure generality, we consider the time domain to be reals $\mathbb{R}$.

Definition 4 (Streams, time streams and timed streams). Let the stream domain $\mathbb{S}$ be the set of infinite sequences in $\mathbb{D}^\infty$. The domain of time streams $\mathbb{T}$ is the set of infinite sequences of time points in $\mathbb{R}^\infty$ with (not necessarily strictly) increasing time points which diverge[2].
The domain of *timed streams* $\mathbb{S} \times \mathbb{T}$ are infinite sequences in $(\mathbb{D} \times \mathbb{R})^\infty$ such that every timed stream can be denoted as $\mathbf{S} \times \mathbf{T}$ for two appropriate streams.
We use $\sqsubseteq$ to stand for the standard prefix order on sequences, λ for the empty sequence, and "$\cdot$" for concatenation.

Note that a time stream may be regarded as a particular case of a data stream. As in [16], functions $\mathbf{F}$ are built from the following basic functions on streams.

Definition 5 (Functions on streams). Consider the following functions from $\mathbb{S}^k$ to $\mathbb{S}$:

2. That is, we assume time streams to be non Zeno.

1) Data transformations: lift any k-ary data transformation function $\mathbf{f} : \mathbb{D}^k \mapsto \mathbb{D}$ to a stream transformation function $\mathbb{S}^k \mapsto \mathbb{S}$, with the same name: $\mathbf{f}(a_1 \cdot \mathbf{S}_1, \ ... \ a_k \cdot \mathbf{S}_k) = \mathbf{f}(a_1, \ ... \ a_k) \cdot \mathbf{f}(\mathbf{S}_1, \ ... \ \mathbf{S}_k)$.

2) Standard order preserving stream manipulating functions "first", "remainder" and "append" (which we rarely use explicitly): $\mathbf{First}(a \cdot \mathbf{S}) = a$; $\mathbf{R}(a \cdot \mathbf{S}) = \mathbf{S}$ (skips the first element of a stream); $\mathbf{app}(\mathbf{S}, i_0 \cdot init) = i_0 \cdot \mathbf{S}$ (adds an initial element to the left by pushing the input stream to the right).

Example 3 (Illustrating Example). Consider node g of Fig. 1 with input X and output Y. We give the equations for all output streams using functions of Definition 5. Node g has a local variable $count$ which gives rise to 2 streams: c_M, the stored values used as input of g, and c, the values produced by g. `threshold` is a constant parameter $thsh$. Note that g is independent of the actual data read, it just consumes a data item at each iteration. We obtain the following fixed-point equations:

- $Y = g_Y(X, c_M, thsh)$
- $c = g_c(X, c_M, thsh)$
- $c_M = \mathbf{app}(0, c)$ where
- $g_c(x \cdot X, c \cdot c_M) =$ [if $(c + 1 < thsh)$ then $(c + 1)$ else 0] $\cdot \ g_c(X, c_M)$
- $g_Y(x \cdot X, c \cdot c_M) =$ [if $(c + 1 < thsh)$ then λ else 1] $\cdot \ g_Y(X, c_M)$

A typical function applies some transformation to the first elements of the input streams, produces an output (or alternatively produces nothing), and is applied recursively to proceed with the remainder of the streams. But a function may in each recursion step read zero or more elements from some input streams, and write zero or more elements to its output, as long as inputs are read in a FIFO order, the number of elements to be read is deterministically defined, and there is some "progress".

4.2. Semantics of Timed KPN

We define the semantics of a timed node with a release pattern and a deadline. In order to do so, we show that we can extend each function $\mathbf{F}$ on data streams (the semantic function for one of the functions of a KPN) to a function $\mathbf{F}_\delta$ on timed streams, such that (1) $\mathbf{F}_\delta$ defines a pair of streams consisting of the data stream defined by $\mathbf{F}$, and the stream of time points at which data elements are written. (2) $\mathbf{F}_\delta$ is an ordinary Kahn function if time streams are just considered as particular data streams. (3) the time extension corresponds to the intuition of release pattern P and the output delay δ of Definition 2. We now state the proposition, the remainder of the subsection is dedicated to its proof.

Proposition 1. The semantics of a TKPN is a deterministic mapping from timed input streams to timed output streams defined by a set of functions $\mathbf{F}_\delta$.

We prove this proposition by constructing a function $\mathbf{F}_\delta$ for any data stream transformation $\mathbf{F}$. Fig. 3 illustrates $\mathbf{F}_\delta$

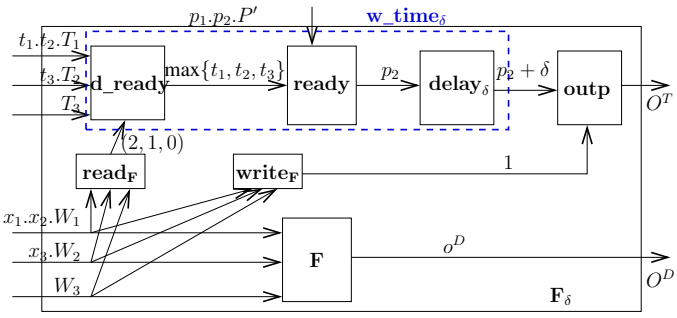

Figure 3. Graphical representation of F_δ

for a function $\mathbf{F}$ with 3 input streams. The data output is the one produced by $\mathbf{F}$. The time points associated with data elements are meant to represent the time point at which the data is inserted into the FIFO. Suppose that it holds for the input streams. This motivates our method for computing the time stamps of output data. It works as follows: (1) calculate the maximal time point associated with the data elements read by $\mathbf{F}$ (the time when all required data are in the FIFO), (2) calculate the actual "release" or "ready" time which is the release point of P just after data is ready. (3) the output writing time point is obtained by adding δ and (4) this time point is output if and only if $\mathbf{F}$ outputs a data item at this step. First, we introduce the necessary auxiliary functions:

1) "data-ready" function $\mathbf{d\_ready}$: takes as input k time streams $\mathbf{T}_i$, one for each input data stream of $\mathbf{F}$, and for each stream the number of elements to be read, and outputs the time when all data is ready: $\mathbf{d\_ready}((t_{11}...t_{1\ell_1} \cdot \mathbf{T}_1, \ ... \ t_{k1}...t_{k\ell_k} \cdot \mathbf{T}_k), (\ell_1 \cdot L_1, \ ... \ \ell_k \cdot L_k)) = max\{t_{1\ell_1}, \ ... \ t_{k\ell_k}\} \cdot \mathbf{d\_ready}((\mathbf{T}_1, \ ... \ \mathbf{T}_k), (L_1, ... \ L_k))$.
In each time stream, $t_{i\ell_i}$ is the time stamp of the most recent data, hence $max\{t_{1\ell_1}, \ ... \ t_{k\ell_k}\}$ is the global *ready time* of the input. Some ℓ_i may be zero, meaning that no data is read from the corresponding stream, and there is no $t_{i\ell_i}$ contributing to the maximum.
2) "ready" function $\mathbf{ready}$: given time streams P and R (representing respectively a "release pattern" and "data ready times"), the stream of "ready times" for computation, the time points from which the deadline runs, is obtained by eliminating elements of P at which no data is ready: $\mathbf{ready}(p \cdot P', r \cdot R') =$ if $(p \geq r)$ then $p \cdot \mathbf{ready}(P', R')$ else $\mathbf{ready}(P', r \cdot R')$.
3) "delay" function $\mathbf{delay}$: increases all time points of a time stream by a "delay" δ. For a stream $\delta \geq 0$ (i. e. element wise ≥ 0), $\mathbf{delay}(\mathbf{T}, \delta) = \mathbf{T} + \delta$. If δ is a constant, we note $\mathbf{delay}_\delta(\mathbf{T})$ the corresponding function with a single argument.
4) Given a k-ary function $\mathbf{F}(X_1^D, \ ... \ X_k^D)$ on data streams, we denote
$\mathbf{read_F}(X_1^D, \ ... \ X_k^D)$ and $\mathbf{write_F}(X_1^D, \ ... \ X_k^D)$

two functions which read the same inputs as $\mathbf{F}$. $\mathbf{read_F}$ outputs a k-tuple of integers indicating the number of elements that $\mathbf{F}$ reads at each computation step, and $\mathbf{write_F}$ outputs the number of elements that $\mathbf{F}$ outputs at each computation step. For any function $\mathbf{F}$, these functions can be obtained by "code analysis"[3].

5) $\mathbf{outp}$: given a time stream and a stream of numbers in $[0...1]$, it outputs the time point if the number is 1: $\mathbf{outp}(t{\cdot}T, c{\cdot}C)$ = if $(c == 1)$ then $t{\cdot}\mathbf{outp}(T, C)$ else $\mathbf{outp}(T, C)$. It is easy to extend to the case $c > 1$, where the time point is written more than once. We use this function in a context where c represents the number of elements written by a function $\mathbf{F}$, meaning that $\mathbf{outp}$ guarantees that the number of time points written matches the number of elements written by $\mathbf{F}$.

Now we can define $\mathbf{F}_\delta$ by composing the previously defined functions as suggested by Fig. 3, where function $\mathbf{w\_time}_\delta$ is represented by a blue dashed box.

6. $\mathbf{F}_\delta((X_1, \ ... \ X_k), P) = (\mathbf{F}(X_1^D, \ ... \ X_k^D), \mathbf{outp}(\mathbf{w\_time}_\delta((X_1^T, \ ... \ X_k^T), \mathbf{read_F}(X_1^D, \ ...X_k^D), P), \mathbf{write_F}(X_1^D, ...X_k^D))$ where
7. $\mathbf{w\_time}_\delta((X_1^T, \ ... \ X_k^T), (\ell_1, \ ... \ \ell_k), P)$ = $\mathbf{delay}_\delta$ $(\mathbf{ready}(P, \mathbf{d\_ready}((X_1^T, \ ... \ X_k^T), (\ell_1, \ ... \ \ell_k))))$.

To sum up, $\mathbf{F}_\delta$ has the promised characteristics: (1) it defines a pair of streams consisting of the data stream defined by $\mathbf{F}$, and a stream of time points at which data elements are written. This proofs that $\mathbf{F}$ is preserved. (2) It is an an ordinary Kahn function if time streams are considered as particular data streams. This proves timing determinism based on [16]. (3) It produces time streams according to the intuition of Definition 2. This completes the proof.

Section 3.2 makes the choice that deadlines run from the first release point after all data is ready. The definitions here correspond to this choice. But the definition of $\mathbf{F}_\delta$ can easily be adapted to alternative choices.

According to Definition 3, δ represents an exact output write delay, that is, the data is actually written into its FIFO at the time point defined by $\mathbf{F}_\delta$. Note that exactly the same definition may be used to represent a best or worst case execution time or a "latest due time", for example.

Example 4 (Illustrating example, continued). Consider again function g of Fig. 1. Now, g is executed periodically with period `period` (noted `p`), and has a output delay `deadline` (noted `dl`). As functions on data remain untouched, we only need to add equations defining the time streams associated with Y, c and c_M using the previous definitions. As we know that at each step exactly one data is read from each input, the definition of timed streams is slightly simplified. Note that function g_Y may or may not produce output, whereas g_c always produces output:

- $Y^T = \mathbf{outp}(\mathbf{w\_time}_{\mathtt{dl}}((X^T, c_M^T), (1,1), \ \mathtt{p}), \mathbf{write}_{g_Y}(X^D, c_M^D))$,
- $c^T = \mathbf{outp}(\mathbf{w\_time}_{\mathtt{dl}}((X^T, c_M^T), (1,1), \mathtt{p}), 1)$, and
- $c_M^T = c^T$.

For function M representing the variable `count`, we consider that once the variable is written (defined by time stream c^T), it is available immediately for the next step, and no additional delay is added.

4.3. Semantics of MIMOS

We now provide the semantic underpinning for the full model MIMOS (Extended TKPN) by defining also "register" and "merge" as transformers of timed streams. As motivated in Section 3.3, such functions depend on the data that is put in the FIFO (or a register) at the release time points of the function. At the semantic level, this means that they read all the (not yet treated) data items with time stamp up to some time point – the release time of the function – and compute some output depending on these data items.

As motivated in Section 3, a merge-node is a timed stream transformer merging incoming timed streams into a timed stream – a total order – according to the time points at which data items inserted in the respective FIFO's. We view registers also as a timed stream transformer transforming the incoming timed stream to a timed stream containing data items (and also their arrival times) according to the register-reading ratio, i.e., the release pattern of the target node.

We define the semantics of Extended TKPN and prove its determinism. In order to do so, we show that we can define k-ary functions $\mathbf{rreg}$ and $\mathbf{merge}$ from timed streams to timed streams (1) which correspond to the intuition of the concepts "register" and "merge" introduced in Section 3.3, and (2) which are ordinary Kahn functions if time streams are considered as ordinary data streams

We first state the proposition that, together with Proposition 1, guarantees Theorem 2, and the remainder of the subsection is dedicated to its proof.

Proposition 2 (Extended TKPN). The semantics of an Extended TKPN is a deterministic mapping from timed input streams to timed output streams.

We prove this proposition by constructing the above mentioned functions $\mathbf{rreg}$ and $\mathbf{merge}$ which together with the previously defined functions $\mathbf{F}_\delta$ define the semantics of Extended TKPN.

$\mathbf{rreg}$ has two arguments, the timed data stream $(\mathbf{S}, \mathbf{T})$ written to the register, and a time stream $\mathbf{T}^S$ representing the time points up to which the data are to be read at every activation. The output is a timed data stream representing the (timed) data actually read. For the sake of simplicity, suppose that $\mathbf{T}$ starts at time 0 and $\mathbf{T}^S$ at a time ≥ 0.

1. Define $\mathbf{rreg}((\mathbf{S}, \mathbf{T}), \ \mathbf{T}^S)$ inductively:
   $\mathbf{rreg}((s_1, t_1) \cdot (s_2, t_2) \cdot (\mathbf{S}, \mathbf{T}), t \cdot \mathbf{T}^S)$ = if $t_2 < t$ then $\mathbf{rreg}((s_2, t_2) \cdot (\mathbf{S}, \mathbf{T}), t \cdot \mathbf{T}^S)$ else $(s_1, t_1) \cdot$

3. By "code", we mean the definition of $\mathbf{F}$ in terms of the basic functions of Definition 5.

$\mathbf{rreg}((s_1,t_1)\cdot(s_2,t_2)\cdot(\mathbf{S},\mathbf{T}),\mathbf{T}^S)$.
We may denote **rreg** as a pair of functions $(\mathbf{rreg}^D,\mathbf{rreg}^T)$.

The if-clause represents the case where the first element (s_1,t_1) is an "overwritten data" to be skipped, and the else-clause the case where (s_1,t_1) is the newest data with a time stamp prior to t. Note that in this second case, (s_1,t_1) is not "consumed" but reread at the next iteration, so as to guarantee that for any time point t' of the remaining $\mathbf{T}^S$, there is at least one data older than t'. Elements that are written into the "register" but "overwritten" before the next time point of reading are filtered out, and elements that are to be read more than once are written several times. This represents the functionality of a "register" read at the time points $\mathbf{T}^S$.

merge has as arguments (at least) three arguments, timed data streams $(\mathbf{S}_i,\mathbf{T}_i)$ representing the FIFOs to be merged, and a time stream $\mathbf{T}^S$ representing the time points up to which the data are to be read and merged at each activation. The output is a timed data stream representing the merged FIFO and their data arrival times.

2. Define $\mathbf{merge}((\mathbf{S}_1,\mathbf{T}_1),\ (\mathbf{S}_2,\mathbf{T}_2),\ \mathbf{T}^S)$ inductively:
$\mathbf{merge}((d_1,t_1)\cdot(\mathbf{S}_1,\mathbf{T}_1),\ (d_2,t_2)\cdot(\mathbf{S}_2,\mathbf{T}_2),\ t\cdot\mathbf{T}^S)$ =
if $t_1 < t$ then $(d_1,t)\cdot\mathbf{merge}((\mathbf{S}_1,\mathbf{T}_1),\ (d_2,t_2)\cdot(\mathbf{S}_2,\mathbf{T}_2),\ t\cdot\mathbf{T}^S)$
else if $t_2 < t$ then $(d_2,t)\cdot\mathbf{merge}((d_1,t_1)\cdot(\mathbf{S}_1,\mathbf{T}_1),\ (\mathbf{S}_2,\mathbf{T}_2),\ t\cdot\mathbf{T}^S)$
else[4] $\mathbf{merge}((d_1,t_1)\cdot(\mathbf{S}_1,\mathbf{T}_1),\ (d_2,t_2)\cdot(\mathbf{S}_2,\mathbf{T}_2),\ \mathbf{T}^S)$

That is, data with time stamps inferior to t are taken from the two input streams by first moving the data items from the left queue, and then those from the right. This corresponds to the informal definition which does not order elements of one reading interval according to their arrival times but only according to the priority, and their time stamps are the "sample time points". Proceeding to the next time point of time stream $\mathbf{T}^S$ (if no relevant data item is left) corresponds to a new activation of the merge function.

To sum up, the functions **rreg** and **merge** have the required characteristics: (1) they produce the intuition of the concepts "register" and "merge" introduced in Section 3.3, and (2) they are Kahn functions. All functions of an Extended TKPN can be composed from functions of the form $\mathbf{F}_\delta$, by **rreg** or by **merge**. This guarantees their determinism. This completes the proof.

Function **merge** can be implemented if the data items present in the FIFO at the activation time points can be deterministically defined. If data items can be written at sufficiently precisely defined time points, this is the case. The guarantee that data items must be present at a certain time point, also allows to detect when there is "no data item available in a given interval", which may be either just normal or mean that a run-time error has occurred and trigger some exception handler.

4. that is, $t_1,t_2 \geq t$ meaning no data to be merged up to t is left

Note also that the condition on deterministic writing times cannot be relaxed without losing the guarantee of determinism if **rreg** is implemented by a simple register (that is memory). If time points associated with data streams represent "latest" writing times for example, determinism can nevertheless be preserved by using more than just one memory: one for the data to be ready at the next release time, and one or more memories for release time points further in the future. Similarly, to achieve the timed merge function one would need to group data according to the activation period of their "latest writing times". This is an adaptation of Caspi's protocol defined in [9] to our framework. Note however, that in both cases it requires to time stamp data explicitly at implementation level.

Example 5 (Illustrating example, continued). Again, consider function g (now Fig. 2 of Example 2). Now, `threshold` is defined by an input Z, a register. This may an important effect, on both time and data.

- As a register holds a valid data at any time, Z^T does not influence $\mathbf{d\_ready}$, and time streams do again not depend on Z^T.
- The data streams of Y and c have now 3 input streams and are computed by replacing parameter $thsh$ by $\mathbf{rreg}^D(Z,\mathbf{ready}(\mathrm{p},\mathbf{d\_ready}(X^T,c_M^T)))$.

5. Analysis Problems

To update a system by adding a new system function, its model (in MIMOS) should be modified to integrate the new function. To enable that the modified model can be compiled into a program executable on a platform with limited resources, it must be verified to meet given timing and resource constraints. Thanks to the deterministic semantics of MIMOS, properties verified on a MIMOS model will be preserved by the execution of the compiled code on a platform satisfying the resource assumptions adopted in the verification process.

There are two principle timing and resource constraints: (1) the memory requirements must be bounded and (2) system functions including the new one must satisfy the end-to-end latency constraints [2]. In general, these verification problems are undecidable. However, with proper assumptions and restrictions, there are efficient solutions for practical purposes [19], [2]. First, the required memory of a system in operation depends on the buffer size required by the FIFOs, which can be specified as follows.

Definition 6 (Required buffer size (RBS)). Assume that the data written to and read from a FIFO buffer are specified, respectively, by timed streams $(a_1,t_1)(a_2,t_2)...$ and $(a_1,t'_1)(a_2,t'_2)....$ Also, let $\omega(t)=\max\{i|t_i\leq t\}$ and $\gamma(t)=\max\{i|t'_i<t\}$. The FIFO's *required buffer size* (RBS) is defined as $\max\{\omega(t)-\gamma(t)|t\geq 0\}$.

In words, $\omega(t)$ is the total number of items written to a FIFO up to (including) time t, and $\gamma(t)$ is the total number of items read from the FIFO strictly before t. Based on this, the RBS

of a FIFO denotes the maximum number of items which may simultaneously exist in the queue. Indeed, computing RBS in a *process network* has been shown undecidable in the general case [7]. Despite this, the measure is computable for special settings. For instance, if for each node, the number of produced and consumed items is fixed in all firings, as is the case in synchronous data flow (SDF) [20], the problem has efficient solutions [20]. Further, for those KPNs in which data producing/consuming pattern of the nodes is periodic (except for a bounded initial time), it is shown that the required capacity for a FIFO is bounded if and only if writing and reading rates are *asymptotically* the same [10].

The RBS of a MIMOS model depends on the release pattern of the nodes, the pattern by which input data arrives, and also the data consumption pattern. According to these factors, a variety of instances of the problem of computing (a bound on) RBS can be defined, which we leave for future work. Here, we just provide some initial observations.

A fairly direct consequence of Definition 6 is that the RBS of a FIFO in a MIMOS model is bounded if and only if a constant c exists for which: $\forall t \geq 0 : \omega(t) - \gamma(t) \leq c$. Based on this, we conjecture that: *The RBS of a FIFO is bounded if and only if reading and writing rates are asymptotically the same, i.e.,* $\lim_{t\to\infty}\{\omega(t)/\gamma(t)\} = 1$.

The other measure to be analyzed is *end-to-end latency*, which essentially reflects the responsiveness of a system. It is important that when the input changes, or for an event arriving in the input queue, the system provides a response (or react) within a bounded delay. As an essential requirement in a real-time system [2], we define the end-to-end latency for each output channel of the system with respect to an input on which it depends.

Definition 7 (Worst-case end-to-end latency, $\mathbf{e2e}(i)$***).*** Consider a system function specified by a k-ary function $\mathbf{F_s}$ on streams. Let $(I_1, \ldots I_k)$ be a set of input timed streams for which $\mathbf{F_s}(I_1, ...I_k) = (Y^D, Y^T)$. Consider now, for some i, $1 \leq i \leq k$, a modified version of I_i, called I'_i, which is obtained by changing the j-th entry of I_i from (a, t) to (b, t). Assume $\mathbf{F_s}(I_1, \ldots, I'_i, \ldots I_k) = (Y'^D, Y'^T)$. Let $j' = \min\{m | Y'^D_m \neq Y^D_m \text{ or } Y'^T_m \neq Y^T_m\}$, where X_m denotes the m-th element of a stream X. We define $delay(I_1, ...I_k, i, j, b) = Y'^T_{j'} - t$. Accordingly, we define $delay(I_1, ...I_k, i) = \max\{delay(I_1, ...I_k, i, j, b) | \forall j, b\}$. The worst-case end-to-end latency for the i-th input line is then

$$\mathbf{e2e}(i) = \max\{delay(I_1, ...I_k, i) | \forall (I_1, ...I_k)\}$$

Computing end-to-end latency can be studied in terms of the release patterns of the nodes in a MIMOS model. For instance, for a set of periodic tasks communicating through a set of registers, which can be viewed as a special case of Extended TKPN, the problem has been explored in [17] which presents a worst-case analysis method with exponential time complexity. Also, a polynomial-time approach is provided for computing an upper bound. Both methods are limited to task sets scheduled with a fixed-priority policy on a single processor. Investigating the problem in MIMOS for different release patterns and target platforms such as multi-core and distributed architectures is left to future work.

6. Conclusions

This paper presents a deterministic timed model (MIMOS) to enable a model-based approach allowing for not only building deterministic real-time systems, but also verifying and updating them after deployment. MIMOS is a timed extension of Kahn's Process Network with (1) timing constraints on the execution of KPN nodes and (2) register-channel and merge-node to deal with time-dependent data and functions. MIMOS is proven to preserve functional and timing determinism. More precisely, given a set of input data streams and the corresponding arrival times of data items in the input channels of a MIMOS model, the set of output data streams computed by its network nodes, and the time points at which the data items are inserted into its output channels is unique.

To further develop a programming (or coordination) language based on MIMOS, and a compiler for such a language, several challenging verification problems (see Section 5) must be solved. First, a program (or a MIMOS model) must be analyzed to ensure that its memory requirement is bounded and meet the platform limitations. Second, the end-to-end latency for system functions computed must satisfy given timing requirements. Future work includes also a symbolic semantics for the model allowing uncertainty in the implementation e.g. when data exchange through reading from and writing to channels may occur over time intervals.

References

[1] AUTOSAR - Specification of RTE Software (2019)

[2] Abdullah, J., Dai, G., Yi, W.: Worst-case cause-effect reaction latency in systems with non-blocking communication. In: DATE. pp. 1625–1630 (2019)

[3] AUTOSAR: AUTomotive Open System ARchitecture, https://www.autosar.org, https://www.autosar.org

[4] Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L., Wiese, A.: A generalized parallel task model for recurrent real-time processes. In: IEEE 33rd Real-Time Systems Symposium. pp. 63–72. IEEE (2012)

[5] Baruah, S., Chen, D., Gorinsky, S., Mok, A.: Generalized multiframe tasks. Real-Time Systems **17**(1), 5–22 (1999)

[6] Baudart, G.: A synchronous approach to quasi-periodic systems. Theses, PSL Research University (Mar 2017), https://tel.archives-ouvertes.fr/tel-01507595

[7] Buck, J.T., Lee, E.A.: Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model. Ph.D. thesis, University of California, Berkeley (1993), aAI9431898

[8] Caspi, P.: The quasi-synchronous approach to distributed control systems. Tech. rep., CMA/009931, Verimag, CrysisProject "The Cooking Book" (2000)

[9] Caspi, P., Mazuet, C., Reynaud Paligot, N.: About the design of distributed control systems: The quasi-synchronous approach. In: SAFECOMP 2001, Budapest, Proceedings. LNCS, vol. 2187, pp. 215–226 (2001)

[10] Cohen, A., Duranton, M., Eisenbeis, C., Pagetti, C., Plateau, F., Pouzet, M.: N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. ACM SIGPLAN Notices **41**(1), 180–193 (2006)

[11] Dörflinger, A., Albers, M., Fiethe, B., Michalik, H., Möstl, M., Schlatow, J., Ernst, R.: Demonstrating controlled change for autonomous space vehicles. In: NASA/ESA Conf. on Adaptive Hardware and Systems, Colchester UK. pp. 95–102. IEEE (2019)

[12] Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation **205**(8), 1149–1172 (2007)

[13] Graf, S., Quinton, S., Girault, A., Gößler, G.: Building correct cyber-physical systems: Why we need a multiview contract theory. In: 23rd Int. Conf., FMICS 2018, Maynooth, Ireland. LNCS, vol. 11119, pp. 19–31 (2018)

[14] Halbwachs, N.: Synchronous Programming of Reactive Systems. Springer US (2013)

[15] Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. Proceedings of the IEEE **91**(1), 84–99 (2003)

[16] Kahn, G.: The semantics of a simple language for parallel programming. Information processing **74**, 471–475 (1974)

[17] Kloda, T., Bertout, A., Sorel, Y.: Latency upper bound for data chains of real-time periodic tasks. Journal of Systems Architecture p. 101824 (2020)

[18] Kopetz, H., Bauer, G.: The time-triggered architecture. Proc. of the IEEE **91**(1), 112–126 (2003)

[19] Krcál, P., Yi, W.: Communicating timed automata: the more synchronous, the more difficult to verify. In: Int. Conf. on Computer Aided Verification. pp. 249–262. LNCS (2006)

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

[21] Lee, E.A., Parks, T.M.: Dataflow process networks. Proceedings of the IEEE **83**(5), 773–801 (1995)

[22] Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM (JACM) **20**(1), 46–61 (1973)

[23] Natarajan, S., Broman, D.: Timed C: An extension to the C programming language for real-time systems. In: 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 227–239. IEEE (2018)

[24] Stigge, M.: Real-time workload models: Expressiveness vs. analysis efficiency. Ph.D. thesis, Acta Universitatis Upsaliensis (2014)

[25] Stigge, M., Ekberg, P., Guan, N., Yi, W.: The digraph real-time task model. In: 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium. pp. 71–80. IEEE (2011)

[26] Stigge, M., Yi, W.: Graph-based models for real-time workload: a survey. Real-time systems **51**(5), 602–636 (2015)

[27] Yi, W.: Towards customizable CPS: composability, efficiency and predictability. In: Duan, Z., Ong, L. (eds.) Formal Methods and Software Engineering - 19th Int. Conf. on Formal Engineering Methods, ICFEM 2017, Xi'an, China, Nov. 13-17, 2017. LNCS, vol. 10610, pp. 3–15 (2017)