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

Wang Yi[1], Morteza Mohaqeqi[1], and Susanne Graf[2]

[1] Uppsala University, Sweden
{wang.yi,morteza.mohaqeqi}@it.uu.se
[2] Univ Grenoble Alpes, CNRS, France
susanne.graf@imag.fr

Abstract. Inspired by the pioneering work of Gilles Kahn on concurrent systems, we model real-time systems as a network of software components each of which is specified to compute a collection of functions according to given timing constraints. The components communicate with each other and their environment via two types of channels: (1) FIFO queues for buffering data, and (2) Registers for sampling time-dependent data streams from sensors or output streams of other components executed at different rates. We present a fixed-point semantics for this model which shows that each system function of a network computes for a given set of input (timed) streams, a unique (timed) output stream. Thanks to the deterministic semantics, a model-based approach is enabled for not only building systems but also updating them after deployment, allowing model-in-the-loop simulation to verify the complete behaviour of the resulting system.

1 Motivation

Today, a large part of the functionality of Cyber-Physical 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 Cyber-Physical Systems will become open platforms. They will allow integration of an expanding number of software components over their life-time, e.g., in order to customize and enhance their functionality according to the varying needs of individual users. To enable this, we must design and build systems that allow for updates after deployment. Furthermore, it must be guaranteed that the resulting systems not only preserve the original as well as the extended functionality, but also stay safe after updates. Unfortunately, current design methodologies for real-time systems, in particular safety-critical systems, offer only limited or no support for modifications and extensions on systems after deployment without demanding re-designing the whole system.

This paper develops a semantic model for real-time systems that can be updated in a model-based and incremental manner. We model such systems as a network of real-time software components connected by communication channels in the style of Kahn Process Networks (KPN) [15], allowing asynchronous data exchange. We present a simple but expressive model, MIMOS, to formalize such abstract network models. In designing the model and its semantics, we adopt the following principles:

* This work is partially supported by the projects: ERC CUSTOMER and KAW UPDATE.

Determinism. In a model-based approach to software design and update, using deterministic model is crucial to ensure that any property validated on a system model holds also on the final implementation i.e. executable code generated from the model. For safety-critical systems, future software updates may have to be validated and deployed based on the system model modified according to the intended updates. For example, to update the software of a pacemaker in-operated in a human body, safety properties of the intended updates may be validated and tested in a model-based approach instead of testing directly on human body, which may risk human life.

Separation of functionality and timing. The model of a system should allow to specify and to reason about the system functions independently of their implementation that may be subject to timing and resource constraints. This gives the advantage that the functional correctness can be validated efficiently without taking into account the complex timing behavior of the implementation. When the system inputs (e.g., sampled data from sensors) are time-dependent, the system output is also time-dependent. In such cases, we need to reason about streams of reals (called here, time streams), representing time points at which sensor data are sampled or outputs are written. Time streams are simply another type of input and output streams for the system functions. In fact, time streams are generated by the system scheduler, and in turn used to sample the input and output streams computed by the system functions.

Separation of computation and communication. We assume non-blocking data exchange between system components, implemented by either asynchronous FIFO channels for buffering system inputs and outputs, or registers for storing sampled time-dependent data. This allows system components to be specified as *independent real-time tasks*, whose timing behaviours can be analyzed efficiently. In particular, the underlying schedulability analysis for deployment will be greatly simplified compared with the case for dependent real-time tasks. More importantly, it enables the component-wise construction of systems avoiding interference between the original system and newly integrated components in case of updates.

Updatability (avoidance of interference). The model of a system should allow for modifications by integrating new components to implement 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 [12]), 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.

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 [13], and the time-triggered paradigm promoted by Kopetz [17], 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, software updates of real-time systems after deployment have attracted increasing interest. A model-based approach to the design and updates for cyber-physical systems is proposed in [25]. The work of [10] 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 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 model named MIMOS to 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. The 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 over data streams such that each of them, for a given set of (timed) input streams, defines a unique (timed) output stream; furthermore (2) the network can be modified by integrating new components for adding new system functions or replacing the existing components with 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 [14] 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 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 [7] is presented in [5]. 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) [15]. In the literature, there have been various extensions to KPN. A special case of KPNs is dataflow process networks (DPN) [20]. 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 [9]. 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 [22]. 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 that of MIMOS. However, while Timed C is a general programming language without a 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 Network (KPN) [15]. 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 model to formalize system models. In this section, we present the main primitives and informal semantics of MIMOS. A fixed-point semantics is given in Section 4.

3.1 Preliminaries on KPN

This subsection reviews the original 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 has a set of inputs and a set of outputs, and it computes a set of functions, one for each output. For a set of input sequences, each of the functions defines a unique output sequence. A node may have memory which cannot be accesses by other nodes.*
- *Channels are of unbounded capacity. A channel connects exactly one writer node to exactly one reader node. However, multiple channels may be connected to single output.*
- **read** *from a channel is blocking, that is, a node can only execute if all its input channels contain enough data. This means that a node cannot test the emptiness of channels;* **write** *to a channel is non-blocking.* □

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

Example 1. Listing 1 shows a program representing an example KPN. Nodes are defined by `process` keyword. The procedure executed by a node is written in a `Repeat` block. The structure of this program is shown 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.

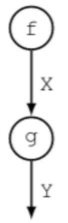

Fig. 1: Structure of the program in Listing 1.

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 obtained by computing node functions in an arbitrary order [15].

An essential property of KPNs is their *determinism*. It is guaranteed under any sufficiently fair scheduler, i.e., schedulers which do not postpone a process indefinitely.

Theorem 1 (Functional Determinism [15]). *Given a set of input streams (histories on input edges), the set of output streams computed by a KPN is uniquely defined.* □

Theorem 1 states 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 Kahn Process Networks

The order- and speed-independent functional determinism of KPN leads to a natural function preserving extension of KPN to timed versions of KPN which allow to represent real-time systems as KPN in which each node is executed according to some *release pattern* — that is, a sequence of time points — and a deadline for each release.

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

As *release pattern* one may choose any reoccurring real-time task model [24], such as periodic tasks [21], generalized multiframe [4], or DRT [23] and timed automata [11] as long as they are deterministic.

A TKPN is a KPN, that is each node computes a tuple of functions on streams, one for each output channel. One may generalize this model by allowing the definition of a different deadline for each output of a node.

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. At each release time point, if all needed inputs are available, the node computes and delivers the resulting outputs within the specified deadline. In order to achieve timing determinism, a node reads at the release time and delivers outputs at the given deadline. This read-execute-write approach is similar to the *implicit* communication model of AUTOSAR [1].

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, as nodes read and write at deterministically defined time points, it enjoys also the timing determinism. In order to formulate this properties, we need to consider *timed* event histories or streams, as the time-points at which the outputs are delivered depends on the time-points at which the inputs become available.

Theorem 2 (Functional and Timing Determinism of TKPN). *For any given set of timed input streams (histories of inputs values and the time-point at which the values are available), the set of timed output streams computed by a TKPN is uniquely defined.*

The result is established in Proposition 1 of Section 4. □

3.3 MIMOS: TKPN with Registers

In real-time applications, value streams produced by the environment may be time-dependent. In Cyber-Physical Systems, typical examples would be values from sensors capturing physical phenomena. The system usually does not need all values produced by a sensor but at any time, it would use the newest available 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 would typically lead to memory overflow. Or, a too slow sensor could block the system. Neither situation is desirable. In such cases, it is useful to have a communication channel which keeps always the most recently written value. We extend TKPN with such channels, called *register*.

Definition 3 (MIMOS: TKPN with registers). *TKPN extended by registers is TKPN where some channels can be a register instead of a FIFO. We call this extension of TKPN* MIMOS. □

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, and **read** from a register is non-blocking and reads the current value. When both **read** and **write** occur at the same time, the current value is updated by **write** before **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, and (2) guarantee absence of buffer overflow regardless of the speed at which these values are produced and read.

Theorem 3 (Functional and Timing Determinism of MIMOS). *For any given set of timed input streams (histories of inputs values and the time-point at which the values are available), the set of timed output streams computed by a TKPN with registers is uniquely defined.*

This result is established by Proposition 2 of Section 4. □

```
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;
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 MIMOS program.

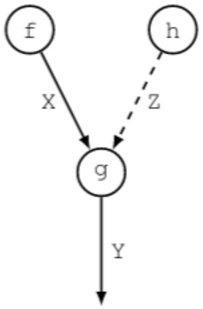

Fig. 2: The structure of the program in Listing 2. Dashed arrow indicates a *register*.

3.4 Design and Update with MIMOS

A model-based approach can be sketched as follows[3]. 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 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 satisfies 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.

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

4 Fixed-Point Semantics

In this section, we present a formal semantics for MIMOS. We use the notations introduced in [15] to prove the order and time independent determinism of KPNs of Definition 1. The notion of *timed stream* is introduced to define the semantics of TKPNs of Definition 2 and to show their timing determinism.

4.1 Preliminaries on KPN [15]

We recall the basic notations from [15]. The function **F** associated with each node of a KPN is represented as a function from a set of input streams to a set of output streams.

We now formally define streams and functions. We consider streams of elements from 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 finite and infinite sequences in $\mathbb{D}^\infty$. The domain of time streams $\mathbb{T}$ is the set of finite and infinite sequences of time points in $\mathbb{R}^\infty$ consisting of (not necessarily strictly) increasing time points. Infinite time streams must diverge.*

The domain of timed streams $\mathbb{S} \times \mathbb{T}$ *are finite and infinite sequences in* $(\mathbb{D} \times \mathbb{R})^\infty$.

Note that a tuple of streams (of the same length) can also be seen as a stream of tuples, and conversely. We alternate freely between these views. In particular, a timed stream may either be denoted as $\boldsymbol{S} \times \boldsymbol{T}$ for two appropriate streams of the same length or as a single stream of appropriate pairs.

Sometimes, we want to view a stream of $\mathbb{S}$ explicitly as a stream of finite "segments", such that a segment is now considered an "element". We denote by Σ the domain of finite segments *of $\mathbb{D}$, and by $\mathbb{S}^\Sigma$ the domain of streams of Σ.*[4]

We use $\sqsubseteq$ to stand for the standard prefix order on sequences, λ for the empty sequence, and "$\bullet$" for concatenation. Denote ϵ the "empty element", the neutral element for concatenation. □

Node functions **F** are built from the following basic functions mapping (tuples of) streams to (tuples of) streams.

Definition 5 (Functions on streams).

1. *Data transformations: functions $\mathbf{F}^{\mathbb{D}} : \mathbb{D}^n \mapsto \mathbb{D}^m$ applied to elements of streams. We call $\mathbf{F}$ the corresponding function lifted to streams:*
   $\mathbf{F}(a_1 \bullet \boldsymbol{S}_1,\ \dots\ a_k \bullet \boldsymbol{S}_n) = \mathbf{F}^{\mathbb{D}}(a_1,\ \dots\ a_n) \bullet \mathbf{F}(\boldsymbol{S}_1,\ \dots\ \boldsymbol{S}_n)$.
2. *Standard order preserving stream manipulating functions "first", "remainder" and "append" as in [15]:*
   - $\mathbf{first}(a \bullet \boldsymbol{S}) = a$ *(sometimes* **f** *for short).*
   - $\mathbf{R}(a \bullet \boldsymbol{S}) = \boldsymbol{S}$ *(skips the first element of a stream).*
   - $\mathbf{app}(\mathbf{ini}, \boldsymbol{S}) = \mathbf{ini} \bullet \boldsymbol{S}$ *(adds an initial element and pushes* $\boldsymbol{S}$ *to the right).* □

[4] Note that a stream in $\mathbb{S}^\Sigma$ is also a stream in $\mathbb{S}$ (for an appropriate Data domain).

Example 3 (Illustrating Example). Consider node g of Fig. 1 (Example 1) with input X and output Y. We present here the definitions of all output streams using the functions of Definition 5. Node g has a local variable *count* which gives rise to two streams: C_M, the previously stored values used as input of g, and C, the new value produced by g. We treat `threshold` as a parameter Th, a — possibly constant and infinite — stream. Function G associated with node g is a pair (G_Y, G_c), and the memory C_M is defined by function G_M. They are (recursively) defined by the following equations[5]

$$G_Y(X, C_M, Th) = G_Y^D(\mathbf{f}(X), \mathbf{f}(C_M), \mathbf{f}(Th)) \bullet G_Y(\mathbf{R}(X), \mathbf{R}(C_M), \mathbf{R}(Th))$$
$$\text{with } G_Y^D(x, c, th) = \text{if } (c+1 \geq th) \text{ then } 1 \text{ else } \epsilon$$
$$G_C(X, C_M, Th) = G_C^D(\mathbf{f}(X), \mathbf{f}(C_M), \mathbf{f}(Th)) \bullet G_C(\mathbf{R}(X), \mathbf{R}(C_M), \mathbf{R}(Th))$$
$$\text{with } G_C^D(x, c, th) = \text{if } (c+1 < th) \text{ then } (c+1) \text{ else } 0$$
$$G_M(C) = 0 \bullet C \text{ (initially 0, then } C \text{ “shifted to the right”)}$$

One may observe that node g applies a data transformation to the first elements of the inputs, produces an output element, or alternatively produces nothing[6], and then is applied recursively to the remainder of the input streams. Note also that here, the values of the output Y do not depend on input X; but X determines the length of Y. “Memories” such as C_M are defined by an initial element followed by the input stream which they “memorize” (this is the meaning of function G_M).

In the general case, a function may in each recursion step read zero or more elements from its input streams, and write zero or more elements to its outputs. That is, we can write any function $\mathbf{F} : \mathbb{S}^n \mapsto \mathbb{S}^m$ (other than the simpler memory functions) in the form:

$$\mathbf{F}(X) = \mathbf{Data_F}(\mathbf{Read_F}(X)) \bullet \mathbf{F}(\mathbf{Rem_F}(X)) \tag{1}$$

Thus, the functions $\mathbf{Read_F}$, $\mathbf{Data_F}$ and $\mathbf{Rem_F}$ fully characterize $\mathbf{F}$, where

- $\mathbf{Read_F} : \mathbb{S}^n \mapsto (\Sigma^n - \{\lambda\})$ extracts the (non-empty) initial input segments to be transformed by $\mathbf{Data_F}$
- $\mathbf{Data_F} : \Sigma^n \mapsto \Sigma^m$ the ”data transformation” or ”step” function of $\mathbf{F}$ (which in the general case transforms segments to segments) and defines the segments appended to the output streams.
- $\mathbf{Rem_F} : \mathbb{S}^n \mapsto \mathbb{S}^n$ defines which suffix to be considered for the recursive application of $\mathbf{F}$. Very often, we have $\mathbf{Read_F}(X) \bullet \mathbf{Rem_F}(X) = X$, that is exactly the inputs “read” by $\mathbf{Read_F}$ are “consumed” from the input streams at each “step”
- all these functions must be definable by basic functions of Def. 5.

The semantics of a KPN is defined by the union of the equation systems of its nodes. Kahn’s results [15] stating that such an equation system has a unique solution, that is, a KPN defines a function on streams, is formulated in Theorem 1.

[5] where for readability reasons, instead of notation **app**(a, X), we use its definition $a \bullet X$

[6] such as G_Y which produces element ”1” only if $c + 1 \geq th$ otherwise produces nothing, that is, ϵ

4.2 Semantics of Timed KPN

We define now the semantics of a timed node with a release pattern $P \in \mathbb{T}$ and a deadline δ. In order to do so, we show that we can extend a node function $\mathbf{F}$ on data streams (the node's semantic function) to a function $\mathbf{F}_\delta$ on timed streams, such that: (1) $\mathbf{F}_\delta = (\mathbf{F}, \mathbf{F}^T)$ defines a tuple of streams consisting of the data streams defined by $\mathbf{F}$, and the corresponding time streams defining the time points at which each data element is written into its destination FIFO[7]. (2) $\mathbf{F}_\delta$ is a Kahn function if time streams are interpreted as particular data streams. (3) The time extension expresses the intuition of release pattern P and the output delay δ of Def. 2. That is, the computation of F is divided into ”steps” defined by the activation pattern, where the necessary data are read at activation, and the result of computation written out at the deadline. We now state the proposition which in turn proves Theorem 2 of Section 3. The remainder of the subsection is dedicated to its proof.

Proposition 1. *The semantics of a TKPN is a function from Timed input streams to Timed output streams defined by a set of node functions* $\mathbf{F}_\delta$ *which are Kahn functions on Timed Streams and enjoy the three above mentioned properties.*

Proof: We consider for each node a fixed deadline δ[8], and we prove this proposition by constructing for any function $\mathbf{F} : \mathbb{S}^n \mapsto \mathbb{S}^m$ an appropriate function $\mathbf{F}_\delta : \mathbb{S}^n \times \mathbb{T}^{n+1} \mapsto \mathbb{S}^m \times \mathbb{T}^m$ of the form $(\mathbf{F}, \mathbf{F}^T)$, with $\mathbf{F}^T : \mathbb{S}^n \times \mathbb{T}^{n+1} \mapsto \mathbb{T}^m$.

Fig. 3 illustrates the structure of $\mathbf{F}_\delta$ which we explain while introducing the auxiliary functions f required for the definition of $\mathbf{F}^T$. Most functions f are presented in the form of Eq. 1, that is, defined by functions $\mathbf{Read}_f$, $\mathbf{Data}_f$ and $\mathbf{Rem}_f$, where here $\mathbf{Rem}_f(X)$ is always the function consuming $\mathbf{Read}_f(X)$ from X. If $\mathbf{Read}_f(X)$ is “trivial” (reads the first element of all its inputs), we may not mention it.

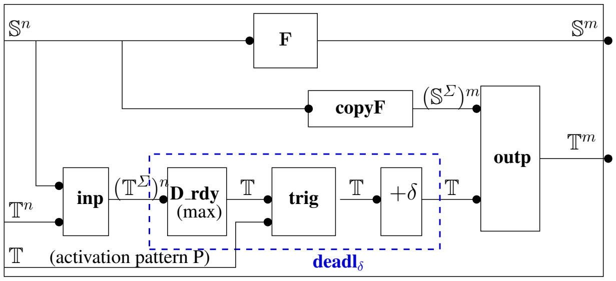

Fig. 3: Graphical representation of F_δ

1. Function **inp** reads the appropriate prefix from the timed input streams of $\mathbf{F}_\delta$ and outputs them in the form of a tuple of segments. If $\mathbf{Read}_\mathbf{F}$ is data independent, then $\mathbf{Read}_{inp} = \mathbf{Read}_\mathbf{F}$. Otherwise, it is a function that extends $\mathbf{Read}_\mathbf{F}$ by simultaneously reading the timed input streams in the same way, so that also in this case,

[7] in a corresponding TKPN implementation

[8] Making δ an input which may vary over time is straightforward

the tuple of segments read from the data streams and from the time streams have an identical structure (size). **Data**$_{inp}$ is simply the projection on the tuple of time segments.

2. The *deadline*, that is, the time point at which the output of **F** is written is calculated by a composition of three functions on time streams: the first one, **D_rdy**, produces the time points at which the inputs of **F** are ready (all required data are available), the second one, **trig**, uses the activation pattern P to produce the time points at which **F** is triggered, and the third one, $+\delta$, produces the time points at which the outputs of **F** are written into their destination streams:
3. Function **D_rdy** reads the first tuple of segments of its input time streams, and **Data**$_{D\_rdy}$ calculates the maximum of all available time points, that is, the latest date at which on of the corresponding data items has been written.
4. Function **trig** reads the first element from its input stream (produced by **D_rdy**), a time point t, and reads the smallest prefix $p_1 \bullet ... \bullet p_k$ of P such that $p_k \geq t$[9] because, according to Section 3.2, once all input data are available (time point t), the execution of **F** must be triggered at the earliest possible activation time point of the activation pattern (p_k). This function can be defined as follows:
   $\textbf{trig}(T, P) = \text{ if } (\textbf{f}(T) \leq \textbf{f}(P)) \text{ then } \textbf{f}(P) \bullet \textbf{trig}(\textbf{R}(T), \textbf{R}(P)) \text{ else } \textbf{trig}(T, \textbf{R}(P))$
5. Function $+\delta$ reads a time point from its input and adds δ to it. This is the desired time point of writing for all data items written by **F** at the corresponding "step".
6. Function **outp** makes sure that for each data item appended by **F** to one of the output streams at some step, the time point of writing is appended at the corresponding position of the time streams. Its first input comes from a modified version of **F** that presents the output of **F** explicitly as a tuple of segments[10]. **Read**$_{outp}$ reads such a segment tuple and a time point t (the time point of writing) from its second input. **Data**$_{outp}$ replaces every data element of this tuple by t.[11]

It is easy to see that all these functions can be written in terms of the basic functions of Def. 5, and therefore are Kahn functions. Finally, it satisfies also the third condition, as it clearly defines the intended meaning of activation pattern and deadline of Def. 2. This completes the proof. □

Example 4 (Illustrating example, continued). Consider again function g of Fig. 1. Now, g is executed with a periodic activation pattern `p`, and has a global output delay `dl` .

As the function(s) **F** on data remain untouched, we only need to add new equations defining $\mathbf{F}^T$, that is the time streams Y^T, C^T and C_M^T associated with Y, C and C_M. The situation here is a bit simpler than the general case: (1) we know that at each step exactly one item is read from each input, implying that function **Read**$_{inp} \equiv$ **first**, and (2) function G_Y may or may not output a data item, whereas G_C and G_M always produce one item. Therefore **Data**$_{outp}(x, c, th, t)$ = if $(c + 1 \geq th)$ then (t, t, t) else (ϵ, t, t)[12].

[9] which must exist

[10] it may use a strongly abstracted version of **F** which only preserves the structure of the output

[11] E.g., if **F** produces one element on each output stream, that is an m-tuple $(d_1 ... d_m)$, then it produces the m-tuple $(t ... t)$.

[12] G_M has the same deadline as G, meaning that the data put in the "memory" is immediately available

Furthermore, we suppose that the activation pattern is ”well chosen”, that is, at every activation time point there is indeed sufficient data available[13]. This allows to easily write the corresponding equations, where we abbreviate $\mathbf{deadl}_\delta(\mathbf{f}(X^T, C_M^T, Th^T), \mathbf{f}(P))$ by D, and $\mathbf{R}(X^T, C_M^T, Th^T, P)$ by **Rem**. Note that only Y^T depends on the data streams)

- $G_Y^T((X, C_M, Th), (X^T, C_M^T, Th^T, P))$ =
  [if $\mathbf{f}(C_M) + 1 \geq \mathbf{f}(Th)$ then D else ϵ] $\bullet$ $G_Y^T(\mathbf{Rem})$
- $G_C^T(X^T, C_M^T, Th^T, P) = D \bullet G_C^T(\mathbf{Rem})$
- $G_M^T(C^T) = 0 \bullet C^T$ □

4.3 Adding registers to Timed KPN

Finally, we provide the semantic underpinning for the full MIMOS model where some of the channels are registers. At the semantic level, a register is a node whose function transforms the timed stream produced by the source node into the timed stream read by the target node depending on a time stream representing the time points at which the register is read[14].

As the function of a register node is time dependent, we do not define time and function separately. The function **Reg** reads a value t from the trigger input, and the smallest prefix of the timed stream that contains a pair with a time value $> t$[15]. It appends to the output the last pair of this prefix with time value $\leq t$, and leaves for the recursive application the suffix including the element it has output. Therefore, the output stream defines for each trigger time point the (timed) value at the register input. We do not define the functions **Read**, **Data** and **Rem** for Reg, to avoid too much repetition but we define **Reg** directly[16].

Definition 6 (Stream transformation for a register). *Let be* $(\boldsymbol{S}, \boldsymbol{T})$ *a timed stream, and* $\mathsf{\mathbf{Tr}}$ *the time stream of trigger points. The function* **Reg** $: (\mathbb{S} \times \mathbb{T}) \times \mathbb{T} \mapsto \mathbb{S} \times \mathbb{T}$ *is defined by the following fixpoint equation, where* $\mathbf{2nd}(X)$ *is an abbreviation for* $\mathbf{f}(\mathbf{R}(X))$*:*

```
$\mathbf{Reg}((\boldsymbol{S}, \boldsymbol{T}),\ \mathsf{\mathbf{Tr}})$ =
    if $\mathbf{f}(\boldsymbol{S}, \boldsymbol{T}) = \mathbf{f}(\mathsf{\mathbf{Tr}})$ then
        if $\mathbf{2nd}(\boldsymbol{S}, \boldsymbol{T}) > \mathbf{f}(\mathsf{\mathbf{Tr}})$ then $\mathbf{f}(\boldsymbol{S}, \boldsymbol{T}) \bullet \mathbf{Reg}((\boldsymbol{S}, \boldsymbol{T}),\ \mathbf{R}(\mathsf{\mathbf{Tr}}))$ % output 1st
        else % that is $\mathbf{2nd}(\boldsymbol{S}, \boldsymbol{T}) = \mathbf{f}(\mathsf{\mathbf{Tr}})$
        $\mathbf{Reg}(\mathbf{R}(\boldsymbol{S}, \boldsymbol{T}),\ \mathsf{\mathbf{Tr}})$ % eliminate 1st
    else % that is $\mathbf{f}(\boldsymbol{S}, \boldsymbol{T}) < \mathbf{f}(\mathsf{\mathbf{Tr}})$
        if $\mathbf{2nd}(\boldsymbol{S}, \boldsymbol{T}) \leq \mathbf{f}(\mathsf{\mathbf{Tr}})$ then $\mathbf{Reg}(\mathbf{R}(\boldsymbol{S}, \boldsymbol{T}),\ \mathsf{\mathbf{Tr}})$ % eliminate 1st
        else % that is $\mathbf{2nd}(\boldsymbol{S}, \boldsymbol{T}) > \mathbf{f}(\mathsf{\mathbf{Tr}})$
        $\mathbf{f}(\boldsymbol{S}, \boldsymbol{T}) \bullet \mathbf{Reg}((\boldsymbol{S}, \boldsymbol{T}),\ \mathbf{R}(\mathsf{\mathbf{Tr}}))$ % output 1st
```

[13] In the general case, the equations are more complicated as a variable length segment is to be read from P as indicated by the definition of function **trig**

[14] which is the trigger time of the node reading the register

[15] which must exist if the stream is infinite. If it is finite, the definition of **Reg** is a bit more complicated, but it is easy to see that this can be fixed

[16] Note some similarity with the definition of function **trig**

Clearly, this function outputs a youngest pair $\leq$ the trigger time. If there are several pairs with the same date, the latest written is chosen. If initially the first value of **T** is 0, then it is guaranteed that $(\mathbf{S}, \mathbf{T})$ contains always an "old" pair, that is one with a time value $\leq$ the first value of $\mathbf{Tr}$. The reason is that the last output pair is reused in the next step. The definition gives also the expected result if the sequence of trigger time points is not strictly increasing — even if this usually is not expected.

If the register is rarely updated, and the date of the "next" data element is much larger than the "current", the same element may be read over and over again, according to the intuition of a register.

Note that a node function $\mathbf{F}^{\delta}$ could be ill defined in the case that $\mathbf{Read_F}$ depends on a value read from a register, because then there could be a circular dependency between the value read and trigger time point. We call a function **F** with register inputs *well defined* it is not ill defined in that sense.

We illustrate the effect of replacing a FIFO by a register input on $\mathbf{F}_{\delta}$ on hand of our running example which is clearly *well defined* because none of its **Read** functions is data dependent.

Example 5 (Illustrating example, continued). Again, consider function g from the previous example. Now, input Th is read from a register. This affects both time and data.

- As a register holds a valid data at any time, Th^T does not affect the trigger time point. Therefore, it is not anymore an input of the functions G^T. The new equations for G^T are obtained from those in Ex. 4 by eliminating Th^T from the inputs. This may allow the functions to be triggered earlier and more often.
- The value read from Th is now time dependent. The equations for G_Y and G_C are obtained from those of Ex. 3 by replacing Th by the register value read at the trigger time points, that is by $\mathbf{Reg}((Z, Z^T), \mathbf{trig}(\mathbf{D\_rdy}(\mathbf{f}(X^T, C_M^T)), P))$. This is a sequence of values obtained from Th by over- and under-sampling, depending on the activation pattern P.[17] □

We now formulate the proposition which guarantees Theorem 3. It is very similar to Prop. 1, but the function on data streams is not time independent anymore.

Proposition 2. *The semantics of a TKPN with* well defined *nodes including registers is a function from timed input streams to timed output streams defined by the set of (new) node functions* $\mathbf{F}_{\delta}$. *The function* **Reg** *associated with a register is a Kahn function and expresses well the informal semantics of Section 3.*

Proof: as **Reg** is defined using the functions of Definition 5, we stay within Kahn's framework. This guarantees that the entire network defines unique function. We have already argued that The functions **Reg** express well the intuition of a register as described Section 3. In particular, the fact that the value read at time t is always the last one written up to t (including t), reflects the "write over read precedence mentioned there. This completes the proof. □

Note that the semantics of the function reading a register can be expressed within the framework of Kahn. Nevertheless, as one can see from Definition 6, this is done

[17] in fact, due to our simplifying assumptions in Example 4, only oversampling

at the price of some ”look a head” into future time points, necessary to make sure to choose the “latest value written up to t”. It might be difficult to implement this semantics, in particular in a distributed setting or in the presence of even minimal jitter. We envisage two solutions for preserving determinism in this case. (1) approximation, that is determinism up to some ϵ[18], and (2) adapting the solution of [8], which consists in reading old enough data, that can be guaranteed to be present, despite possible jitter.

5 Analysis Problems

To enable that a MIMOS model can be compiled into a program executable on a platform with limited resources, the model should be verified to meet expected requirements. There are two principle ones: (1) the memory requirements must be bounded and (2) timing requirements on the system functions must be satisfied, in particular for safety-critical applications. In general, these verification problems are undecidable. However, with proper assumptions and restrictions, there are efficient solutions for practical purposes [18, 2]. Thanks to the determinism of MIMOS, verified properties on a MIMOS model will be preserved by the execution of code generated in compilation.

First, it is vital to know that the required memory never exceeds the available memory. In the execution of a MIMOS (program), the consumed memory depends on the buffer size required by the FIFOs, which can be specified as follows.

Definition 7 (Required buffer size (RBS)). *Assume that the data written to and read from a FIFO buffer are specified by timed streams $(a_1, t_1)(a_2, t_2)$... and $(a_1, t'_1)(a_2, t'_2)$..., respectively. Also, let $\omega(t) \equiv \max\{i|t_i \leq t\}$ and $\gamma(t) \equiv \max\{i|t'_i < t\}$. 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 can simultaneously exist in the queue. Indeed, computing RBS in a *process network* has been shown undecidable [6]. Despite this, the measure is computable for some subsets of KPN. 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) [19], the problem has efficient solutions [19]. 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 [9].

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 is not our current focus. Here, we just provide some initial observation.

A fairly direct consequence of Definition 7 is that the RBS of a FIFO in a MIMOS model is bounded if and only if there exists a constant c for which $\forall t \geq 0 : \omega(t) -$

[18] note that registers are used for reading continuous data streams

$\gamma(t) \le c$. 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, the system provides a response (or react) with a bounded delay. The response should be observable on an output channel. We define the end-to-end latency for each output channel of the system with respect to an influencing input.

Definition 8 (Worst-case end-to-end latency, e2e(i)). *Assume that the behavior of a* MIMOS *model is determined by a k-ary function* $\mathbf{F}$ *on streams. Let* $(I_1, \dots I_k)$ *be a set of (possibly infinite) external input timed streams for which* $\mathbf{F}(I_1, ...I_k) = (Y^D, Y^T)$. *Consider now, for some i, $1 \le i \le k$, 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* $\mathbf{F}(I_1, \dots, I'_i, \dots I_k) = (Y'^D, Y'^T)$. *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 with respect to the release patterns of the nodes in a MIMOS model. For instance, for a set of periodic tasks communicating through some registers, which can be viewed as a special case of MIMOS, the problem is explored in [16], where a worst-case analysis method with exponential time complexity is proposed. 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

The MIMOS model presented in this paper is to enable a model-based approach for not only systems design but also dynamic updates on systems after deployment (or in operation). It is deterministic: for a given set of (timed) input streams, the set of (timed) output streams determined by a MIMOS model are unique. The determinism allows that the complete behavior of the resulting system can be verified by simulation prior to the implementation and any intended update. Differently from the semantic model for the family of synchronous programming languages such as Lustre for real-time programming, MIMOS adopts asynchronous communications via FIFO channels and registers. MIMOS allows integration of new components on a system after deployment for new functions without re-designing the whole system or interfering with the existing system functionality. Additionally, existing components may be replaced also by new ones fulfilling given requirements.

As future work, MIMOS will be further developed to be a modelling and programming language for embedded systems design and update. A compiler will be developed to generate executable code from MIMOS models for both simulation and final implementation on a given target platform.

References

1. *AUTOSAR - Specification of RTE Software*, 2019.
2. Jakaria Abdullah, Gaoyang Dai, and Wang Yi. Worst-case cause-effect reaction latency in systems with non-blocking communication. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1625–1630, 2019.
3. AUTOSAR. AUTomotive Open System ARchitecture, https://www.autosar.org.
4. Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.
5. Guillaume Baudart. *A synchronous approach to quasi-periodic systems*. Phd dissertation, PSL Research University, March 2017.
6. Joseph Tobin Buck and Edward A. Lee. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California, Berkeley, 1993. AAI9431898.
7. Paul Caspi. The quasi-synchronous approach to distributed control systems. Technical report, Technical Report CMA/009931, Verimag, CrysisProject "The Cooking Book", 2000.
8. Paul Caspi, Christine Mazuet, and Natacha Reynaud Paligot. About the design of distributed control systems: The quasi-synchronous approach. In Udo Voges, editor, *Computer Safety, Reliability and Security, 20th International Conference, SAFECOMP 2001, Budapest, Hungary, September 26-28, 2001, Proceedings*, volume 2187 of *Lecture Notes in Computer Science*, pages 215–226. Springer, 2001.
9. Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. *ACM SIGPLAN Notices*, 41(1):180–193, 2006.
10. Alexander Dörflinger, Mark Albers, Björn Fiethe, Harald Michalik, Mischa Möstl, Johannes Schlatow, and Rolf Ernst. Demonstrating controlled change for autonomous space vehicles. In *NASA/ESA Conference on Adaptive Hardware and Systems, AHS, Colchester, UK, July 22-24*, pages 95–102. IEEE, 2019.
11. Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.
12. Susanne Graf, Sophie Quinton, Alain Girault, and Gregor Gößler. Building correct cyber-physical systems: Why we need a multiview contract theory. In Falk Howar and Jiri Barnat, editors, *Formal Methods for Industrial Critical Systems - 23rd International Conference, FMICS 2018, Maynooth, Ireland, September 3-4, 2018, Proceedings*, volume 11119 of *Lecture Notes in Computer Science*, pages 19–31. Springer, 2018.
13. Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Springer US, 2013.
14. Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.
15. Gilles Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.
16. Tomasz Kloda, Antoine Bertout, and Yves Sorel. Latency upper bound for data chains of real-time periodic tasks. *Journal of Systems Architecture*, page 101824, 2020.
17. Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.
18. Pavel Krcál and Wang Yi. Communicating timed automata: the more synchronous, the more difficult to verify. In *International Conference on Computer Aided Verification*, pages 249–262. Springer, 2006.
19. Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

20. Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.
21. Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.
22. Saranya Natarajan and David Broman. 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)*, pages 227–239. IEEE, 2018.
23. Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. The digraph real-time task model. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 71–80. IEEE, 2011.
24. Martin Stigge and Wang Yi. Graph-based models for real-time workload: a survey. *Real-time systems*, 51(5):602–636, 2015.
25. Wang Yi. Towards customizable CPS: composability, efficiency and predictability. In Zhenhua Duan and Luke Ong, editors, *Formal Methods and Software Engineering - 19th International Conference on Formal Engineering Methods, ICFEM 2017, Xi'an, China, November 13-17, 2017, Proceedings*, volume 10610 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2017.