Implementing Synchronous Models on Loosely Time Triggered Architectures

Stavros Tripakis, Claudio Pinello, Albert Benveniste, *Fellow*, *IEEE*,
Alberto Sangiovanni-Vincentelli, *Fellow*, *IEEE*, Paul Caspi, and Marco Di Natale, *Member*, *IEEE*

Abstract—Synchronous systems offer clean semantics and an easy verification path at the expense of often inefficient implementations. Capturing design specifications as synchronous models and then implementing the specifications in a less restrictive platform allows us to address a much larger design space. The key issue in this approach is maintaining semantic equivalence between the synchronous model and its implementation. We address this problem by showing how to map a synchronous model onto a loosely time-triggered architecture that is fairly straightforward to implement as it does not require global synchronization or blocking communication. We show how to maintain semantic equivalence between specification and implementation using an intermediate model (similar to a Kahn process network but with finite queues) that helps in defining the transformation. Performance of the semantic preserving implementation is studied for the general case and for a few special cases.

Index Terms—Synchronous models, distributed systems, loosely time-triggered architecture, semantics-preserving implementation, code generation, embedded systems.

1 Introduction

Many are the advantages of using synchronous models of computation to describe design functionality: among others, predictability, strong theoretical backing that allows formal verification of design properties, the existence of paths to implementation where timing and functional properties can be verified independently, and limited concurrency that prevents the explosion of possible behaviors of the design.

However, this cornucopia of positive aspects comes at a price: it is hard to implement synchrony, especially on distributed execution platforms. A solution for implementing synchronous models on distributed platforms is to use a clock synchronization protocol (e.g., see [1], [2]) to synchronize the clocks of the different execution nodes of the distributed architecture. This approach is followed by the Time Triggered Architecture [3]. Techniques for generating semantic-preserving implementations of synchronous models on TTA have been studied in [4]. However, this approach carries cost and timing penalties that may not be acceptable for some applications. In particular, TTA is not easily implementable for long wires (such as in systems where control intelligence is widely distributed) or for wireless communications and may require expensive resources (such as area and performance for hardware implementations, memory for software ones) utilization.

Hence, there has been growing interest in less constrained architectures, such as the Loosely Time-Triggered Architecture (LTTA) [5]. LTTA is characterized by a communication mechanism called *Communication by Sampling* (CbS). In this paper, we extend the LTTA concept to relax further the constraint of the architecture. In particular, we assume

- writings and readings are performed independently at all nodes connected to the medium using different local clocks and
- the communication medium behaves like a shared memory, i.e., values are sustained and are periodically refreshed, based on a local clock owned by the medium.

This architecture is very flexible and efficient as it does not require *any* clock synchronization and it is not blocking both for writes and reads. Consequently, risk of failure propagation throughout the distributed computing system is reduced and latency is also reduced, albeit at the price of increased jitter; see [6]. However, data can be lost due to overwrites that may occur because readers may be slower than writers and the memory is not enough to store all of the data that have been written but not read yet. If, as in safety critical applications that involve discrete control for operating modes or handling protection, data loss is not permitted, either the drift of the clocks of the nodes and of the medium can be kept small, as in earlier analysis of this architecture, and, consequently, fairly simple devices are enough to guarantee correctness of the operation, or more complex approaches are needed.

- *S. Tripakis and C. Pinello are with Cadence Research Laboratories, 2150 Shattuck Avenue, Berkeley, CA 94704. E-mail: {tripakis, pinello}@cadence.com.*
- *A. Benveniste is with IRISA/INRIA, Campus de Beaulieu, 35042 Rennes, France. E-mail: Albert.Benveniste@irisa.fr.*
- *A. Sangiovanni-Vincentelli is with the Electrical Engineering and Computer Science Department, University of California, Berkeley, Berkeley, CA 94720-1770. E-mail: alberto@eecs.berkeley.edu.*
- *P. Caspi is with VERIMAG/Université Joseph Fourier, Centre Equation, 2 avenue de Vignate, 38610 Gières, France. E-mail: caspi@imag.fr.*
- *M. Di Natale is with the Computer Engineering and Science Department, ReTiS Lab., Scuola Superiore S. Anna, Pisa, Italy. E-mail: marco@sssup.it.*

Manuscript received 14 June 2007; revised 21 Jan. 2008; accepted 26 Feb. 2008; published online 14 May 2008.
Recommended for acceptance by S.K. Shukla and J.-P. Talpin.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2007-06-0234.
Digital Object Identifier no. 10.1109/TC.2008.81.

In this paper, we address the more general situation where *no assumption is made about the relative speed of clocks*. The problem we address is how to map a simple synchronous functional description (albeit an extension to more general synchronous models is possible but not addressed in this paper) to an implementation based on LTTA so that semantics is not altered. In particular, we consider a semantic property to maintain in the mapping *stream equivalence preservation*. The problem is far from being simple because of the semantics "distance" between the two models. To solve it, we resort to the Platform-Based Design (PBD) (see, e.g., [7]) paradigm, where appropriate intermediate layers of abstraction are introduced to ease the transition from one model to the other. To this end, we introduce a layer of abstraction called FFP. FFP is essentially a Kahn Process Network (KPN), but with bounded FIFOs. We first show how to map the synchronous specification onto the FFP platform so that semantics is preserved and then how to map the FFP thus obtained into an LTTA so that semantics is preserved.

Our layered architecture consisting of LTTA and FFP has not been considered as such for distributed software real-time systems, hence the novelty of our approach. The authors are personally aware of cases in the aeronautics, nuclear, automation, and rail industries where the LTTA architecture with limited clock deviations has been used with success. The LTT bus based on CbS was first proposed in [5] and studied for a single writer-reader pair; Romberg and Bauer [8] proposes a variation of LTTA where some master-slave resynchronization of clocks is performed.

In [9], an FFP-like architecture is proposed and results on buffer sizing are given. How to maintain correct time-based semantics in CbS-based time-sensitive systems is studied in [10]. Finally, Kossentini and Caspi [11] develop an alternative approach, where up-sampling is used in combination with "thick" events as a way to preserve semantics. Approaches similar to our proposal have been studied in the area of hardware, for example, the so-called *latency insensitive* designs [12], [13] and the related *elastic circuits* [14].

We use Marked Directed Graphs (MDG) as tools to establish semantic preservation, as well as to obtain performance results on throughput and latency. Performance studies of marked graph models were carried out by a number of authors from whom we borrowed to build our method: [15] is a pioneering reference and [16] is the basic reference for the approach of performance evaluation using the max-plus algebra. The reader is referred to [13] for a number of references in the context of hardware design.

The paper is organized as follows: In Section 2, we describe our target application, namely, synchronous models. In Section 3, we introduce the target execution platform, LTTA. In Section 4, we describe the middle layer, FFP. In Section 5, we show how synchronous models can be implemented on FFP so that their semantics is preserved. In Section 6, we show how FFP can be implemented on top of LTTA. In Section 7, we offer bounds on throughput for the LTTA implementation and introduce a number of special cases where the constructions we use in mapping synchronous models to LTTA can be optimized. In Section 8, we obtain similar bounds and optimizations for latency. Section 9 concludes the paper and discusses future work.

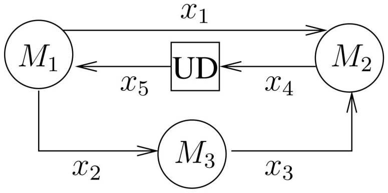


Fig. 1. A synchronous model with three machines.

2 Synchronous Models

For the sake of simplicity, we consider in this paper a basic single-clock synchronous model. Our results can be extended to more general synchronous models, in particular, synchronous languages such as Esterel, Lustre, or Signal [17].

A synchronous model consists of a set of communicating Mealy machines (a machine may be infinite state). Structurally, this can be represented as a directed graph, the nodes of which correspond to machines and the edges to communication links: If there is a link $M \rightarrow M'$, then an output of M is an input to M'. Since the composition of Mealy machines may not always be well-defined because of dependency cycles, we assume that every loop in the graph is "broken" by a *unit delay* (UD) element. A UD has an initial value which is also the value of its output at the initial instant. At every subsequent instant, the value of its output is equal to the value of its input in the previous instant.

An example of a synchronous model is shown in Fig. 1. There are three machines in this model, labeled M_1, M_2, and M_3. There is also a unit-delay element, depicted as a square block labeled UD. Notice that both feedback loops in this model are "broken" by the UD element.

We assume that the model has no *self-loops*, that is, there is no link $M_i \rightarrow M_i$ (not even a unit-delay link). This is not a restrictive assumption: Such links would need to be unit-delay links (by our initial assumption); therefore, they essentially correspond to part of the internal state of M_i. This internal state does not have to be exposed at the synchronous model level (it can be "hidden" inside M_i).

We define a partial order[1] $\prec$ on the set of machines in a synchronous model as follows: Given two machines M_i and M_j, if there is a link without UD from M_i to M_j, then $M_i \prec M_j$. We then complete $\prec$ with its reflexive and transitive closure. From the assumption that every loop in the synchronous model is broken by a UD, $\prec$ is indeed a partial order. M_i is a *minimal element* with respect to $\prec$ if there is no $M_j \neq M_i$ such that $M_j \prec M_i$.

For example, in the synchronous model shown in Fig. 1, we have $M_1 \prec M_3 \prec M_2$. In this case, the order is total and M_1 is its unique minimal element.

We give semantics to this model as follows: Every communication link is seen as an infinite *stream*, that is, an infinite sequence of values in some given domain. For simplicity, we will assume that all streams in the model take values in the same domain of values V. This is not a fundamental assumption and can be lifted at the expense of a more complicated presentation that we avoid. Under this assumption, every stream is a total function $x : \mathbb{N} \rightarrow V$, where $\mathbb{N}$ denotes the set of positive integers: $\mathbb{N} = \{1, 2, 3, \ldots\}$.

[1] That is, a reflexive, transitive, and antisymmetric binary relation.

In a synchronous model, each stream is the output of a unique machine. For instance, in the example shown in Fig. 1, x_1 is the output of M_1, x_3 is the output of M_3, and so on. Notice that we have included a separate stream x_5 as the output of the UD element.

Each machine M_i generally has an internal state, s_i, belonging in some set of states S_i. S_i may be infinite. S_i may also be a singleton (a set with a unique element), modeling a "memoryless" or "combinational" machine. The initial state of M_i is s_i^0. Then, M_i can be seen as a function $M_i : V^n \times S_i \to V^m \times S_i$, where n and m is the number of inputs and outputs of M_i, respectively. The function takes as input the values of the machine's input streams and state at time k and produces as output the values of the machine's output streams at time k, as well as the value of the machine's state at time $k+1$ ("next" state). For instance, for machine M_1 in Fig. 1, we have, for $k = 1, 2, \ldots$:

$$(x_1(k), x_2(k), s_1(k+1)) = M_1(x_5(k), s_1(k)).$$

The unit-delay element can also be modeled as a function similar to the one shown above. But, in this case, the internal state of the UD is always equal to its output at the same time k. Therefore, the function for a UD can be simplified to $y(k+1) = x(k)$, where x and y are, respectively, the input and output streams of the UD element. The output y is initialized to the initial value of the UD, say, v: $y(1) = v$. For instance, for the UD shown in Fig. 1, we have

$$\begin{aligned} x_5(1) &= v, \\ x_5(k+1) &= x_4(k), \quad \text{for } k = 1, 2, \ldots. \end{aligned}$$

As given above, the semantics of an entire synchronous model can be defined as a set of equations on streams. Under the assumption that every loop is broken by a UD element, this set of equations has a unique solution where every stream is defined at every time k. Assuming the functions M_i of each machine are computable, this solution can be computed up to any k by repeatedly "firing" the machines in a statically specified order. This order is any total order that respects the partial order $\prec$.

3 Loosely Time Triggered Architectures

An LTTA consists of a set of processing *nodes* (e.g., computers) communicating via a network. Each node runs a single program (or *process*): The program is executed quasi-periodically, triggered by the clock of the node. The network is assumed to implement the CbS communication paradigm, described in detail below.

An example is shown in Fig. 2. In this example, there are three nodes labeled as N_1, N_2, and N_3 and four CbS links. For simplicity, we only consider point-to-point links in this paper. Multicast links (delivering the same data to multiple consumers) do not add expressive power to the model. However, a multicast link can often be implemented in a more efficient way than multiple point-to-point links (e.g., by sharing some buffers).

3.1 The LTTA Communication API

The CbS links in an LTTA platform implement a set of communication services to its processes. The services can be described in terms of an application programming interface (API). The API is a set of functions that applications (i.e., processes) can call. Each of these functions has a set of *preconditions* describing the assumptions that must hold whenever the function is called and *postconditions* that describe the guarantees that the function provides when it returns. The CbS API contains the following functions:

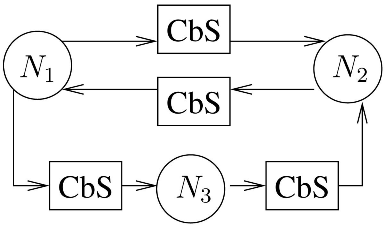


Fig. 2. An example LTTA.

- `write(z)`. It "writes" message `z` to the CbS buffer, overwriting any previous content, and returns nothing.
- `read()`. It returns the message currently stored in the CbS buffer.

CbS links are unidirectional: Only the source node can call the `write()` function and only the destination node can call the `read()` function. Messages are delivered after an unknown but bounded time in the order in which they are sent (we assume lossless in-order communication). The CbS API guarantees *atomicity* of function calls so that no data is corrupted when `write()` and `read()` are executed concurrently. Note, however, that no guarantees are provided regarding the "freshness" of data returned by `read()`. In particular, it may be the case that `write(z)` is called at time t and `read()` at time $t' > t$; however, `read()` does not return message `z` but an older message. This is due to the time it may take to complete `write()` and to transmit the message.

3.2 "Extended" CbS

To simplify our arguments and proofs, we extend the "pure" CbS model by adding a service on the reader side, `isNew()`, that returns the Boolean `true` if there is a new (unread) message in the buffer and `false` otherwise. Calls to `isNew()` do not change a message from "new" to "old," only a call to `read()` can do that. `isNew()` is not a "primitive" function. In the rest of the paper, we will refer to extended CbS simply as CbS. In fact, we can implement `isNew()` on top of pure CbS as follows: The writer process of every CbS channel will include to every message a *sequence number*, i.e., the value of a counter that is incremented after every `write()`. In principle, sequence numbers increase forever and no two messages have the same sequence number (we relax this assumption below so that only bounded counters are needed).

The reader process maintains a variable `lsn` that stores the sequence number of the last message received. This variable is updated at every `read()`. Let `sn` be the sequence number of the message that is currently in the CbS channel. Then, `isNew()` returns `false` if `sn` = `lsn` and `true` otherwise. That is, the message that is currently in the CbS channel is new if and only if its sequence number differs from the sequence number of the last message received.

Infinite counters do not exist in practice, so the sequence numbers attached to the messages by the writer need to be

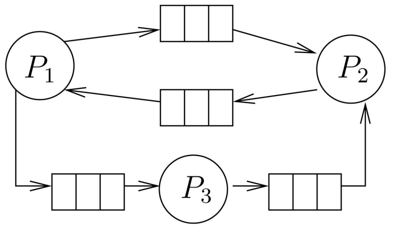


Fig. 3. An FFP with three processes and four queues.

finite. This can be achieved as follows: Suppose the writer uses a counter modulo N, that is, sequence numbers vary from 0 to $N-1$. The above implementation of `isNew()` is still correct provided that *there are at most* $N-1$ *consecutive writes between any two consecutive reads*. Indeed, this forbids the case where the sequence numbers "wrap around."

It is beyond the scope of this paper to show how CbS primitives can be implemented. For a general discussion, see [18], [19] and references therein.

3.3 Structure of LTTA Processes

Every LTTA process has a standard form:

```
initialize;
while (true) {
 await trigger;
 body;
}
```

The `initialize` section initializes the state of the process (if any) and possibly also some CbS buffers. The `body` section is executed at each trigger and is a nonblocking and terminating function that may update the state and call any of the CbS API functions in any order. Furthermore, we will assume that the execution of `body` will always terminate before the arrival of the next trigger (see the discussion on performance in Section 7). The trigger is an event defined externally (not inside the process). It can be periodic (or, more realistically, quasi-periodic), e.g., defined by a clock. It can also be an event associated to some interrupt.

4 Finite FIFO Platforms (FFPs)

This intermediate layer is needed to simplify the mapping of synchronous models on LTTA platforms. Our goal is to build the layer so that it has architectural similarities with LTTA and semantics that is fairly close to the one of synchronous models.

An FFP consists of a set of sequential processes communicating via directed, point-to-point, lossless, and FIFO queues of finite length. As such, an FFP is similar to a KPN [20], with the difference that, in a KPN, the queues are unbounded. Another difference is that, in an FFP, unlike in a KPN, processes do not *block*. Both reads and writes are nonblocking in an FFP and, as we shall see, the processes have the responsibility for checking that the queue is nonempty before doing a read and that the queue is nonfull before doing a write.

An FFP is shown in Fig. 3. In this case, there are three processes, P_1, P_2, and P_3, and four FIFO queues. The drawing of queues in the figure is only illustrative and does not imply anything about queue size. In an FFP, queues may have different sizes. Also note that some processes may have no input queues or no output queues.

4.1 The FFP Communication API

Each queue in an FFP provides the following communication API:

- `isEmpty()`: returns `true` if the queue is empty and `false` otherwise.
- `isFull()`: returns `true` if the queue is full and `false` otherwise.
- `put(z)`: appends message `z` at the end of the queue, provided that the queue is not full; if the queue is full, then the behavior is undefined.
- `get()`: if the queue is nonempty, removes the first message (i.e., the message at the head of the queue) and returns it to the caller; otherwise, behavior is undefined.

Notice that the `put()` and `get()` functions impose constraints on the environment (i.e., the caller), stated as preconditions above: Before calling `put()`, the environment must make sure the queue is not full and, before calling `get()`, it must make sure the queue is not empty. We will see below that the structure of the processes we consider guarantees these preconditions by construction.

As with LTTA, the communication links in an FFP are unidirectional: Only the source process can call the `isFull()` and `put()` functions and only the destination process can call the `isEmpty()` and `get()` functions. They also guarantee atomicity.

4.2 Structure of FFP Processes

An FFP process has the same form as an LTTA process (Section 3.3). Clearly, the `initialize` and `body` sections call functions of the FFP API instead of the CbS API. As with LTTA processes, we assume that the `body` section always terminates before the arrival of the next trigger.

5 Mapping Synchronous Models on FFP

Given a synchronous model, we build an FFP that is proven to preserve streams. Our results on the preservation of semantics borrow ideas from a number of previous works. The link between synchronous models and Kahn networks was studied in detail, for instance, in [21], for more general synchronous models. In [22], the use of skipping to synchronize distributed implementations with FIFO communication is studied. They focus on strongly connected networks of Moore machines. Therefore, each process belongs to a cycle and each link has a UD (and FIFO sizes are not discussed). Here, the semantic preservation result can be seen as a small generalization of the existing equivalence proof, but it is derived in a way that makes the proof of the main result of the paper easier to understand.

The main result of this section is the following:

Theorem 1. *Every synchronous model can be implemented in a semantics-preserving way on an FFP where queues have size at most 2.*

We provide a constructive proof of this theorem in Sections 5.1 and 5.2. To do so, we introduce a directed marked graph abstraction that is also useful to obtain the performance results later on. In this respect, the link with token-based actor models was discussed, e.g., in [23], [17]. The interest of the hardware community in asynchronous architectures has also led to a number of results in the same direction, e.g., see [24], [25].

In Section 5.3, we show that increasing the size of the queues does not impact correctness. In Section 5.4, we look at special cases of synchronous models which allow us to avoid the `isFull()` function and to simplify the construction, yielding a more efficient LTTA implementation.

5.1 From a Synchronous Model to an FFP

From each machine M_i of the synchronous model, we will construct a process P_i for the FFP. There will be a queue from P_i to P_j if and only if there is a link from M_i to M_j in the synchronous model:

- If this link has a no unit-delay element, then the queue will have size 1.
- If the link has a unit-delay element, then the queue will have size 2; queues that correspond to links with unit delays are called *UD queues*.

For instance, for the synchronous model shown in Fig. 1, the corresponding FFP is shown in Fig. 3. Note that the figure does not attempt to depict the size of the queues: According to the rule above, the queues from P_1 to P_2, from P_1 to P_3, and from P_3 to P_2 have size 1, whereas the queue from P_2 to P_1 has size 2. Notice that UD elements do not result in processes in the FFP.

Let M be a machine in the synchronous model with initial state s^0 and with n_i inputs and n_o outputs. Let P be the corresponding process in the FFP, with its input queues $Q_1, \ldots, Q_{n_i}$ and its output queues $R_1, \ldots, R_{n_o}$. Without loss of generality, assume that the first j outputs of M are connected to unit-delay elements with initial values $v_1, \ldots, v_j$, whereas the rest of the $n_o - j$ outputs are connected directly to other machines. Note that $0 \leq j \leq n_o$ and $n_i, n_o \geq 0$.

Then, the code for P is given as follows: The `initialize` part initializes the state of P to s^0 and its UD queues:

```
R_1.put(v_1);… ;R_j.put(v_j);
```

At each trigger, process P is ready to execute its `body`:

```
body {
  if (all input queues are nonempty and
       all output queues are nonfull) {
    read inputs from input queues;
    compute new state and outputs;
    write outputs to output queues;
  } /* else do nothing: skip this round */
}
```

However, this is only done if it can be guaranteed that the process can safely read its inputs and write its outputs, that is, if all of its input queues are nonempty and all of its output queues are nonfull. The `if` condition checks exactly this. The test can be implemented simply as follows:

```
not( Q_1.isEmpty() or … or Q_ni.isEmpty() )
and
not( R_1.isFull() or … or R_no.isFull() )
```

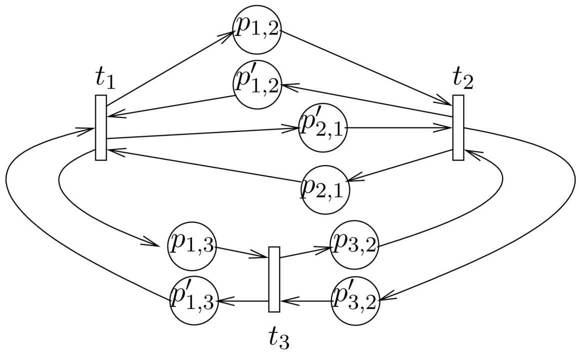


Fig. 4. Viewing the SFFP in Fig. 3 as a marked directed graph.

If the test fails, then the process does not execute any more statements in its `body`: We say that the process *skips* and goes back to the beginning of the while loop to wait for the next trigger. If the test succeeds, the process proceeds to execute the statements of its `body`: We say that the process *fires*. The `read inputs` section can be implemented (if there are inputs) simply as

```
in_1: = Q_1.get(); … ;in_ni: = Q_ni.get();
```

where `in_i` are variables local to the process.

Then, the process computes its outputs and updates its state. The computation part depends on function M itself, which we assumed to be computable. Suppose that, after this procedure, the outputs are stored in local variables `out_i`. Then, writing the outputs can be implemented simply as

```
R_1.put( out_1 ); … ;R_no.put( out_no );
```

An FFP process constructed as described in this section is called an SFFP process. The corresponding platform is called an SFFP.

5.2 Semantical Preservation

Preservation can be proven in two main steps. Due to space limitations, we present only a sketch of these steps below.

Step 1: An SFFP never deadlocks. That is, it is always possible for some process to fire (instead of skipping indefinitely). Indeed, an SFFP can be viewed as an *MDG* [26]. MDGs are a subclass of Petri nets [27] where each place has a unique input transition and a unique output transition. Fig. 4 illustrates how to translate an SFFP to an equivalent MDG. The figure shows the MDG obtained by the SFFP shown in Fig. 3.

The general translation procedure is given as follows: Every SFFP process P_i is mapped into an MDG transition t_i. Every SFFP queue from P_i to P_j is mapped into two MDG places: a *forward* place $p_{i,j}$ with input transition t_i and output transition t_j and a *backward* place $p'_{i,j}$ with input transition t_j and output transition t_i. Suppose the queue has size k. If it is not a UD queue, then k tokens are initially placed in $p'_{i,j}$ and no token in $p_{i,j}$. If it is a UD queue, then an initial token is placed in $p_{i,j}$ and $k-1$ tokens are placed in $p'_{i,j}$. Notice that, by construction, every backward place initially has at least one token. Backward places model the fact that a process does not write to a full queue. The queue from P_i to P_j is full if and only if $p'_{i,j}$ is empty, in which case transition t_i cannot fire. On the other hand, every time process P_j reads (i.e., transition t_j fires), a token is placed in $p'_{i,j}$ which marks that the queue is nonfull again.

This MDG is an abstract model that is equivalent to the SFFP as far as firing of processes is concerned. This means that every sequence of firings of processes in the SFFP can be mapped into a sequence of firings of the corresponding transitions in the MDG, and vice versa. Note that the MDG only represents the "atomic" firings of processes in the SFFP, where a process completes its body without interruption (i.e., interleavings) from other processes. SFFP will also have nonatomic executions. However, these can be mapped into equivalent atomic executions by reordering the statements that execute so that, as soon as a process is enabled, it fires atomically. This can always be done because firings of different processes are independent. Similar techniques of reducing redundant interleavings are used in the verification of concurrent systems [28] and will not be further detailed here. Also note that the MDG only represents the firings of processes and not their "skips." The latter do not affect the state of the queues thus do not influence functional semantics.[2]

It is a classic result in [26] that an MDG in which each directed circuit has a positive token count is *live*, meaning that every transition in this MDG can be fired infinitely often. This is indeed the case of an MDG obtained as described above. MDG circuits formed by $p_{i,j}$ and $p'_{i,j}$ places contain at least one token since every backward place contains at least one token initially. For the same reason, MDG circuits formed exclusively by backward places contain at least one token. Finally, MDG circuits formed exclusively by forward places correspond to loops in the SFFP: These loops contain at least one unit delay, so the corresponding circuit has at least one token in the forward place modeling the UD queue.

Step 2: Any execution of the MDG, and essentially also of the corresponding SFFP, *is one possible execution of the corresponding KPN* [20]. The latter is the same as the SFFP, except that its queues are unbounded. A classic result in [20] is that such a network is *determinate*, that is, it has a unique semantics which is independent of the order in which processes are fired. Since the MDG is live, the semantics is infinite, that is, every queue can be written infinitely often and thus can be seen as an infinite stream of values. Now, one possible firing order is the one where processes are fired repeatedly respecting the partial order $\prec$. This firing order obviously generates the same streams as in the synchronous model. Since the semantics is independent of the firing order, every other order must also generate the same streams; thus, the semantics is preserved.

5.3 Increasing the Size of Queues

Theorem 1 gives an upper bound on the amount of buffer space required to faithfully implement synchronous models on FFP. Sometimes, however, it is desirable to use longer FIFOs in order to increase the *throughput* of the system, that is, the rate at which processes execute without skipping, thus producing useful outputs. Indeed, this is a classical case of memory versus throughput trade-off, as is also typically found in pipelines.

Increasing the size of the FIFOs does not affect the preservation of the synchronous semantics discussed in Section 5.2. In particular, the MDG is still live since the number of tokens at each place may only increase. Thus, we can state the following generalization of Theorem 1.

Theorem 2. *Let $\mathcal{F}$ be an SFFP obtained from a synchronous model using the method of Section 5.1, except that the size of a non-UD queue may be ≥ 1 and the size of a UD queue may be ≥ 2. Then, $\mathcal{F}$ preserves the semantics of the synchronous model.*

5.4 Avoiding to Check Queue Fullness

An MDG is k-bounded if the number of tokens in any place cannot increase beyond k. Another classic result of that in [26] states that if every transition of an MDG belongs to a circuit with at most k initial tokens, then the MDG is k-bounded. Notice that the total number of tokens in a circuit is invariant.[3]

This means that the queues in a loop of an SFFP will never contain more tokens than the number of UD in that loop. Therefore, if the size of a queue is larger than that number, we do not need to check for fullness before writing into it. That is, processes can omit calls to function `isFull()`, resulting in more efficient implementations on top of LTTA (indeed, as we shall see in Section 6, the implementation of `isFull()` on top of LTTA requires additional communication). More formally,

Theorem 3. *Let $\mathcal{F}$ be an SFFP. Consider the FFP $\mathcal{F}'$ obtained from $\mathcal{F}$ by changing the size k_i of one or more queues Q_i belonging to loops in $\mathcal{F}$ and by omitting calls to* `isFull()` *for these queues. If, for all such queues in $\mathcal{F}'$, we have $k_i \geq U(Q_i)$, where $U(Q_i) = min_{L\in\mathcal{F}|Q_i\in L}U(L)$ and $U(L)$ is the number of UD queues on loop L, then $\mathcal{F}'$ preserves the semantics of the synchronous model.*

Proof. Clearly, $L \in \mathcal{F} \Leftrightarrow L \in \mathcal{F}'$ because we do not remove queues or processes. Given a queue Q_i belonging to some cycle, consider a loop $\bar{L}$ with $Q_i \in \bar{L}$ and for which $U(\bar{L}) = U(Q_i)$. By the above arguments [26], the queue Q_i cannot accumulate more than $U(\bar{L})$ tokens. Since the queue has size $k_i \geq U(\bar{L})$, the source process P_i can omit the call to `isFull()` for Q_i because it would return `true` only if an input queue is empty. □

The above condition is purely "structural" and does not depend on triggering patterns neither on real-time delays such as execution or communication delays. Another way to guarantee that calls to `isFull()` are not necessary is by devising sufficient conditions on triggers and delays. Notice that conditions on triggers alone are not enough since what is important is not only when a process is triggered but also the times at which it performs its reads and writes, as well as the times at which messages arrive at the destination. Conditions of this type are studied in [5], [29].

6 Implementation of FFP on Top of LTTA

Each FFP queue of size k can be implemented on top of the LTTA platform with k CbS channels (using an array notation: `dataCbS`$[0\ldots k-1]$) used as a circular buffer from the source to the destination process and one CbS channel (`backCbS`) from the destination back to the source. The FFP API can be implemented on top of the CbS API as follows:

[2] On the other hand, the number of skips does in principle affect performance, in particular throughput and latency. These performance metrics are studied in Sections 7 and 8.

[3] Although [26] speaks about 1-bounded nets (safe), the result holds for k-bounded nets as well: Since the total number of tokens in a circuit is invariant, it cannot exceed k.

The source process maintains a state variable wc to count the number of written values. This counter is initialized to zero and is updated by put(), which is implemented as follows:

```
put(msg) {
  dataCbS[(wc mod k)].write(msg);
  wc: = wc + 1
}
```

At every put(), the source process writes to the next CbS buffer.[4]

Similarly, the destination process maintains a state variable rc to count the fresh reads (this counter is also initialized to zero). Moreover, there is a state variable oldmsg to store the value of the latest returned message.

```
get() returns (msg) {
  ismsgNew: = dataCbS[(rc mod k)].isNew();
  if (ismsgNew) {
    msg: = dataCbS[(rc mod k)].read();
    rc: = rc + 1
    backCbS.write(rc);
    oldmsg: = msg;
  } else
    msg: = oldmsg;
  return (msg);
}
```

When a new message is returned by the get() operation, we update the counter rc and we send it back to the source process, overwriting the previous value of its local reads counter. Repeated get() calls will keep returning the previous value until a fresh value is stored in position rc mod k by the remote put().

isFull is implemented as follows:

```
isFull() {
  lrc: = backCbS.read();
  return (wc-lrc> = k);
}
```

lrc denotes the local reads counter used by the source process. When the number of writes exceeds the acknowledged number of reads by at least k, the FIFO is deemed full. Notice that the source relies on the destination to provide a fresh value of the reads counter. Since we do not make any assumption on the network delay, the value sent by the destination process may take time to reach the source. Therefore, the source may have an out-of-date value of lrc with respect to the current rc. However, it is always the case that lrc $\leq$ rc. As a result, the estimated number of messages in the queue, wc-lrc, is no smaller than the true number of messages, wc-rc. That is, it is a conservative approximation which ensures that the source process will never attempt to write at a queue that is full.

Counters wc, rc, and lrc can be bounded: They can be modulo $2k$. The test in isFull() then becomes

```
return ( (wc-lrc) mod 2} * k > = k);
```

and increments become xc = (xc + 1) mod 2} * k.

isEmpty is implemented as follows:

```
isEmpty() {
  return not (dataCbS[(rc mod k].isNew());
}
```

If the message at the current read position is not new, then we can conclude that the FIFO is empty. Conversely, if the FIFO is empty, rc is equal to wc and the next new value will be written in position rc mod k.

Notice that each of the FFP API operations is implemented by a finite number of LTTA operations and each LTTA operation is nonblocking.

7 Throughput

In this and the following section, we will study the performance of our SFFP implementations, in terms of throughput and latency. These two metrics obviously depend on clock triggers and also on various types of execution and communication delays in the SFFP. In this section, as well as in the section that follows, we will assume that such delays are negligible. Notice that this assumption is not required for semantical preservation and, indeed, the method presented in Section 5 is correct for any delays. We discuss nonnegligible delays in Section 7.3.

Given this assumption, if all clocks run at exactly the same rate λ and their relative phases can be controlled so as to satisfy the partial order $\prec$, we can fire all processes at each trigger, thus achieving throughput λ. This corresponds to the best behavior. However, the LTTA platform cannot guarantee exact synchronization and phases may eventually drift to a point where triggers violate the $\prec$ order, thus introducing some amount of skipping. Clearly, processes cannot fire more often than their trigger.

In this section, we study worst-case throughput and show how it can be computed for any SFFP. We also show how it can be derived analytically for some special topologies.

7.1 Computing Worst Case Throughput

7.1.1 Real-Time Throughput

Let us first define a notion of "real-time" throughput in an SFFP. We capture triggering events of a process (e.g., the ticks of its clock) as a sequence of real-time stamps. One possible triggering sequence for a process P_i can be represented as an increasing function $c_i : \mathbb{N} \rightarrow \mathbb{R}$, where $c_i(n)$ is the time that P_i is triggered for the $n\mathrm{th}$ time. We assume that the series $c_i(n)$ is unbounded. The clock (i.e., trigger generator) of a process is generally "nondeterministic," that is, it may have more than one behavior: This means that there will be more than one c_i function associated to P_i. Let C_i be the set of such functions. Let $C = \times_i C_i$ be the Cartesian product of C_i. An element $c \in C$ is a vector of c_i.

Given $t \in \mathbb{R}$, we write $\hat{c}_i(t)$ to denote the value of c_i at time t, interpreting the clock as a counter. That is, $\hat{c}_i(t) = \max\{n | c_i(n) \leq t\}$. If the maximum is taken over an empty set, then it is zero, that is, $\hat{c}_i(t) = 0$ if the clock has not ticked at all until time t.

An element $c \in C$ determines an order of execution of all processes in an SFFP $\mathcal{F}$. Assuming that the worst-case execution time of the body of every process is smaller than the minimum interarrival time of successive ticks of its clock, the process is always ready to execute when its clock

[4] The modulo operator mod returns the remainder of the integer division.

ticks. Then, the process may fire or it may skip.[5] For $t \in \mathbb{R}$, let $fireno_{\mathcal{F},c,P_i}(t)$ denote the number of times process P_i fires until (and including) time t and let $skipno_{\mathcal{F},c,P_i}(t)$ denote the number of times it skips. Obviously, for any $\mathcal{F}$, $c \in C$, P_i, and t, we have

$$fireno_{\mathcal{F},c,P_i}(t) + skipno_{\mathcal{F},c,P_i}(t) = \hat{c}_i(t). \qquad (1)$$

Definition 1. *The real-time throughput of process P_i in an SFFP $\mathcal{F}$ with respect to a clock vector c is defined as:*

$$\lambda^{rt}(\mathcal{F}, P_i, c) = \lim_{t\to\infty} \frac{fireno_{\mathcal{F},c,P_i}(t)}{t}.$$

Theorem 4. *Let $\mathcal{F}$ be an SFFP and $\mathcal{F}'$ be an SFFP derived from $\mathcal{F}$ by increasing the size of one queue in $\mathcal{F}$ by 1. Then,*

$$\forall P_i, c, t, fireno_{\mathcal{F},c,P_i}(t) \leq fireno_{\mathcal{F}',c,P_i}(t).$$

Proof. For brevity, we use the notation $n_i(t)$ and $n'_i(t)$ instead of $fireno_{\mathcal{F},c,P_i}(t)$ and $fireno_{\mathcal{F}',c,P_i}(t)$, respectively. The clock c is implicit in this notation. Also, for $t \in \mathbb{R}$, we write t^- for a $t' < t$ such that no clock is triggered between t' and t (in c). Such a t' always exists. The proof is by induction on the total number of triggers up to time t. The result trivially holds when no triggers occur until t. Suppose the result holds when the number of triggers is n. We prove it also holds for $n+1$. We assume, without loss of generality, that the $(n+1)\mathrm{th}$ trigger happens at time t and P_i is the process that is triggered.

We reason by contradiction. Suppose the result does not hold. This means that 1) $n_i(t^-) = n'_i(t^-)$ (by the induction hypothesis) and 2) P_i fires at t in $\mathcal{F}$ but skips at t in $\mathcal{F}'$. There are two cases:

Case 1. P_i skips because some of its input queues, say, one written by process P_j, is empty in $\mathcal{F}'$ at time t. That is, $n'_i(t^-) = n'_j(t^-) + u$, where u is the number of initial tokens in this queue. From 1), we get $n_i(t^-) = n'_j(t^-) + u$. From the induction hypothesis, we have $n_j(t^-) \leq n'_j(t^-)$; thus we get $n_i(t^-) \geq n_j(t^-) + u$. This means that the queue would also be empty at time t in $\mathcal{F}$, which contradicts the assumption that P_i fires at t in $\mathcal{F}$.

Case 2. P_i skips because some of its output queues, say, one read by process P_j, are full in $\mathcal{F}'$ at time t. That is, $n'_i(t^-) + u = n'_j(t^-) + k'$, where k' is the size of this output queue in $\mathcal{F}'$. Again, from 1) and the induction hypothesis, we get $n_i(t^-) + u \geq n_j(t^-) + k'$. Since $k' \geq k$, where k is the size of this queue in $\mathcal{F}$, we get $n_i(t^-) + u \geq n_j(t^-) + k$. This means that the queue would also be full at time t in $\mathcal{F}$, which contradicts the assumption that P_i fires at t in $\mathcal{F}$. □

Corollary 1. *Increasing the size of queues in an SFFP does not decrease throughput.*

7.1.2 Logical-Time Throughput

We would also like to define a notion of "logical-time" throughput. This allows us to make some general claims about the throughput of an SFFP without making assumptions about the real-time rates of the clocks, that is, without having c as a parameter. Moreover, we will see that this logical-time throughput essentially characterizes real-time throughput modulo a scaling factor (Theorem 9).

Consider a clock χ, also represented by an increasing function $\chi : \mathbb{N} \to \mathbb{R}$.

Definition 2. *The logical-time throughput of process P_i in an SFFP $\mathcal{F}$ with respect to a clock vector c and a clock χ is*

$$\lambda^{lt}(\mathcal{F}, P_i, c, \chi) = \lim_{n\to\infty} \frac{fireno_{\mathcal{F},c,P_i}(\chi(n))}{n}.$$

We want to compute the worst-case logical-time throughput, but we cannot define "worst-case" by taking the minimum of $\lambda^{lt}(\mathcal{F}, P_i, c, \chi)$ over all possible c: This is trivially 0 since we can always have a clock c_i, which is increasingly "slow" with respect to χ. To avoid this problem, we introduce the notion of a *slow* clock χ. We say that χ is *slower* than c if, between any two consecutive ticks of χ, there is at least one tick of any clock c_i.[6] Formally,

$$\forall i, \forall n \in \mathbb{N}, \exists k \in \mathbb{N}, c_i(k) \in [\chi(n), \chi(n+1)). \qquad (2)$$

We denote by $C(\chi)$ the set of all c such that χ is slower than c. We can now define the worst-case logical throughput.

Definition 3. *The worst-case logical-time throughput of process P_i in an SFFP $\mathcal{F}$ with respect to a clock χ is*

$$\lambda^{wclt}(\mathcal{F}, P_i, \chi) = \inf_{c\in C(\chi)} \lambda^{lt}(\mathcal{F}, P_i, c, \chi).$$

We now state a result that is fundamental for what will follow. Let $C_{wc}(\chi)$ be a subset of $C(\chi)$ such that, for all $c \in C_{wc}(\chi)$ and for any $n \in \mathbb{N}$, each clock c_i ticks exactly once in the interval $[\chi(n), \chi(n+1))$ and the order of clock ticks in c satisfies the following property: If process P_i is enabled at time $\chi(n)$ and process P_j is disabled at that time, then the clock of P_i ticks after the clock of P_j in the above interval. We call this the "slow" triggering policy.

Lemma 1. *For $c \in C_{wc}(\chi)$, if P_i is not enabled at time $\chi(n)$, then it will not fire in the interval $[\chi(n), \chi(n+1))$.*

Proof. If not, then, from the fact that it fires at time t_i, some other process must have fired some time in the interval $[\chi(n), t_i)$, enabling P_i. Let P_j be the first process to fire in $[\chi(n), t_i)$, then, since no other process fired before, P_j was enabled at time $\chi(n)$ in c. But, this means an enabled process P_j is triggered before a disabled process P_i, which contradicts the definition of $C_{wc}(\chi)$. □

Theorem 5. *For any SFFP $\mathcal{F}$ and any process P_i of $\mathcal{F}$:*

$$\forall c \in C(\chi), \forall c' \in C_{wc}(\chi), \forall n \in \mathbb{N},$$
$$fireno_{\mathcal{F},c,P_i}(\chi(n)) \geq fireno_{\mathcal{F},c',P_i}(\chi(n)).$$

Proof. For brevity, we use $n_i(t)$ and $n'_i(t)$ instead of $fireno_{\mathcal{F},c,P_i}(t)$ and $fireno_{\mathcal{F},c',P_i}(t)$, respectively. Suppose the result does not hold. Let k be the first n for which the inequality is violated. Let P_i be the process for which it is violated. Let $t = \chi(k)$ and $s = \chi(k-1)$ (if $k = 1$, then let $s = 0$). By definition of $C_{wc}(\chi)$, P_i is triggered exactly once in the interval $[s, t)$ in c', say, at time t_i. We have

[5] Two or more processes may be triggered at the same time. All of them that are enabled will then fire.

[6] Such a clock is called "sluggish" in [22].

$n_i(t) < n'_i(t)$, but $n_i(s) \geq n'_i(s)$. This implies that $n'_i(s) = n_i(s)$ and P_i fires at time t_i in c'. This implies that $n'_i(s) = n_i(s)$ and P_i fires at time $t_i \leq t$ in c'. By Lemma 1, this in turn implies that P_i is enabled at time s in c'.

We claim that P_i is also enabled at time s in c. Indeed, from the fact $n'_i(s) = n_i(s)$, P_i has fired the same number of times in c' as in c. Also, since t is the first time the result is violated, for any process P_j that either "feeds" data into an input queue or consumes data from an output queue of P_i, we have $n_j(s) \geq n'_j(s)$. These two facts imply that, for any queue Q of P_i, if Q is a nonempty input queue (respectively, nonfull output queue) at time s in c', then it is also nonempty (respectively, nonfull) at time s in c. This implies P_i is enabled at time s in c.

From the assumption $c \in C(\chi)$, P_i is triggered at least once in the interval $[s, t)$ in c. A process that is enabled can only be disabled by firing itself; therefore, P_i fires at least once in the interval $[s, t)$ in c. This means $n_i(t) \geq n'_i(t)$, which contradicts our hypothesis. □

7.1.3 Computing Worst-Case Logical-Time Throughput

Theorem 5 effectively says that the worst-case logical-time throughput is achieved by the sequence of firings captured by $C_{wc}(\chi)$: At every period between two successive ticks of χ, we first trigger the processes that are disabled in the beginning of this period and then the processes that are enabled. We can use this result to compute the worst-case throughput of an SFFP by analyzing its MDG.

Given an MDG, a *marking* m is a vector of integers giving the number of tokens stored at each place. Starting from a marking m, if a transition t_i fires, then the MDG *reaches* a new marking m' where exactly one token is removed from each of the input places of t_i and exactly one token is added to each of its output places.

Let $\mathcal{R}(\text{MDG}) = (\mathcal{M}, E)$ denote the *reachability graph* of MDG, where each vertex $m \in \mathcal{M}$ is a marking and each edge $e = (m, m') \in E \subseteq \mathcal{M} \times \mathcal{M}$ is annotated with a set $\mathcal{T}_e$ of transitions. $\mathcal{R}(\text{MDG})$ is the *minimal* graph such that we have the following:

- $\mathcal{M}$ contains the initial marking m^0.
- If $m \in \mathcal{M}$ and there exists a set $\mathcal{T}$ of transitions enabled at m and let m' be the marking reached by firing all transitions in $\mathcal{T}$, then $m' \in \mathcal{M}$ and edge $e = (m, m') \in E$, with annotation $\mathcal{T}_e = \mathcal{T}$.

The initial marking m^0 corresponds to the initial set of tokens, as described in Section 5.2. Note that, since MDGs are conflict-free, transitions in $\mathcal{T}_e$ can be fired in any order. Since the MDG is k-bounded, for some k, $\mathcal{R}(\text{MDG})$ is finite.

Given an SFFP $\mathcal{F}$ and its associated MDG, we define for each process $P_i \in \mathcal{F}$ a weighting function $w^i : E \rightarrow \{0, 1\}$ that associates to each edge $e \in E$ of $\mathcal{R}(\text{MDG})$ the weight w^i_e such that $w^i_e = 1$ if the transition t_i associated to P_i belongs to $\mathcal{T}_e$; otherwise, $w^i_e = 0$.

An edge $e = (m, m') \in E$ is called *maximal* if $\mathcal{T}_e$ contains *all* of the transitions that are enabled at m.

Definition 4. *Given an SFFP $\mathcal{F}$ and its associated MDG, the maximally concurrent reachability graph of $\mathcal{F}$, denoted as $\mathcal{R}_S(\mathcal{F})$, is the subgraph obtained from $\mathcal{R}(\mathcal{F})$ by removing all nonmaximal edges.*

By Lemma 1, $\mathcal{R}_S(\mathcal{F})$ implements the "slow" triggering policy. $\mathcal{R}_S(\mathcal{F})$ is deterministic in the sense that, from each marking $m \in \mathcal{M}$, there is exactly one outgoing edge in $\mathcal{R}_S(\mathcal{F})$. This and the fact that $\mathcal{R}_S(\mathcal{F})$ is finite implies that every infinite path in $\mathcal{R}_S(\mathcal{F})$ is a *lasso*, i.e., a finite path ending with a cycle. Also, since $\mathcal{R}_S(\mathcal{F})$ is deadlock-free, every finite path can be extended to an infinite path, i.e., to a lasso. In general, starting from different markings may lead to lassos with different cycles.

Let m be the initial marking corresponding to the initial conditions of $\mathcal{F}$. Consider the lasso of $\mathcal{R}_S(\mathcal{F})$ starting at m. Let $\ell_{\mathcal{F}}$ be the cycle of this lasso. Let $|\ell_{\mathcal{F}}|$ denote the length of $\ell_{\mathcal{F}}$ and let $w^i(\ell_{\mathcal{F}}) = \sum_{e \in \ell_{\mathcal{F}}} w^i_e$ be the sum of weights of all edges in $\ell_{\mathcal{F}}$. Define

$$\lambda^*(\mathcal{F}, P_i) = \frac{w^i(\ell_{\mathcal{F}})}{|\ell_{\mathcal{F}}|}.$$

Note that $\lambda^* \in [0, 1]$. The following theorem states that the worst-case logical-time throughput is independent of χ and can be computed as λ^*.

Theorem 6. *For any SFFP $\mathcal{F}$ and any process P_i of $\mathcal{F}$:*

$$\forall \chi, \lambda^{wclt}(\mathcal{F}, P_i, \chi) = \lambda^*(\mathcal{F}, P_i).$$

Proof. By the definitions of $\mathcal{R}_S(\mathcal{F})$ and $C_{wc}(\chi)$, for any $c \in C_{wc}(\chi)$, the markings at times $\chi(n)$ visit in $\mathcal{R}_S(\mathcal{F})$ the lasso starting from m and ending in $\ell_{\mathcal{F}}$. Then, by Theorem 5 and Definition 3, $\forall \chi, \lambda^{wclt}(\mathcal{F}, P_i, \chi) \geq \lambda^*(\mathcal{F}, P_i)$ and, since $C_{wc}(\chi) \subset C(\chi)$, equality holds. □

We next show that, in a connected SFFP $\mathcal{F}$, all processes have the same worst-case throughput, denoted as $\lambda^*(\mathcal{F})$.

Theorem 7. *Given a connected SFFP network $\mathcal{F}$ and any cycle ℓ of $\mathcal{R}_S(\mathcal{F})$:*

$$\forall P_i, P_j \in \mathcal{F}, \; w^i(\ell) = w^j(\ell).$$

Proof. First, consider the case where P_i and P_j are adjacent in $\mathcal{F}$ and $P_i \prec P_j$. By contradiction, assume that $w^i(\ell) \neq w^j(\ell)$, then, at each repetition of cycle ℓ, process P_i would be fired either more or fewer times than process P_j. Then, the number of tokens in the adjacent place would either increase or (respectively) decrease. This contradicts the definition of a cycle in $\mathcal{R}_S(\mathcal{F})$ because the marking at the beginning and end of the cycle must be the same. By the connectedness of $\mathcal{F}$, there exists an undirected path between processes P_i and P_j. By the previous argument, any two adjacent processes P and P' on this path must fire the same number of times along ℓ. □

The following is a corollary of the above result.

Theorem 8. *Given a connected SFFP $\mathcal{F}$,*

$$\forall P_i, P_j \in \mathcal{F}, \; \lambda^*(\mathcal{F}, \; P_i) = \lambda^*(\mathcal{F}, P_j).$$

Proof. The proof is obtained by applying the following lemma to cycle $\ell_{\mathcal{F}}$. □

We have implemented the algorithms to compute the worst case logical-time throughput and latency (see Section 8) in a prototype tool. The tool is written in Haskell (www.haskell.org) and can also generate Petri net reachability graphs and output these graphs in the Dot format (www.graphviz.org). The Dot tool can then be used to do

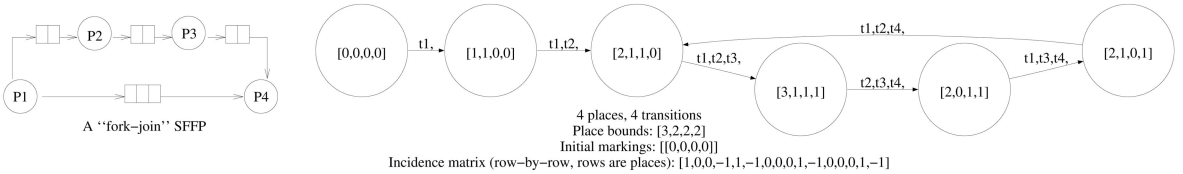


Fig. 5. The lasso of a "fork-join" SFFP with four processes. Only maximal edges are shown.

automatic layout and produce pictures such as the one shown in Fig. 5 (right). This picture was generated for an SFFP with a "fork-join" topology and four processes in total (Fig. 5 (left)). For this example, the tool computes $\lambda^* = \frac{3}{4}$ by counting the number of times that a process fires in the cycle of the $\mathcal{R}_S$ reachability graph shown in the figure.

7.1.4 Linking Real-Time and Logical-Time Throughput

The following theorem states that if every process has a clock that ticks at a "rate" no smaller than $\frac{1}{\Delta}$, then the real-time throughput is no less than the logical-time throughput scaled by $\frac{1}{\Delta}$.

Theorem 9. *Let $\mathcal{F}$ be an SFFP. Let Δ be any positive real number. Let c be a vector of clocks such that $\forall i, \forall n, c_i(n+1) - c_i(n) \leq \Delta$. Then, for any process P_i of $\mathcal{F}$:*

$$\lambda^{rt}(\mathcal{F}, P_i, c) \geq \frac{\lambda^*(\mathcal{F}, P_i)}{\Delta}.$$

Proof. Let χ be the perfectly periodic clock with period Δ, i.e., $\chi(n) = n\Delta$. By Theorem 5, for $c' \in C_{wc}(\chi)$, we have $fireno_{\mathcal{F},c,P_i}(n\Delta) \geq fireno_{\mathcal{F},c',P_i}(n\Delta)$; thus, $\lambda^{rt}(\mathcal{F}, P_i, c) \geq \frac{1}{\Delta} \cdot \lambda^{lt}(\mathcal{F}, P_i, c', \chi)$. The result follows by Theorem 6. □

7.2 Logical-Time Throughput for Special Topologies

7.2.1 Chains

Consider an SFFP $\mathcal{C}$ consisting of a chain of N distinct processes $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_N$, with a single-place queue between P_i and P_{i+1}, for $i = 1, \ldots, N-1$. The MDG associated to chain $\mathcal{C}$ has $2(N-1)$ places. Let the marking be ordered so that the number of tokens in the forward place $p_{i,i+1}$ is in position $2i-1$ and that of the backward place $p'_{i,i+1}$ is in position $2i$. Then, the initial marking is $m^0 = (0, 1, 0, 1, \ldots, 0, 1)$.

For ease of presentation, we will use the *submarking* $\bar{m}$ obtained from m by projecting away the odd elements, i.e., keeping only the token count in forward places and omitting the backward places. This can be done since each pair of complementary places has exactly 1 token in total. Correspondingly, $\bar{m}^0 = (0, \ldots, 0)$. Then, the number of possible markings is 2^{N-1}. We will show that they are all reachable. Fig. 6 depicts the graphs $\mathcal{R}(\mathcal{C})$ and $\mathcal{R}_S(\mathcal{C})$ for a chain $\mathcal{C}$ with $N = 4$ processes. Notice that, for a chain $\mathcal{C}$, two adjacent processes cannot be enabled at the same marking because the shared queue is either empty or full, thus disabling either the reader or (respectively) the writer.

Theorem 10. *Given an SFFP chain $\mathcal{C}$ with 1-place queues, the graph $\mathcal{R}_S(\mathcal{C})$ has a unique cycle consisting of two markings m^o and m^e, with submarkings $\bar{m}^o = (0, 1, 0, 1, \ldots, \mathrm{mod}(N, 2))$ and $\bar{m}^e = (1, 0, 1, 0, \ldots, \mathrm{mod}(N+1, 2))$. All odd-numbered processes are enabled in m^o and disabled in m^e. All even-numbered processes are enabled in m^e and disabled in m^o.*

Proof. We claim that, after $N-2$ steps starting from m^0, we reach in $\mathcal{R}_S(\mathcal{C})$ marking m^o if N is even and m^e if N is odd. At step j, marking m^j has components:

$$\bar{m}_i^j = \begin{cases} 0, & \text{if } i > j, \\ \mathrm{mod}(i+j+1, 2), & \text{if } i \leq j, \end{cases} \tag{3}$$

and the only (maximal) outgoing edge is annotated with the set of enabled transitions

$$\mathcal{T}^j = \{t_i : \mathrm{mod}(i+j+1, 2) = 0, \ i \leq j+1\}. \tag{4}$$

The above is true for $j = 0$, i.e., for the initial marking. Assuming it is true for some j, we want to prove it will be true for $j+1$.

By firing all the transitions in $\mathcal{T}^j$, we reach $\hat{m}^j$, where all of the output places of the fired transitions have one token and all of their input places have zero. Moreover, all other places have the same amount of tokens as in m^j. For simplicity, let us consider j to be even, then, by (3), only the even-numbered elements of $\bar{m}^j$ (the queues) up to element $\bar{m}_j^j$ will have a token. By (4), only the odd-numbered transitions up to $j+1$ are enabled. Hence, $\hat{\bar{m}}^j$ will be the same as $\bar{m}^{j+1}$. The complementary argument holds for j odd.

For brevity, the proof of uniqueness is omitted. □

Corollary 2. *For any chain SFFP $\mathcal{C}$, $\lambda^*(\mathcal{C}) = \frac{1}{2}$.*

When the size of queues in a chain is at least 2, the chain achieves maximal logical-time throughput, 1.

Theorem 11. *For any SFFP chain $\mathcal{C}$ with queues of sizes $k_i \geq 2$, $\lambda^*(\mathcal{C}) = 1$.*

Proof. Starting from m^0, after j steps, we reach marking m^j with components $\bar{m}_i^j = 1$ for $i \leq j$ and $\bar{m}_i^j = 0$ for $i \geq j$. Consider the marking $m^{\mathbf{1}} = m^{N-1}$ reached after $N-1$ steps, with submarking $\bar{m}^{\mathbf{1}} = (1, 1, \ldots, 1)$. Notice that, at $m^{\mathbf{1}}$, each queue is both nonempty and nonfull; hence, all processes in $\mathcal{C}$ are enabled and firing all of them results

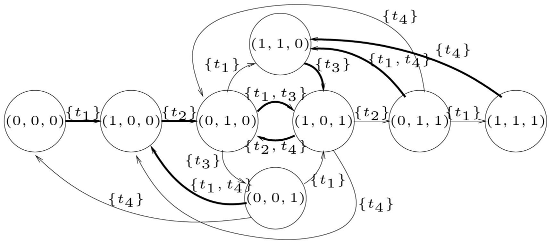


Fig. 6. Graphs $\mathcal{R}$ and $\mathcal{R}_\mathcal{S}$ for a four-process chain with one-place queues. Nodes are labeled by their submarking. Thick arrows denote maximal edges.

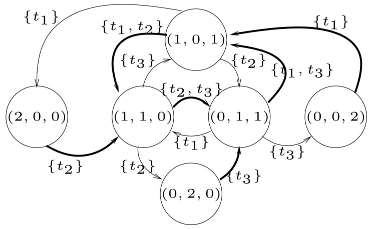


Fig. 7. Graphs $\mathcal{R}$ and $\mathcal{R}_S$ for a three-process loop $\mathcal{L}$ with two initial conditions and 2-place queues.

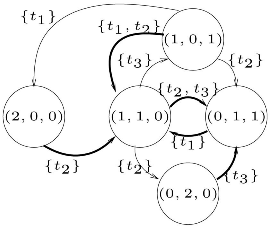


Fig. 8. The example in Fig. 7 where the third queue has size 1 instead of 2.

in the same marking, i.e., in $\mathcal{R}_S(\mathcal{C})$ marking $\bar{m}^1$ has an edge to itself. □

7.2.2 Loops

Consider an SFFP $\mathcal{L}$ consisting of a loop of N distinct processes $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_{N-1} \rightarrow P_N \rightarrow P_1$, with N total queues. Consider the MDG corresponding to $\mathcal{L}$. The state of the queues of $\mathcal{L}$ is characterized by the submarking $\bar{m}$, where $\bar{m}_i$ corresponds to the output queue of P_i. $\bar{m}_i$ is called *safe* if $\bar{m}_i \leq 1$ for all i. Let M be the total number of initial tokens in the loop, then the invariant $\sum_i \bar{m}_i = M$ holds for every reachable marking of the MDG. Since $\mathcal{L}$ comes from a synchronous model where every loop is broken by a unit delay, $M \geq 1$.

We first study the case where queues have sizes $k_i \geq 2$. Then, we characterize throughput for the case where queues have size M; hence, by Theorem 3, processes never skip due to full queues. Finally, we consider the case where some queues have size 1.

Fig. 7 depicts the graphs $\mathcal{R}(\mathcal{L})$ and $\mathcal{R}_S(\mathcal{L})$ for a loop with $N = 3$, $M = 2$, and two-place queues.

Lemma 2. *Given an SFFP loop $\mathcal{L}$, with M initial conditions and with queues of sizes $k_i \geq 2$, each cycle of $\mathcal{R}_S(\mathcal{L})$ is composed of safe submarkings.*

Proof. Since $k_i \geq 2$, at a safe submarking $\bar{m}$, all queues are nonfull. The first observation is that, from a safe submarking $\bar{m}$, we never go to an unsafe submarking along a maximal edge. In fact, since transitions are fired at most once per edge, to reach a submarking $\bar{m}'$ with $\bar{m}'_i = 2$, necessarily $\bar{m}_i = 1$ and we must not fire its consumer process P_{i+1} (using modulo arithmetic, $i = N \Rightarrow i + 1 = 1$). Since queues are nonfull, P_{i+1} is enabled at $\bar{m}$ but is not fired, which contradicts edge maximality.

Second, since $M \leq N$, any unsafe submarking has at least one empty queue.

Then, an unsafe submarking is followed by a submarking in $\mathcal{R}_S(\mathcal{L})$, where unsafe queues stay constant only if preceded by a nonempty queue and reduce their token count otherwise. Given a "train" of adjacent empty queues, the first queue becomes nonempty (but safe), and the last queue reduces the token count on the following queue. If the latter is safe, the train shifts forward; otherwise, the train shortens. Clearly, after at most N edges, we reach a safe submarking. □

Lemma 3. *Given an SFFP loop $\mathcal{L}$, with M initial conditions and queues of sizes $k_i \geq 2$, each simple cycle ℓ in $\mathcal{R}_S(\mathcal{L})$ has length $|\ell|$ with $h|\ell| = N$ for some $h \in \mathbb{N}$ and consists of all of the distinct rotations of a safe submarking.*

Proof. For $M = 1$, the proof is trivial and $h = 1$. We now discuss the case $1 < M \leq N$. By Lemma 2, all submarkings in a cycle ℓ are safe. A safe submarking $\bar{m}$ is followed in $\mathcal{R}_S(\mathcal{L})$ by a safe submarking, which is its forward rotation, so, after N edges, we must return to the same submarking $\bar{m}$. Necessarily, the path of length N from $\bar{m}$ describes a cycle in $\mathcal{R}_S(\mathcal{L})$. It may not be a simple cycle, in which case, we have repeated a simple cycle ℓ an integer number of times h. □

Theorem 12. *Given an SFFP loop $\mathcal{L}$, with M initial conditions, and queues of sizes $k_i \geq 2$, each simple cycle ℓ in $\mathcal{R}_S(\mathcal{L})$ has weight $w(\ell) = \frac{M}{N}|\ell|$. Hence, $\lambda^*(\mathcal{L}) = \frac{M}{N}$.*

Proof. By Lemma 3, starting from a safe submarking $\bar{m}$, after N consecutive edges in $\mathcal{R}_S(\mathcal{L})$, we describe a simple cycle ℓ exactly $\frac{N}{|\ell|}$ times. Moreover, since all of the tokens shifted back to their initial positions, each process fired exactly M times. Hence, $\frac{N}{|\ell|} w(\ell) = M$ and the result follows. □

Notice that the result of Theorem 12 also holds when queues have size M and greater and we can omit checking for queue fullness. For $M = 1$, the result is simple: Only one process is enabled at each step and $\lambda^*(\mathcal{L}) = \frac{1}{N}$ (note that $N \geq 2$). For $M \geq 2$, increasing the size of the queues cannot decrease throughput and $\lambda^*(\mathcal{L}) = \frac{M}{N}$ is the maximum achievable with M initial conditions.

Let us now consider the case where the SFFP loop $\mathcal{L}$ has single-place queues, except at positions with initial conditions, as described in Section 5.1. An example is shown in Fig. 8.

Theorem 13. *Given an SFFP loop $\mathcal{L}$, with M initial conditions and queue sizes as in Section 5.1. Then, $\lambda^*(\mathcal{L}) \leq \min\{\frac{M}{N}, \frac{1}{2}\}$.*

Proof. Reducing the size of queues cannot increase throughput; hence, by Theorem 12, $\lambda^*(\mathcal{L}) \leq \frac{M}{N}$.

Consider a process P writing into a single-place queue. It will not be able to write into the queue in two consecutive edges of $\mathcal{R}_S(\mathcal{L})$ because, after the first write, the place is full and the process is disabled. Hence, $\lambda^*(\mathcal{L}, P) \leq \frac{1}{2}$ and, by Theorem 8, the result follows. □

7.3 Throughput of Synchronous Models Implemented on LTTA

We have provided definitions and a set of results on throughput at the SFFP level, assuming negligible delays. This is clearly not a realistic assumption in LTTA; therefore, in order to study throughput of implementation of synchronous models on LTTA, we need to extend our method. We believe this can be done along the lines of what we presented above. Some ideas are provided here. They will be more thoroughly investigated in future work.

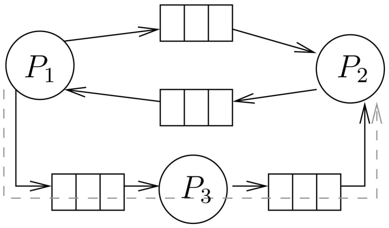


Fig. 9. A path on an FFP platform.

The basic idea is to adapt the construction of the MDG, modeling an SFFP process P using two, instead of one, transitions, t^T and t^B: t^T models the test of the firing condition of P and t^B models the execution of its body. The two transitions are connected in a tight loop with two complementary (forward and backward) places with one initial token in the backward place: t^B fires exactly once per each firing of t^T. We can use the algorithms presented earlier in this section to compute λ^* on the adapted MDG of an SFFP $\mathcal{F}$: Let this be denoted as $\lambda^+(\mathcal{F}, P_i)$, for a process P_i of $\mathcal{F}$. We expect the following:

Conjecture 1. *For any SFFP $\mathcal{F}$:*

$$\frac{1}{2} \geq \lambda^+(\mathcal{F}, t_i^T) \geq \frac{\lambda^*(\mathcal{F}, P_i)}{2}.$$

The first inequality is obvious due to the tight loop between t_i^T and t_i^B.

In an SFFP with nonnegligible delays, $fireno()$ and λ^{rt} must be defined with respect to an additional parameter δ, representing all of the delays encountered along an execution. Thus, $\lambda^{rt}(\mathcal{F}, P_i, c, \delta)$ represents the real-time throughput of P_i given triggering pattern c and delays δ. Let $\Delta_{\mathcal{F}}^{\max} > 0$ be the sum of the worst-case network transmission delay and the worst-case execution time, for all possible δ. We expect the following:

Conjecture 2. *Let $\mathcal{F}$ be an SFFP. Let c be a vector of clocks such that $\forall i, \forall n, c_i(n+1) - c_i(n) \leq \Delta_{\mathcal{F}}^{\max}$. Then, for any process P_i of $\mathcal{F}$, we have*

$$\lambda^{rt}(\mathcal{F}, P_i, c, \delta) \geq \frac{\lambda^+(\mathcal{F}, t_i^T)}{\Delta_{\mathcal{F}}^{\max}}.$$

Indeed, for every c, the two transitions t_i^T and t_i^B fire at most $\Delta_{\mathcal{F}}^{\max}$ time units apart, thus modeling the worst-case delays. Then, Theorem 9 would justify the result.

8 Latency

8.1 Computing Worst-Case Latency

8.1.1 Real-Time Latency

We define a notion of "real-time" latency in an SFFP $\mathcal{F}$ with respect to a *path* in $\mathcal{F}$. A path is a sequence of successive queues. The path must be acyclic. An example of a path is shown in Fig. 9. It consists of two successive queues.

Consider a path π from P_i to P_j: P_i is the process writing to the first queue in π and P_j is the process reading from the last queue in π. Let c be a clock vector: As stated above, c completely determines the behaviors of $\mathcal{F}$. Then, every message z generated by P_i takes a certain amount of time to "travel" along π until it is consumed by P_j. Let t_z be the time z is generated and t'_z be the time it is consumed. Denote the difference of $t'_z - t_z$ by $travel_{\mathcal{F},c,\pi}(z)$. Different messages z may spend different amounts of time traveling along π. We define latency as the worst-case travel time.

Definition 5. *The real-time latency along path π in an SFFP $\mathcal{F}$ with respect to a clock vector c is defined as*

$$\mu^{rt}(\mathcal{F}, \pi, c) = \sup_z travel_{\mathcal{F},c,\pi}(z).$$

8.1.2 Logical-Time Latency

We use the same idea as in Section 7.1.2: we define logical-time latency with respect to a reference clock χ. Let $travel^{\chi}_{\mathcal{F},c,\pi}(z)$ denote the number of ticks of χ in the interval of time that z spends traveling along path π. Formally, $travel^{\chi}_{\mathcal{F},c,\pi}(z) = \hat{\chi}(t'_z) - \hat{\chi}(t_z)$.

Definition 6. *The logical-time latency of path π in an SFFP $\mathcal{F}$ with respect to a clock vector c and a clock χ is*

$$\mu^{lt}(\mathcal{F}, \pi, c, \chi) = \sup_z travel^{\chi}_{\mathcal{F},c,\pi}(z).$$

The worst-case logical-time latency is defined as

$$\mu^{wclt}(\mathcal{F}, \pi, \chi) = \sup_{c \in C(\chi)} \mu^{lt}(\mathcal{F}, \pi, c, \chi).$$

Theorem 5 characterizes the worst-case clock triggering pattern for throughput. We are now going to prove a similar result characterizing the worst-case clock-triggering pattern for latency. First, we need a definition. Let χ be a clock and $c \in C(\chi)$. Let z be a message produced in the execution according to c. We define $C^{c,z}_{wc}(\chi)$ to be the set of all c' such that 1) c' is identical to c for $t \leq t_z$, i.e., up to the point when z is "born"; 2) for $t > t_z$, c' implements the "slow" triggering policy.

Theorem 14. *For any SFFP $\mathcal{F}$ and any path π of $\mathcal{F}$:*

$$\forall c \in C(\chi),\ \forall z,\ \forall c' \in C^{c,z}_{wc}(\chi),$$
$$travel^{\chi}_{\mathcal{F},c,\pi}(z) \leq travel^{\chi}_{\mathcal{F},c',\pi}(z).$$

Proof. Let m be the queue marking at time t_z. If π has n queues labeled $Q_1, \ldots, Q_n$ from source to destination, let m_i be the number of tokens in Q_i at m (thus including z). Then, in order for z to be consumed by the destination of π, say, process P_j, P_j has to fire $\sum_i m_i$ times. The longest amount of time that it takes P_j to do this corresponds to the situation where P_j fires at the slowest possible rate. By Theorem 5, this slowest rate is achieved by the "slow" triggering policy, which is what $c' \in C^{c,z}_{wc}(\chi)$ implements from m after time t_z. □

8.1.3 Computing Worst-Case Logical-Time Latency

The worst-case logical-time latency can be computed using $\mathcal{R}_S(\mathcal{F})$ and the result of Theorem 14. As in the proof of the latter, consider the queue marking m at time t_z. Being reachable, m belongs to $\mathcal{R}_S(\mathcal{F})$. Since executions using the "slow" triggering policy correspond to paths in $\mathcal{R}_S(\mathcal{F})$, we can simply "walk" along the lasso starting at m, counting how many edges $\mu_m(\mathcal{F}, \pi)$ it takes to fire P_j exactly $\sum_i m_i$ times. Then, by construction, the following holds:

Lemma 4. *$\forall \mathcal{F}, \forall \pi, \forall \chi, \forall c \in C(\chi)$, $\forall z, \forall c' \in C^{c,z}_{wc}(\chi)$:*

$$travel^{\chi}_{\mathcal{F},c',\pi}(z) = \mu_m(\mathcal{F}, \pi).$$

Notice that $\mu_m(\mathcal{F}, \pi)$ depends on the marking m but not on the value of the message z nor on χ. Define

$$\mu^*(\mathcal{F}, \pi) = \max_{m \in \mathcal{R}_S(\mathcal{F}), s.t. m_1 > 0} \mu_m(\mathcal{F}, \pi),$$

where we only explore markings for which some message z was born in the first queue along π, i.e., such that $m_1 > 0$.

Theorem 15. *For any SFFP $\mathcal{F}$ and any path π of $\mathcal{F}$:*

$$\forall \chi, \mu^{wclt}(\mathcal{F}, \pi, \chi) = \mu^*(\mathcal{F}, \pi).$$

Proof. From Definition 4, Lemma 4, and Theorem 14, it follows that $\mu^{wclt}(\mathcal{F}, \pi, \chi) \leq \mu^*(\mathcal{F}, \pi)$. Equality follows from the fact that $C^{c,z}_{wc}(\chi) \subseteq C(\chi)$. □

Contrary to throughput, worst-case latency does not behave in a "monotonic" fashion with respect to queue sizes. In particular, increasing the size of queues may either increase or decrease worst-case latency. It obviously may increase it since a token may find a greater number of tokens in front of it in a queue. But, increasing queue size may also increase throughput, thus resulting in a higher rate of token consumption and lower latency. We can only state the following (a corollary of Theorems 14 and 8).

Theorem 16. *Let $\mathcal{F}$ be a connected SFFP and $\mathcal{F}'$ be an SFFP derived from $\mathcal{F}$ by increasing the size of one queue in $\mathcal{F}$ by 1. Suppose $\lambda^*(\mathcal{F}') = \lambda^*(\mathcal{F})$. Then, for any path π, $\mu^*(\mathcal{F}, \pi) \leq \mu^*(\mathcal{F}', \pi)$.*

8.1.4 Linking Real-Time and Logical-Time Latency

Theorem 17. *Let $\mathcal{F}$ be an SFFP. Let Δ be any positive real number. Let c be a vector of clocks such that $\forall i, \forall n, c_i(n+1) - c_i(n) \leq \Delta$. Then, for any path π of $\mathcal{F}$,*

$$\mu^{rt}(\mathcal{F}, \pi, c) < \Delta \cdot (\mu^*(\mathcal{F}, \pi) + 1).$$

Proof. Define χ to be the perfectly periodic clock with period Δ. We then obviously have

$$travel_{\mathcal{F},c,\pi}(z) < \Delta \cdot \left(travel^{\chi}_{\mathcal{F},c,\pi}(z) + 1 \right)$$

and the result follows from Theorem 15. □

8.2 Special Cases/Topologies

8.2.1 Chains

Lemma 5. *Given an SFFP chain $\mathcal{C}$ with queues of sizes $k_i \geq 1$, the maximum number of consecutive markings in $\mathcal{R}_S(\mathcal{C})$ such that process P_i in the chain $\mathcal{C}$ can have its input queue empty is $i - 1$.*

Proof. Consider a marking m such that $\bar{m}_j = 0$ for $j < i$, i.e., all predecessor queues are empty. Let m^h be the marking reached from m in $\mathcal{R}_S(\mathcal{C})$ after h steps (so $m = m^0$). Then, for all markings m^h with $h = 0, \ldots, i-1$, the input queue of process P_i is empty and, in m^i, the queue is nonempty. This proves that there exists a marking from which it takes exactly $i - 1$ edges to fill the input queue of P_i.

Now, we prove that it cannot take longer. We can prove this result by induction on the position i in the chain. For $i = 1$, the result holds vacuously because there are no input queues to P_1. Assuming the result holds for P_i, we want to prove that it holds for P_{i+1}. Consider the path in $\mathcal{R}_S(\mathcal{C})$ starting from some marking m. Clearly, the output queue of P_i is the input queue of P_{i+1} and we need to prove that it is nonempty for at least one marking m^h with $h \leq i+1$. By assumption, process P_i will have its input queue nonempty in at least one marking $m^{h'}$ with $h' \leq i$. Hence, P_i was triggered at least once with a nonempty input queue. Two cases are in order, either its output queue was already full at $m^{h'}$ or it was nonfull. In the former case, the result is true (queue was nonempty). In the latter case, P_i fired and in its output queue became nonempty at the next marking $m^{h'+1}$. Since $h' + 1 \leq i + 1$, we proved the result for P_{i+1}. □

Lemma 6. *Given an SFFP chain $\mathcal{C}$ with queues of sizes $k_i \geq 1$, the maximum number of consecutive markings in $\mathcal{R}_S(\mathcal{C})$ such that process P_i in the chain $\mathcal{C}$ can have its output queue full is $N - i$. The proof is analogous to that of the previous lemma. In particular, the maximum is attained from a marking where all successor queues are full.*

Lemma 7. *Given an SFFP chain $\mathcal{C}$ with queues of sizes $k_i \geq 1$, the maximum number of consecutive edges in $\mathcal{R}_S(\mathcal{C})$ such that process P_i does not fire is equal to*

$$\max\{i-1, N-i\}.$$

The proof of Lemma 7 is based on the observation that process P_i can skip only if its input queue is empty or if its output queue is full (or both). The result then follows from Lemmas 5 and 6.

Theorem 18. *The worst-case logical-time latency in a chain $\mathcal{C}$ with one-place queues is $\mu^*(\mathcal{C}, \pi) = 2N - 3$ for the path π from P_1 to P_N.*

Proof. Consider the marking where all queues are full, except for the first queue (e.g., submarking (0, 1, 1) in Fig. 6). By firing P_1, we generate a new token z and we reach the marking with all queues full (e.g., (1, 1, 1)). By Lemma 6, for P_1 in $N - 1$ edges in $\mathcal{R}_S(\mathcal{C})$, the process P_2 moves the token z from the first queue to the second and we reach marking m^o. Then, we need to wait $N - 2$ ticks for z to "shift" in the chain and be consumed by P_N. □

Theorem 19. *The worst-case logical-time latency in a chain $\mathcal{C}$ with queues of size $k \geq 2$ is $\mu^*(\mathcal{C}, \pi) = k(N-1)$ for the path π from P_1 to P_N.*

Proof. Consider the marking where all queues are full except for the first queue (e.g., submarking (0, 1, 1) in Fig. 6). By firing P_1, we generate a new token z and we reach the marking with all full queues (e.g., (1, 1, 1)). By Lemma 6, for P_1 in $N - 1$ edges in $\mathcal{R}_S(\mathcal{C})$, the process P_2 moves a token from the first queue to the second. Now, there are $k - 1$ tokens in each queue and z is the last one. This is a marking where each queue is nonempty and nonfull, hence a self-cycle in $\mathcal{R}_S(\mathcal{C})$. At each edge, process P_N will consume one token and, after $(k-1)(N-1)$ edges, the token z is consumed. □

8.2.2 Loops

Theorem 20. *The worst-case latency in an SFFP loop $\mathcal{L}$ with M initial conditions and M-place queues corresponds to path π from P_i to itself and is $\mu^*(\mathcal{L}, \pi) = N + M - 1$.*

Proof. Let us focus, without loss of generality, on process P_1. The worst-case latency corresponds to the following scenario, cf., Fig. 6. Process P_1 fires from the marking where there is one token in its input queue and the other $M-1$ tokens are in its output queue; let the token just produced be the token z. The corresponding marking has all M tokens in the first queue. By following the lasso in $\mathcal{R}_S(\mathcal{L})$, after the $M-1$ edges, we reach the safe marking with one token in each of the first M queues and z is still in the first queue. We are now on the cycle in the lasso. After N edges, z is consumed by P_1. □

9 Conclusions

9.1 Summary

In this paper, we proposed a way of implementing synchronous models on a distributed real-time system platform, the Loosely-Time-Triggered (LTT) Architecture, consisting of a network of nodes triggered by local nonsynchronized clocks and communicating via CbS links.

To perform the implementation so that stream equivalence is preserved, we introduced an intermediate FFP. To avoid buffer overflow, a control mechanism was proposed which causes process P_i to skip communicating at each tick of its local clock when buffer overflow at one if its output links could result. This control mechanism is like backpressure used in latency insensitive protocols in hardware design.

We provided performance results for this implementation. In particular, we bounded from below its throughput and from above its latency in terms of topological characteristics of the FFP. These bounds were first obtained relative to a given reference clock χ and then refined to real-time bounds when the local clocks of the nodes are quasi-periodic.

9.2 Discussion

One may argue that stream semantics preservation is a questionable feature for real-time control systems. We believe that more work is needed to have a complete answer to this question. Some thoughts are briefly provided here. Our approach assumes that emissions can be postponed, i.e., data can be delayed. In practice, it is often the case that input data to the network of processes, such as those produced by sensors, are independent of network pace (e.g., a sensor sending samples periodically on a CAN bus). In such a case, the sensor cannot "skip." However, it is still possible to include the sensor in our framework, in two ways: The sensor writes to an (extended) CbS channel at its own rate, overwriting its previous value. Data can be lost if the sensor runs faster than the reader process, but the reader always has access to the "freshest" data. If the reader is faster, we can choose to use the same data multiple times or "skip." Actuators can also be included, connecting them to the network either through SFFP FIFOs (in which case, they may cause processes to "skip") or through (extended) CbS channels (in which case, data may be lost). Notice that, although clocks of the system (sensors, computers, and actuators) are subject to both relative jitter and drift, *the SFFP effective firings are only subject to jitter and suffer from no drift*. The overall latency of the system is also bounded. The bottom line is that LTTA offers interesting features for distributed real-time computer controlled systems.

9.3 Future Work

We noted that our algorithms to compute worst-case throughput and latency are based on a form of reachability analysis of marked graphs and are, to our knowledge, original. However, we believe that similar results can be obtained using the linear-algebra-based method of Sifakis [15] or the max-plus approach of Baccelli et al. [16]. Investigating this alternative and comparing it with our current approach in terms of accuracy and efficiency is one direction for future research. Another direction is to extend the throughput and latency analysis provided in Sections 7.1 and 8 to the case where execution time and communication delays are not negligible. It would be also useful to provide analysis not only for the worst case but also for the average case, with respect to assumptions on the random variables of the system. Adding jitter to the performance metrics, apart from throughput and latency, is another promising direction for research. Extending the synchronous model (for instance, by adding *multirate* features) is also needed. Finally, we would like to study how to add fault-tolerant features to our layered platform approach.

Acknowledgments

This research was supported in part by the European Commission under the projects IST-2001-34820 ARTIST and IST-004527 ARTIST2 Networks of Excellence on Embedded Systems Design, by the US National Science Foundation under the project ITR (CCR-0225610), and by the GSRC.

References

[1] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, pp. 558-565, 1978.

[2] N.A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[3] H. Kopetz, *Real-Time Systems*. Kluwer Academic, 1997.

[4] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert, "From Simulink to SCADE/Lustre to TTA: A Layered Approach for Distributed Embedded Applications," *Proc. Languages, Compilers, and Tools for Embedded Systems*, pp. 153-162, 2003.

[5] A. Benveniste, P. Caspi, P.L. Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis, "A Protocol for Loosely Time-Triggered Architectures," *Proc. Second Int'l Conf. Embedded Software*, pp. 252-265, 2002.

[6] K.-E. Årzén, "Timing Analysis and Simulation Tools for Real-Time Control," *Proc. Int'l Conf. Formal Modelling and Analysis of Timed Systems*, 2005.

[7] A. Sangiovanni-Vincentelli, "Quo Vadis SLD? Reasoning about Trends and Challenges of System Level Design," *Proc. IEEE*, vol. 95, no. 3, pp. 467-506, 2007.

[8] J. Romberg and A. Bauer, "Loose Synchronization of Event-Triggered Networks for Distribution of Synchronous Programs," *Proc. Second Int'l Conf. Embedded Software*, pp. 193-202, 2004.

[9] M. Baleani, A. Ferrari, L. Mangeruca, and A.L. Sangiovanni-Vincentelli, "Efficient Embedded Software Design with Synchronous Models," *Proc. Second Int'l Conf. Embedded Software*, pp. 187-190, 2005.

[10] A. Benveniste, B. Caillaud, L.P. Carloni, P. Caspi, A.L. Sangiovanni-Vincentelli, and S. Tripakis, "Communication by Sampling in Time-Sensitive Distributed Systems," *Proc. Second Int'l Conf. Embedded Software*, pp. 152-160, 2006.

[11] C. Kossentini and P. Caspi, "Approximation, Sampling and Voting in Hybrid Computing Systems," *Proc. Int'l Workshop Hybrid Systems: Computation and Control*, pp. 363-376, 2006.

[12] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli, "Theory of Latency-Insensitive Design," *IEEE Trans. Computer Automated Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059-1076, 2001.

[13] L.P. Carloni, "The Role of Back-Pressure in Implementing Latency-Insensitive Systems," *Electronic Notes in Theoretical Computer Science,* vol. 146, no. 2, pp. 61-80, 2006.

[14] J. Cortadella and M. Kishinevsky, "Synchronous Elastic Circuits with Early Evaluation and Token Counterflow," *Proc. Design Automation Conf.,* pp. 416-419, 2007.

[15] J. Sifakis, "Use of Petri Nets for Performance Evaluation," *Proc. Int'l Symp. Measuring Measuring, Modelling and Evaluating Computer Systems,* pp. 75-93, 1977.

[16] F. Baccelli, G. Cohen, G.J. Olsder, and J.-P. Quadrat, *Synchronisation and Linearity.* Wiley, 1992.

[17] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P.L. Guernic, and R. de Simone, "The Synchronous Languages 12 Years Later," *Proc. IEEE,* vol. 91, no. 1, pp. pp. 64-83, 2003.

[18] F. Xia, A. Yakovlev, I. Clark, and D. Shang, "Data Communication in Systems with Heterogeneous Timing," *IEEE Micro,* vol. 22, no. 6, pp. 58-69, Nov./Dec. 2002.

[19] F. Xia, F. Hao, I. Clark, A. Yakovlev, and E.G. Chester, "Buffered Asynchronous Communication Mechanisms," *Fundamenta Informaticae,* vol. 70, no. 1, pp. 155-170, 2006.

[20] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proc. IFIP Congress, Information Processing,* 1974.

[21] P. Caspi and M. Pouzet, "Synchronous Kahn Networks," *Proc. Int'l Conf. Functional Programming,* pp. 226-238, 1996.

[22] S. Even and S. Rajsbaum, "The Use of a Synchronizer Yields Maximum Computation Rate in Distributed Networks," *Proc. 22nd Ann. ACM Symp. Theory of Computing,* pp. 95-105, 1990.

[23] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proc. IEEE,* vol. 79, no. 9, pp. 1270-1282, 1991.

[24] J. Cortadella, A. Kondratyev, L. Lavagno, and C.P. Sotiriou, "Desynchronization: Synthesis of Asynchronous Circuits from Synchronous Specifications," *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems,* vol. 25, no. 10, pp. 1904-1921, 2006.

[25] K.C. Gorgônio, J. Cortadella, F. Xia, and A. Yakovlev, "Automating Synthesis of Asynchronous Communication Mechanisms," *Fundamenta Informaticae,* vol. 78, no. 1, pp. 75-100, 2007.

[26] F. Commoner, A.W. Holt, S. Even, and A. Pnueli, "Marked Directed Graphs," *J. Computer and System Science,* vol. 5, pp. 511-523, 1971.

[27] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proc. IEEE,* vol. 77, no. 4, pp. 541-580, Apr. 1989.

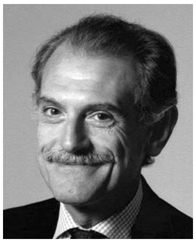
[28] A. Valmari, "Eliminating Redundant Interleavings during Concurrent Program Verification," *Proc. Parallel Architectures and Languages Europe,* pp. 89-103, 1989.

[29] M.D. Natale, A. Benveniste, P. Caspi, C. Pinello, A. Sangiovanni-Vincentelli, and S. Tripakis, "Applying LTTA to Guarantee Flow of Data Requirements in Distributed Systems Using Controller Area Networks," *Proc. Design, Automation and Test in Europe Workshop Dependable Software Systems,* 2008.

Stavros Tripakis received the PhD degree in computer science from Verimag Laboratory, Grenoble, France, in December 1998. From 1999 to 2001, he was a postdoctoral researcher at the University of California, Berkeley. From 2001 to 2006, he worked as a CNRS research scientist at Verimag. Since February 2006, he has been a research scientist at Cadence Research Laboratories, Berkeley, California. His research interests include embedded software, real-time and distributed systems, and computer-aided design and verification.

Claudio Pinello received the PhD degree in electrical engineering and computer sciences from the University of California, Berkeley, in 2004. He joined the Cadence Research Laboratories, Berkeley, in 2006. Before then, he held research positions at PARADES, Rome, at the BMW Technology Office, Palo Alto, California, at INRIA Rhône Alpes, France, at Cadence Berkeley Labs, and at General Motors Research. His interests are on embedded systems design, fault-tolerant distributed systems, and control theory and applications. He is a corecipient of two best paper awards at DAC 2007 and RTAS 2007.

Albert Benveniste received a degree from the École des Mines de Paris in 1971. He performed his These d'État in mathematics, probability theory, in 1975. From 1976 to 1979, he was an associate professor in mathematics at the Université de Rennes I. Since 1979, he has been the directeur de recherche at INRIA-Rennes. His current interests include change detection in signal processing and automatic control, vibration mechanics, reactive and real-time embedded systems design in computer science, and network and service management in telecommunications. He was a coinventor, with Paul Le Guernic, of the synchronous language signal for embedded systems design. Since 1996, he has been active in distributed algorithms for network and service management in telecommunications, where he has contributed to distributed fault diagnosis. He has published more than 100 journal papers and several hundred conference proceedings papers. He is a fellow of the IEEE.



Alberto Sangiovanni-Vincentelli holds the Buttner endowed chair of electrical engineering and computer scienc at the University of California, Berkeley. He was a cofounder of Cadence and Synopsys. He is a member of the General Motors Scientific and Technology Advisory Board. He received the Distinguished Teaching Award of UC, the worldwide 1995 Graduate Teaching Award of the IEEE for "inspirational teaching of graduate students," and the Kaufman Award of the EDA Council for pioneering contributions to EDA. He is the author of more than 800 papers and 15 books on EDA, embedded, and hybrid systems. He is a member of the NAE and a fellow of the IEEE.

Paul Caspi received a degree from the École Polytechnique, Paris, and a Docteur ès Sciences degree from the Institut National Polytechnique de Grenoble in 1978. Since then, he worked at CNRS Laboratoire Verimag until he retired in October 2008 and he is currently a consultant in safety-critical control systems. His research interests include safety-critical control systems from both hardware and software points of view. This has led him to contribute to the design of the Lustre synchronous language, which played a pioneering part in model-based automatic code generation and has become a standard in European avionic and railway industries. He received jointly with Nicolas Halbwachs the Monpetit prize of the Académie des Sciences for this achievement.

Marco Di Natale received the PhD degree in computer engineering and is currently an associate professor at the Scuola Superiore S. Anna, Pisa, Italy, teaching embedded systems and real-time systems courses. His research interests include the modeling, design, and timing evaluation of embedded systems and the evaluation of distributed architectures. More recently, he started investigating the opportunity for the synthesis of the software architecture implementation with respect to timing constraints and metrics. He is a member of the IEEE.