Efficient Distribution of Triggered Synchronous Block Diagrams

Yang Yang
Stavros Tripakis
Alberto L. Sangiovanni-Vincentelli

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-115
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-115.html

October 21, 2011

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the U. S. Army Research Office (ARO #W911NF-11-2-0038), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.

Efficient Distribution of Triggered Synchronous Block Diagrams*

Yang Yang Stavros Tripakis Alberto Sangiovanni-Vincentelli
EECS Dept., UC Berkeley
{yangyang,stavros,alberto}@eecs.berkeley.edu

October 21, 2011

Abstract

Most of the design challenges for complex cyber-physical systems, where a digital controller governs a multi-physics plant, relate to the distributed nature of the systems to be controlled. Cars, airplanes, and power distribution grids are well-known examples. The characteristics of the communication network that connects the system components affect the derivation of the control law and the verification of design correctness. For this reason, there has been a strong interest in using languages and methodologies that facilitate the use of formal methods. These languages and methodologies are mostly based on a synchronous paradigm that, while satisfies the need for formalization, often results in an inefficient implementation requiring substantial overhead when compared to approaches that do not enforce synchronicity on the execution platform. Therefore, the interest is high for techniques that on one hand, maintain the formal properties of synchronous models, and on the other hand, enable the use of asynchronous and distributed execution platforms with little overhead.

In this paper we address the problem of automatic synthesis, and in particular automatic and semantics-preserving implementation of Triggered Synchronous Block Diagrams (SBDs) on distributed, asynchronous execution platforms. This problem was studied for "pure" SBDs (where all blocks are triggered in every synchronous step) in [23]. The method of [23] can be adapted to Triggered SBDs by using trigger elimination [16], where triggers are transformed to standard inputs. However, this often results in unnecessary communication overhead. In this paper we propose methods to minimize this overhead, thus improving the efficiency of the approach. We consider both general Triggered SBDs where the values of triggers are dynamically computed and are thus not known a-priori, as well as Timed SBDs where triggers are statically known, usually specified by (period, initial phase) pairs.

1 Introduction

Cyber-physical systems involve complex interactions between physical environment and electronic control systems, and usually consist of heterogeneous spatially-distributed subsystems exchanging information on asynchronous networks. Cyber-physical systems are difficult to design and to verify given these characteristics. There is a great deal of interest in developing approaches that prevent unexpected and unwanted behaviors. As witnessed in hardware design, synchronous approaches are effective in providing a formal framework for design. Various synchronous design tools and languages were proposed for modeling, simulation, verification and synthesis of complex cyber-physical systems, including SCADE, Lustre [9], Simulink and others. Simulink, based on the model-based design paradigm, is widely used in many application domains (e.g. automotive, avionics, industrial control) for capturing both the control systems and the physical environment/plants . At the heart of these languages, synchronous Block Diagrams (SBDs) [11] are usually chosen as the model of computation, because they facilitate formal analysis of the system behavior and verification of the design correctness. Synchronicity is particularly interesting in safety-critical cyber-physical systems given the predictability of synchronous designs.

*This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the U. S. Army Research Office (ARO #W911NF-11-2-0038), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota.

However, the implementation of synchronous designs is often inefficient requiring significant resource overhead and yielding inferior performance. In an ideal world, we would like to maintain the formalization aspects of synchronous designs while exploiting the efficiency of asynchronous implementations. Indeed, cyber-physical systems being addressed today do pose significant pressure on verification and on the effectiveness of the implementation platform. We are interested in methods that starting with a synchronous abstract design, derive an implementation that is semantically equivalent (hence any property valid in the abstract is also valid in the concrete) and does not require adherence to the synchronous paradigm (therfore minimizes any overhead needed to achieve semantic equivalence). We believe this approach, if supported by appropriate synthesis tools, is a strong candidate for standardization in design flows used in cyber-physical system design.

The fundamental component in an SBD is a *block*, which can be modeled as a (not necessarily finite) state machine with inputs and outputs à la Mealy. Outputs of blocks can be connected to inputs of other blocks to form a diagram. The semantics of such diagrams are *synchronous* in the sense that all blocks proceed in *lock-step*. Provided the diagram has no cyclic dependencies (within a step), all blocks "fire" within a synchronous step in a certain order, so that the external outputs of the diagram are computed by propagating the external inputs throughout the diagram. The firing of a block corresponds to a local reaction step of the corresponding state machine: the machine reads its local inputs, computes its local outputs and updates its local state.

Triggered SBDs are an extension of SBDs where the firing of a block may be controlled by a boolean signal called a *trigger*. At a given synchronous step, if the trigger is $true$, the block fires normally; otherwise, the block *stutters*, that is, keeps its local state and local outputs unchanged, until the next step. Triggered SBDs are useful for modeling *multi-rate* systems, where different parts of the system operate at different time scales. Notice that the triggering patterns need not be periodic. A triggering signal for a block A may be produced by another block B, or it may even be an external input of the diagram. The point is that the behavior of the triggers (i.e., at which steps they are $true$ or $false$) is generally unknown. An exception is the special case of *Timed* SBDs, where triggering patterns are known statically ("at compile time").

Although Triggered SBDs can be translated into equivalent SBDs by a *trigger elimination* procedure that transforms triggers into standard inputs [16], this is often undesirable. In particular, as we show in this paper, it can result in inefficient distributed implementations, with higher communication loads than necessary.

The problem we address in this paper is the *semantics-preserving and communication-efficient distribution of Triggered SBDs on asynchronous execution platforms*. In particular, given a design specification described as a Triggered SBD, how to map it to a distributed, asynchronous execution platform, so that the semantics of the Triggered SBD is *stream-equivalent* to the semantics of the distributed implementation, and the communication overhead between the distributed processes is reduced. Addressing such problem is important as distributed platforms are commonly used in various cyber-physical systems. Without a formal synthesis methodology, we will not be able to guarantee the correctness of the distributed implementation w.r.t. the original synchronous specification.

The semantics-preserving distribution problem has been studied in [23], but only for a "pure" SBD model, where all blocks fire at every synchronous step. In this paper we generalize these results to the case of Triggered SBDs. We also study distribution of Timed SBDs as a special case, for which more efficient implementations can be obtained.

We follow the problem formulation of [23] where "distributed asynchronous execution platforms" are captured by so-called *finite FIFO platforms* (FFPs). An FFP is similar to a Kahn Process Network (KPN) [13], with the difference that while in a KPN queues are unbounded, in an FFP they are of fixed, finite size. Although FFPs model a specific kind of distributed systems and in particular network communication, they can themselves be mapped in a semantics-preserving way to a variety of underlying networks, such as onto the *loosely time triggered architecture* (LTTA) [23]. Therefore, FFPs represent a useful intermediate layer that can serve as a first step in distributing a model onto many different execution platforms (all platforms upon which FIFO queues can be implemented, e.g., using the TCP protocol). This can be done because FFPs make *no assumption about the relative speed of the local clocks of distributed processes*, hence the characterization *asynchronous*.

There is a simple way to solve the distribution problem for Triggered SBDs: first, apply trigger elimination to translate the Triggered SBD into a pure SBD; then, use the mechanisms of [23] for distribution of pure SBDs. Unfortunately, this simple method often results in unnecessary communication overhead: a block always sends output messages even when its trigger is $false$. The block does not fire in this case, so the outputs have the same value as in the previous step, but they are still transmitted to downstream blocks. In this paper we present an implementation method

that eliminates this overhead. This is especially critical in CPS where communication is expensive, for example, in wireless applications where the channel capacity is limited, or where energy savings are essential.

In particular, our implementation method optimizes communication along the following two directions: first, data messages are not sent to processes that are not triggered; second, a process which is not triggered need not send a full data message to its successor processes, but only a flag indicating that the data are the same as in the previous step. In addition to these optimizations that apply to general Triggered SBDs, we also present further optimizations for the case of Timed SBDs.

1.1 Motivating Examples

Fig. 1 shows a Triggered SBD. This diagram models a two-mode system, consisting of two separate sets of communicating blocks, plus a mode control block that triggers only one of the sets at any given time. The output of the control block is a boolean signal: when it is $true$, the blocks of Mode 1 are triggered, and when it is $false$, the blocks of Mode 2 are triggered. Notation-wise, we use different types of arrow heads to distinguish triggering signals from standard inter-block communication signals, and we usually draw triggering signals as incoming to the top of a block.

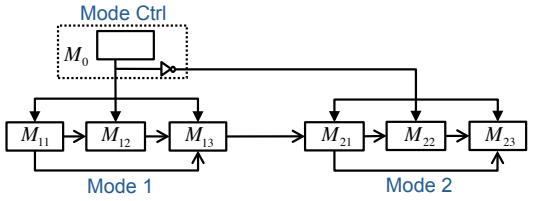

Figure 1: A Triggered SBD.

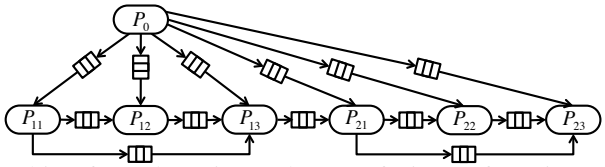

Figure 2: The FFP system resulting from the Triggered SBD of Fig. 1 after trigger elimination [16] and distribution [23].

After applying the trigger elimination method of [16], followed by the distribution method of [23], we get the FFP diagram shown in Fig. 2. The FFP diagram is a model of a distributed system where concurrent processes communicate through FIFO queues. Each block M_0, M_{11}, M_{12}, etc., of the original diagram gives rise to a process P_0, P_{11}, P_{12}, etc., in the FFP. The triggers of the original diagram have now become standard inputs to the FFP processes. Each FFP process P executes the following pseudo-code:

```
P(inputs: ins, trigger; outputs: outs)
{
  initialize state and outs;
  while (true) {
    wait until all input/output queues
       are non-empty/non-full;
    get_inputs(ins, trigger);
    if (trigger) then
      (state, outs) := M.step(state, ins);
    put_outputs(outs);
  }
}
```

where `state` denotes the internal state of M, that P inherits. In addition, output variables `outs` are also state variables in P. Process P behaves as follows. It starts by initializing its state variables (including `outs` – the reason for this will become clear below). It then enters an infinite loop. At each iteration, P waits until all its input queues are non-empty (i.e., contain at least one message) and all its output queues are non-full (i.e., have room for at least one

message). Then, P "fires", that is, it performs a synchronous step: one input message is read from each input queue (including the trigger) and one output message is written to each output queue, using the functions `get_inputs()` and `put_outputs()` (we assume that these functions are "smart enough" to know which variable corresponds to which queue). When the trigger is $true$, P uses the output function of M, `M.step()`, to update the outputs and the state. When the trigger is $false$, no updates are made and the values written at the outputs are the same as in the previous step (i.e., the process "stutters").

All processes in the FFP of Fig. 2 execute concurrently, following the above pattern P. Although the processes are not synchronized, some loose form of synchronization is still imposed because of the queues: a process cannot fire when it waits for an input from another process, or for a downstream process to free up space in an output queue. This distributed concurrent system completes a *logical step* when all messages corresponding to the same synchronous step in the original SBD have been processed.

We use this notion to estimate the communication load in this FFP implementation. We can see that 6 trigger messages plus 7 data messages are transmitted at every logical step. The 6 trigger messages correspond to the messages sent from the control process P_0 to each of the other processes (the negation block is not implemented as a separate process, but is part of the control process). The data messages are sent by the processes among themselves: two messages from P_{11} to P_{12} and P_{13}, one from P_{12} to P_{13}, one from P_{13} to P_{21}, and so on. Let L_T and L_D denote the message lengths for trigger and data messages, respectively. Then, the communication load of the naive implementation is $6 \cdot L_T + 7 \cdot L_D$, measured in bits per logical step.

In the optimized implementation method that we present in this paper, a producer process only sends a message to a consumer process when the consumer is triggered. In our running example, P_{11} only sends messages to P_{12} and P_{13} when the latter are triggered. In this example all processes in the set $\{P_{11}, P_{12}, P_{13}\}$ are triggered simultaneously, and similarly for $\{P_{21}, P_{22}, P_{23}\}$. Moreover, only one of the two sets is triggered at any given logical step. Therefore, in the optimized implementation, at most 4 data messages are transmitted in each logical step: 3 messages among processes of the same mode, plus 1 message from P_{13} to P_{21}. Moreover, the message from P_{13} to P_{21} is only transmitted at the beginning of a mode switch. After that, while the system remains in the same mode, only a control message is transmitted indicating that the data is the same as in the last step. The savings are significant and can be close to $\frac{4}{7} \approx 57\%$, considering that the data messages are usually much longer than trigger/control messages (whose payload is only a few bits).

Even more significant savings arise in the case of Timed SBDs, where triggering patterns are known statically. An example is shown in Fig. 3. The writer block W is fired at every synchronous step, while the reader block R is fired only once every 10 steps. In the naive implementation, process W would send a message to R at every logical step. R would execute to consume these messages and then do nothing 9 out of 10 times. $\frac{9}{10}$ of those messages are unnecessary, and are eliminated by our mechanism.

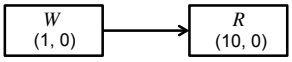

Figure 3: A Timed SBD.

2 Background: Triggered SBDs and FFPs

2.1 Triggered SBDs

A Triggered SBD consists of a set of *blocks* connected to form a *diagram*. Each block has a number of input ports (possibly zero) and a number of output ports (possibly zero). Diagrams are formed by adding connections. There are two types of connections: a *data connection* connects some output port of a block M to some input port of another block M'; a *trigger connection* connects some output port of a block M directly to another block M': we say that M' *has a trigger*. A block can have at most one incoming trigger (it can also have none). An output port can be connected to more than one input ports. However an input port can only be connected to a single output.

Semantically, each block corresponds to a state machine, generally of type Mealy [14]. We say that a block is "Moore" if its output function only depends on its state, and not on the inputs. Every connection in the diagram

corresponds semantically to a *stream*, that is, a function $s : \mathbb{N} \to \mathcal{U}$, where $\mathbb{N} = \{0, 1, 2, ...\}$ is the set of natural numbers, $\mathcal{U}$ is the universe of all possible data values that streams in the diagram can take, and $s(n)$ represents the value of s at the n-th synchronous step. For simplicity, we ignore typing issues, which in practice would only allow connections between ports of compatible types. However, we use terms such as "boolean signal" for streams that only take values in a restricted subset of $\mathcal{U}$, e.g., $\{true, false\}$ for boolean signals.

The semantics of a diagram can be given as a *composite state machine*, obtained by synchronous composition of all machines corresponding to blocks in the diagram. To define the composite state machine, we assume that the diagram is *acyclic*, that is, every dependency cycle visits at least one Moore block. We also assume that there are no "self-loops": this is not a restrictive assumption since blocks can have internal state. The state space of the composite machine is the product of the state spaces of all its component machines, plus all outputs of blocks that have triggers. These outputs become states because when a block is not triggered, its outputs maintain their previous value. The outputs of the composite machine can be defined to be all outputs in the system (including those connected to inputs).

The state of the composite machine is updated by updating the states of all individual components. The output function of the composite machine is defined by defining the value $s(n)$ of every stream s in the diagram, for a given $n \in \mathbb{N}$. Suppose s is the output of machine M. If M has no trigger, $s(n)$ is defined by the output function of M. This requires the local inputs of M to be already known, but since the diagram is acyclic, there always exists a well-defined order in which to evaluate all streams in the diagram at every step n. If M has trigger t and $t(n) = true$, again $s(n)$ is defined by the output function of M. If M has trigger t and $t(n) = false$, $s(n) = s(n-1)$ (if this happens when $n = 0$, some default value is used for $s(0)$). Notice that M having no trigger is equivalent to M having a trigger which is $true$ at every step.

A *Timed SBD* is a special case of a Triggered SBD where every trigger is generated by a (period, initial phase) pair (PPP) $(\tau, \theta) \in \mathbb{N} \times \mathbb{N}$, where τ represents a period and θ an initial phase.[1] For example, the pair $(2, 1)$ generates the stream $false\ true\ false\ true\ \cdots$. Clearly, every PPP can be defined by a finite state machine, so Timed SBDs are a subclass of Triggered SBDs. The important thing about Timed SBDs is that the triggering pattern is known "at compile time". This is not the case for general Triggered SBDs. Note that the implementation methods that we present here, as well as those proposed in [23], are agnostic of the internals of blocks, that is, blocks are treated as black boxes whose internal state machines are not known.

2.2 FFPs

We model the distributed, asynchronous execution platform as a *Finite FIFO Platform* (FFP) [23]. An FFP consists of a set of sequential processes communicating via directed, point-to-point, lossless, FIFO queues of finite length. As such, an FFP is similar to a Kahn process network (KPN) [13], with the difference that in a KPN the queues are unbounded. Another difference is that in an FFP, unlike in a KPN, both reads and writes are non-blocking in an FFP and the processes have the responsibility for checking that the queue is non-empty before doing a read, and that the queue is non-full before doing a write. An example FFP is shown in Fig. 2. It consists of 7 processes and 12 queues (not necessarily of size 3, or of the same size).

Each FFP process is a sequential program that calls special API functions to access the services of the queues, in particular, `isFull()` and `isEmpty()`, to check whether a given queue is full or empty, and `get()` and `put()`, to pop and return the first element of a (non-empty) queue, and to append a message at the end of a (non-full) queue. An example of an FFP process is process P executing the pseudo code shown in Section 1.1 (`get_inputs()` iterates `get()` over all input queues and `put_outputs()` iterates `put()` over all output queues). Note that not all FFP processes must look like that example. Indeed, the distribution methods that we present in Sections 3 and 4 rely on different FFP process structures and achieve better communication efficiency.

2.3 The Distribution Problem

The distribution problem is to automatically generate from a given Triggered SBD, an FFP that is *stream-equivalent* to the Triggered SBD. Generating an FFP means synthesizing the topology of the FFP (processes and FIFO queues) and

[1] More generally, triggers in timed SBDs could be specified by *firing time automata* (FTA) [16]. Our implementation method can be directly extended to FTA, but for simplicity, we limit our discussion to PPPs.

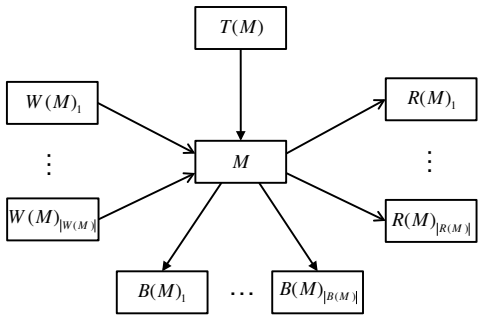

Figure 4: A block M and its surroundings.

the code that each FFP process executes (the topology synthesis is easy, since we assume a 1-1 mapping of blocks to processes, as is done in [23]). In an FFP, a stream is essentially the sequence of values that are written in a given queue. In the naive implementation, stream equivalence requires that every stream s^* produced in the FFP be identical to the corresponding stream s defined by the Triggered SBD. This requirement is too strict for the optimized implementation, where redundant messages are omitted from s^*. Instead, we require only that s^* be identical to s sampled at the points in time when the consumer of s is triggered.

Note that, contrary to Triggered SBDs, streams of FFPs are not guaranteed to be infinite. This is because some processes in an FFP may "deadlock", waiting forever for messages in an input queue or space in an output queue. A proof of semantical preservation must therefore include arguments to show that the resulting FFPs are deadlock-free [23].

As mentioned in Section 1, the straightforward, "naive" solution to the distribution problem is to combine the trigger elimination procedure of [16] with the distribution method of [23]. This method creates communication overhead, however, as illustrated by the examples of Section 1.1. In what follows we propose alternative implementations that eliminate this overhead while preserving the semantics.

3 Distribution of General Triggered SBDs

We first introduce some notation and terminology. Fig. 4 shows the general configuration of a block M and its *surroundings*, as a part of a Triggered SBD.

- If M has a trigger t, $T(M)$ denotes the block that produces t (i.e., that has t as an output). If M has no trigger, $T(M)$ is undefined: we examine this as a special case below.
- The set of blocks that have data connections into M is denoted as $W(M)$. [2]
- $B(M)$ denotes the set of blocks triggered by M.
- $R(M)$ denotes the set of blocks that have data connections from M, except for those blocks that are already in $B(M)$. $R(M)$ is partitioned in two disjoint subsets: $RR(M)$, containing all blocks in $R(M)$ that either have no trigger or have a trigger but are already in $W(B(M))$; and $RT(M)$, containing all the remaining blocks of $R(M)$.

Note that $W(M), R(M), B(M)$ are pairwise disjoint. Also, absence of self-loops ensures that M cannot be a member of any of these three sets. Finally, $T(M)$ cannot be an element of neither $R(M)$ nor $B(M)$ (this would result in cyclic diagrams) but it may be an element of $W(M)$.

[2] For a set W, $|W|$ denotes its cardinality. We use W_1, W_2, etc., to enumerate and denote its elements, so that $W(M) = \{W(M)_1, ..., W(M)_{|W(M)|}\}$. Also, we define $W(X) = \bigcup_{Q \in X} W(Q)$, for a set of processes X.

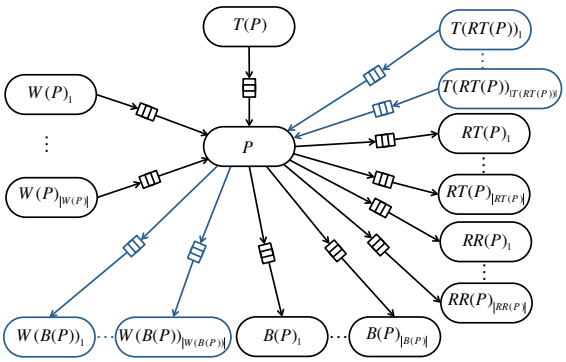

Figure 5: Part of an FFP generated from block M and its surroundings shown in Fig. 4.

3.1 Mapping Triggered SBDs on FFPs

A Triggered SBD is mapped into an FFP in the following way. Every block M in the Triggered SBD is mapped to an FFP process P. Every link between a pair of blocks M and M' in the Triggered SBD is mapped to a FIFO queue between the corresponding FFP processes, from P to P'. The sizes of the queues are as in [23]. In particular, if M is not Moore, a queue of size 1 suffices; if M is Moore, a queue of size 2 suffices: this queue is initialized with a message carrying the initial output of M. Schematically, the Triggered SBD part shown in Fig. 4 results in the FFP part shown in Fig. 5.

Similarly to notation $T(M), W(M), ...$ for blocks, we introduce notation $T(P), W(P), ...$ for processes. That is, if block M is mapped to process P, $T(P)$ denotes the process corresponding to $T(M)$, $W(P)$ denotes the set of all processes P' such that P' corresponds to block $M' \in W(M)$, etc.

As can be seen from Fig. 5, P may have more inputs and outputs (shown in blue) than its corresponding block M. In particular, P receives additional input signals from processes in $T(RT(P))$. This is done in order to minimize data traffic: if a process $P' \in RT(P)$ is not triggered in a given step, P need not send a message to P' for that step. To know whether P' is triggered or not, P needs to receive a message from the process triggering P', that is, from $T(P')$. These additional signals are called *backward* signals and the corresponding queues are called backward queues. They are illustrated in Fig. 6. Backward signals are sent to backward queues at every step.

Symmetrically, P itself may trigger other processes (those in $B(P)$). Therefore, P needs to notify potential writers of processes in $B(P)$ about whether the latter are triggered or not. This explains the additional output queues of P, namely, queues to the process set $W(B(P))$.

We should note that additional queues are introduced by the optimized implementation only if they do not already exist in the naive implementation. For example, the process $T(P)$ may also be in $T(RT(P))$. This is the case in Fig. 2, where $T(P_{11}) = T(P_{12}) = P_0$. Since there is already a queue from P_0 to P_{11}, no additional queue is needed.

Additional backward queues may create apparent dependency cycles in the FFP, as illustrated in Fig. 7. If M_1 already has a forward link to M_3, adding a backward queue from P_3 to P_1 in the FFP creates a cycle. To ensure that such cycles are not problematic, i.e., do not result in deadlocks, a process P is designed in a way such that its execution is structured in *stages*. The stages are ordered so that dependency cycles are not introduced. In the example of Fig. 7, P_1 will transmit to P_3 without waiting for messages from the backward queue. These messages are necessary only in order for P_1 to decide whether to send a message to P_2 or not, and are not needed for P_1 to compute its outputs.

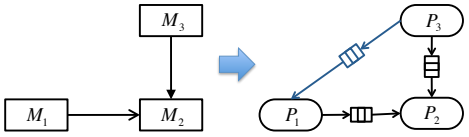

Figure 6: Backward queue sending trigger information about P_2 to P_1.

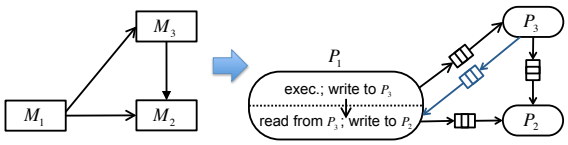

Figure 7: Avoiding deadlocks by structuring each process in stages.

The code that each FFP process P executes is shown in Fig. 8. The code follows the same general scheme as the naive implementation described in Section 1.1: initialization of state variables, followed by execution of an infinite loop. Every iteration of the loop proceeds in a number of stages. First (Stage 0), P determines if it is triggered in the current iteration. If $T(P)$ is undefined, P is implicitly always triggered, therefore `trigger` is set to *true*. Otherwise, P needs to consume a message from the input queue `trigger` coming from process $T(P)$ and containing the value of the trigger. If the queue is empty, P needs to wait until a message arrives. At Stage 1, P fires iff the trigger is *true* and sends messages to $RR(P) \cup B(P) \cup W(B(P))$ (the union of the sets is denoted as $RB(P)$ in the code): these messages are sent at every step, even when P is not triggered. At Stage 2, P sends messages to those processes in $RT(P)$ that are triggered: the rest need not receive data messages. This is part of the traffic optimizations that our method achieves.

Returning to the example of Fig. 7, the transmission from P_1 to P_3 will occur at Stage 1, since $P_3 \in RR(P_1)$. Then, P_3 can execute and transmit back to P_1 via the backward queue. Once P_1 has this information, it can decide whether a message needs to be sent to P_2. If so, this will happen at Stage 2 (in P_1's code). One can see how this careful ordering avoids dependency cycles and deadlocks in this example. More complicated cases exist, however, for instance, where P_3 itself may be triggered, therefore belonging not to $RR(P_1)$ but to $RT(P_1)$. The proof of semantical preservation described in Section 3.2 argues how these cases are also handled correctly by our method.

We now further explain the code of P shown in Fig. 8. `ins` denotes the set of all input queues of P. `outs` denotes the set of all output queues. We use notation such as `ins[i]` to denote the queue from a given process `i`. Similarly, if X is a set of processes, `ins[`X`]` denotes the set of the corresponding queues.

P maintains state variables `ins'` and `outs'`. For each `i`, `ins'[i]` memorizes the last data message received in `ins[i]`. This is used when a process has no "fresh" message for P (i.e., no new message since the last time P was triggered), in which case it only sends a flag to P indicating that the last data message should be used. Symmetrically, for each output queue, `outs'` memorizes the latest message that P produced for that queue. Note that `get_inputs()` and `put_outputs()` use only `ins` and `outs` and do not affect `ins'` and `outs'`.

Messages in `outs'` contain an extra boolean flag `fresh`, indicating that the corresponding output is newly produced, as opposed to one that has already been sent. Initially all output data are fresh: this is because in the first iteration these data must be sent even if P is not triggered. When `put_outputs()` takes `outs'` as argument, it uses the `fresh` flag of each message: if it is *true*, the whole message is sent; otherwise, only the flag is sent, indicating that the data is the same as in the last transmitted message. This reduces communication load, since data messages typically have a larger payload. Note also that each message sent by `put_outputs()` contains all the information that must be transmitted from one process to another. Such a message may therefore include both a trigger and a data part.

For each process `i` in $T(RT(P))$, P also maintains a boolean flag `known[i]`. These flags are used to indicate whether the value of certain triggers is known at a given iteration. All flags are reset to *false* at the beginning of each iteration. As messages are received, the corresponding flags are set to *true*.

`RTunproc` represents the set of all processes in $RT(P)$ that P needs to consider in Stage 2. For each $rt \in RT(P)$, P needs to determine if rt is triggered: if so, P sends a message to rt, otherwise, it does not. P iterates over all processes in $RT(P)$ until all of them have been handled. A process `rt` is selected at random, and P checks whether the triggering status for `rt` is known. If not, P attempts to find out by checking whether the backward queue from `T(rt)` contains a message. If the trigger value for `rt` is known, if it is *false*, P need not send a message to it. If the trigger is *true*, P sends a message if space is available in the corresponding queue. In these cases, `rt` is removed from `RTunproc`, marking the fact that `rt` has been handled.

Stage 2 may appear unnecessarily complicated: why not simply iterate over all processes $rt \in RT(P)$, wait for

```
P (inputs: ins, trigger; outputs: outs)
{
  initialize state, outs, ins' and outs';
  for all i, outs'[i].fresh := true;
  while (true) {
    for all i in T(RT(P)), known[i] := false;
    // Stage 0: determine trigger
    if  (T(P) is defined) {
      wait until trigger queue is not empty;
      get_inputs(trigger);
      if(trigger.fresh = true)
        ins'[T(P)] := trigger;
      if (T(P) in T(RT(P))) known[T(P)] := true;
    }
    else
      trigger := true;
    // Stage 1: fire and send to RB(P), where
    // RB(P) := RR(P) union B(P) union W(B(P))
    wait until no queue to RB(P) is full;
    if (trigger) {
      wait until no queue from W(P) is empty;
      get_inputs(ins[W(P)\T(P)]);
      for (every i in W(P)\T(P) s.t.
           ins[i].fresh = true)
        ins'[i] := ins[i];
      for (every i in W(P)\T(P) s.t. i in T(RT(P)))
        known[i] := true;
      (state, outs) := M.step(state, ins'[W(P)]);
      for all i in (RB(P)) {
        outs'[i].fresh := true;
        outs'[i].data := outs[i];
      }
    }
    put_outputs(outs'[RB(P)]);
    for all i in (RB(P))
      outs'[i].fresh := false;
    // Stage 2: selectively send to RT(P)
    RTunproc := RT(P);
    while (RTunproc != empty) {
      pick a process rt in RTunproc;
      if (T(rt) = P) {
        known[T(rt)] := true;
        ins'[T(rt)] := outs'[rt];
      }
      if (known[T(rt)] = false and the
          queue from T(rt) is not empty)  {
        get_inputs(ins[T(rt)]);
        if (ins[T(rt)].fresh = true)
          ins'[T(rt)] := ins[T(rt)];
        known[T(rt)] := true;
      }
      if (known[T(rt)] = true)
        if (ins'[T(rt)] = false)
          remove rt from RTunproc;
        else if (the queue to rt is not full) {
          put_outputs(outs'[rt]);
          outs'[rt].fresh = false;
          remove rt from RTunproc;
        }
    }
  }
}
```

Figure 8: FFP process for a general Triggered SBD.

a message from $T(rt)$, proceed to decide whether rt is triggered or not, and send a message to rt if it is triggered? The reason is that a fixed order of iterating over processes in $RT(P)$ may result in deadlocks. For example, if $P_1, P_2 \in RT(P)$ and we decide to wait first for a message from $T(P_1)$ and then from $T(P_2)$, there is a deadlock in the case that P_1 is itself triggered by P_2, i.e., $T(P_1) = P_2$. This situation is illustrated in Fig. 9 (the Triggered SBD is shown to the left and the corresponding FFP to the right). Links from P_3 to P and from P_2 to P are backward links. The deadlock arises because: P_2 waits at Stage 1 for a message from P; at the same time, given the above fixed iteration order, P at Stage 2 first waits for a message from $T(P_1)$, i.e., from P_2. This happens before P can wait for a message from $T(P_2)$ to decide whether to send a message to P_2.

This deadlock is avoided in our method: Suppose P_1 is selected first in Stage 2 of P. Then, P attempts to read the trigger signal from $T(P_1) = P_2$, but finds the backward queue from P_2 to P empty, so another process in $RT(P)$ is selected. In this way, no extra dependencies are added among processes, and P eventually handles P_2 before P_1.

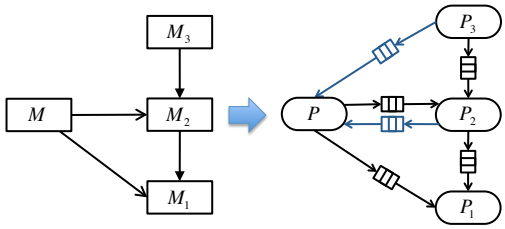

Figure 9: Potential for deadlock with a static iteration order over $RT(P)$.

Note that the deadlock *could* be avoided with a static iteration order, where P handles P_2 before P_1. However, such a static order generally depends on the topology of the diagram. In this paper, we opted for a method that guarantees absence of deadlocks while being *modular*, that is, where the code for P does not depend at all on the diagram (see also discussion in Section 6).

3.2 Semantical Preservation

Stream equivalence between a Triggered SBD G and the FFP generated by our method, denoted as F^*, can be proven in four steps.

Step 1: The Triggered SBD G is transformed to an equivalent pure SBD, denoted as G_s using the trigger elimination method proposed in [16]. Triggers in G are transformed to standard inputs in G_s, and all the blocks in G_s are fired at every synchronous step.

Step 2: The pure SBD G_s is mapped to an FFP, denoted as F_s, using the method proposed in [23], which guarantees stream equivalence between F_s and G_s, and therefore also between F_s and G.

Step 3: In this step, we transform F_s to a new FFP, denoted as F'_s, by adding backward signals (and queues if needed) and restructuring every process into three stages, as with the processes in the FFP F^*. The difference between F'_s and F^* is the following: although a process in F'_s reads the backward signals, it does not use the information; instead, it always sends messages to the output queues at every synchronous step.

The code of a process P in F'_s is shown in Fig. 10. In the case that it has a trigger, the process and its surroundings with details on stages is shown in Fig. 11. [3] The functionality of each of the stages is as follows:

- Stage 0: If P has a trigger input, it reads the input in this stage. Otherwise, P does not contain this stage. The stage is denoted as $P.S_0$ in Fig. 11.
- Stage 1: P reads inputs from $W(P)$, executes, and sends outputs to $RR(P)$ and $TO(P)$. Note that even when the trigger is false, it still reads inputs and it sends outputs (although they are the same as previous ones), the same as in F_s. P does not need to reads the inputs from $T(RT(P))$ in this stage because the output function only depends on the inputs from $W(P)$ to compute the outputs. The stage is denoted as $P.S_1$ in Fig. 11.

[3] The figure for a process without a trigger is almost the same, except that it does not have input queues from a trigger process or stage 0, and it reads inputs from Stage 1 of the writers.

```
P (inputs: ins, trigger; outputs: outs)
{
  initialize state, outs, ins' and outs';
  while (true) {
    for all i in T(RT(P)), known[i] := false;
    // Stage 0: determine trigger
    if  (T(P) is defined) {
      wait until trigger queue is not empty;
      get_inputs(trigger);
      if (T(P) in T(RT(P))) known[T(P)] := true;
    }
    else
      trigger := true;
    // Stage 1: fire and send to RB(P), where
    // RB(P) := RR(P) union B(P) union W(B(P))
    wait until no queue to RB(P) is full;
    if (trigger) {
      wait until no queue from W(P) is empty;
      get_inputs(ins[W(P)\T(P)]);
      for (every i in W(P)\T(P) s.t. i in T(RT(P)))
        known[i] := true;
      (state, outs) := M.step(state, ins[W(P)]);
    }
    put_outputs(outs[RB(P)]);
    // Stage 2: send to RT(P)
    RTunproc := RT(P);
    while (RTunproc != empty) {
      pick a process rt in RTunproc;
      if (known[T(rt)] = false and the
          queue from T(rt) is not empty)  {
        get_inputs(ins[T(rt)]);
        known[T(rt)] := true;
      }
      if (known[T(rt)] = true)
        if (the queue to rt is not full) {
          put_outputs(outs[rt]);
          remove rt from RTunproc;
        }
    }
  }
}
```

Figure 10: FFP process in F_s' generated in step 3 of the proof.

- Stage 2: P sends outputs to all the processes in $RT(P)$, as in F_s.

  `RTunproc` represents the set of all processes in $RT(P)$ that P needs to handle in Stage 2. P iterates over all processes in $RT(P)$ until all of them have been handled. A process `rt` is selected at random, and P checks whether the triggering status of `rt` is known. If not, P checks whether the backward queue from `R(rt)` contains a message and reads the message if it is true. If the triggering status for `rt` is known, P checks whether there is space available in the output queue to rt, and sends a message to the queue if it is true (no matter the triggering status is true or false), and then `rt` is removed from `RTunproc`. Note that in the case that any of the above checking returns false, P skips this iteration immediately and another process in `RTunproc` is selected. In this way, no extra dependencies are added among the processes.

  Stage 2 contains multiple sub-stages, a sub-stage for every process $RT(P)_i$ in $RT(P)$, denoted as $P.S_2.RT(P)_i$ in Fig. 11.

We next show that F_s' is stream-equivalent to F_s. For this, it suffices to prove that no process in F_s' ever deadlocks. This is because every process P in F_s' behaves identically to the corresponding process in F_s, except that P consumes a set of additional messages that it never uses. To prove that no process in F_s' deadlocks, we use the careful structuring of the code into stages, which ensures that the additional backward queues do not create any dependency cycles. It is proven by induction as follows.

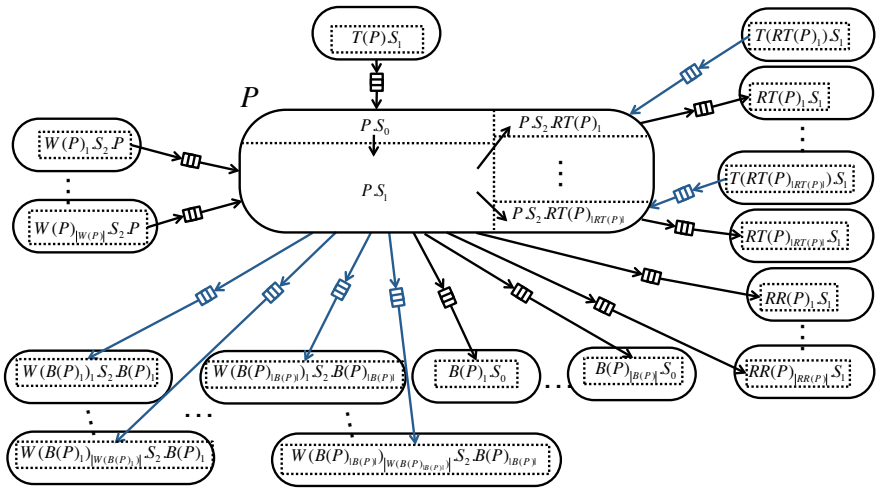

Figure 11: P and its surroundings with stage details in the FFP F'_s, in the case that P has a trigger.

- Basis: We construct an FFP, denoted as $F'_s(0)$, by removing all the additional backward queues (together with the backward signals transmitted through them) in F'_s . $F'_s(0)$ does not have any dependency cycles because it has the same dependency relations as F_s.

- Inductive step: Let $F'_s(i+1)$ be the FFP constructed by adding to the FFP $F'_s(i)$ another additional backward queue, denoted as $Q_a(i+1)$, which is from a process $T(RT(P)_k)$ to P. Specifically, it is a queue from $T(RT(P)_k).S_1$ to $P.S_2.RT(P)_k$ if we consider the stages. Below we prove by contradiction that: given that $F'_s(i)$ has no dependency cycles, $F'_s(i+1)$ has no dependency cycles either.

  Assuming that $F'_s(i+1)$ has a dependency cycle, denoted as C. Since $F'_s(i)$ has no dependency cycles, C must contain the newly added backward queue, i.e., $Q_a(i+1)$. It means that there is a path from $P.S_2.RT(P)_k$ to $T(RT(P)_k).S_1$ which does not contains $Q_a(i+1)$. Since the queue from $P.S_2.RT(P)_k$ to $RT(P)_k.S_1$ is the only output queue of $P.S_2.RT(P)_k$, C must contain a path, denoted as ϕ, from $RT(P)_k.S_1$ to $T(RT(P)_k).S_1$, which doe not contains $Q_a(i+1)$. Therefore, the path ϕ is also in $F'_s(i)$. On the other hand, there is a path in $F'_s(0)$ and therefore also in $F'_s(i)$, denoted as ϕ', from $T(RT(P)_k).S_1$ through $RT(P)_k.S_0$ to $RT(P)_k.S_1$. So $F_s(i)$ contains a cycle composed of ϕ and ϕ', which contradicts with the assumption that $F'_s(i)$ has no cycles.

By adding back all the additional backward queues one by one $F'_s(0)$, we can conclude that the resulting FFP F'_s has no dependency cycles.

Step 4: In this step, we prove that F^* is stream-equivalent to F'_s.

These two FFPs have the same structure, i.e. there is a one-to-one mapping between the processes and the FIFO queues in the two FFPs. Therefore, for a pair of corresponding processes in the two FFPs, P^* in F^* and P in F'_s, P^* has a trigger iff P has a trigger.

As explained in Section. 2.3, what we need to prove is the following: for any pair of corresponding processes in the two FFPs, the input trigger signals (if the processes have triggers) are the same at every synchronous step, and the input data are the same at the synchronous steps when the trigger signals are true.

Two facts are used to support our proof in this step:

(a) For any given pair of processes W^* and R^*, and the queue Q_{W^*,R^*} connecting them in F^*, if R^* is in the set $RT(W^*)$, W^* only sends a message to the queue $Q(W^*, R^*)$ after it receives from $T(R^*)$ a trigger signal of R^* and the value of the trigger signal is true, and R^* only reads a message from the queue after it receives from $T(R^*)$ the same trigger signal and the value of the signal is true; otherwise, W always sends to (and R^* always reads from) the queue at every synchronous step. Therefore, the message sent to the queue by W at a synchronous step is always read by R at the same synchronous step.

On the other hand, For the corresponding pair of processes W' and R', and the queue $Q_{W',R'}$ connecting them in F'_s, the process W' sends a message to the queue $Q_{W',R'}$ at every synchronous step, and R' reads a message from the queue at every synchronous step. Therefore, R at any synchronous step always reads the message from W' produced at the same synchronous step.

(b) Besides, the state variable `outs'[R*]` of W^* keeps the up-to-date output message for R^*: it is updated with the new output data produced by the output function when W^* is triggered and its fresh-bit is set to be true; once the message is sent to the queue its fresh-bit is set to be false . On the other hand, the state variable `ins'[W*]` of R^* memorizes the latest input data it reads from W^*. If R^* receives a message from W^* whose fresh-bit is false, meaning that R^* has already read the latest output data from W^*, it uses the data in `ins'[W*]` (which is the latest input data) as the input from W^* for execution.

The stream equivalence between the two FFPs can be proven by induction as follows:

- Basis: For any pair of corresponding processes in the two FFPs F^* and F', the internal states and the messages in the input queues (if any) are initialized the same.
- Inductive step: Assuming that for any pair of corresponding processes P^* and P' in the two FFPs, the followings are true before and at the k-th step:

  (1) the input trigger signals, including those from $T(P^*) \cup T(RT(P^*))$ and $T(P') \cup T(RT(P'))$, are the same at any of the steps.

  (2) For any $W(P^*)_i \in W(P^*)$ and $W(P')_i \in W(P')$, the input data from $W(P^*)_i$ (or actually from the state variables `ins'` if a message from $W(P^*)$ contains a fresh-bit whose value is false) and $W(P')_i$ are the same at the steps when the trigger signals from $T(P^*)$ and $T(P')$ (which are the same according to (1)) are true.

  (3) the internal states of P^* and P' are the same at any of the steps.

  Below we prove that (1)-(3) are also true for any pair of corresponding processes P^* in F^* and P' in F' before or at the $(k+1)$-th step. It is proven by tracing down the dependency graphs of the stages of processes, denoted as (process.stage)'s (as shown in Fig. 11), in both FFPs. Note that according to the result of Step 3, the dependency graphs of the two FFPs are DAGs, and they are the same.

  First of all, the follows are true before and at the k-th step because the two processes have the same output function:

  (4) the output trigger signals, including those to $B(P^*) \cup W(B(P^*))$ and $B(P') \cup W(B(P'))$, are the same at any of the steps.

  (5) the output messages to $RR(P^*)$ and $RR(P')$ are the same at any of the steps.

  (6) the output messages to $RT(P^*)$ and $RT(P')$ are the same at the steps when the trigger signals of the consumers are true. (Note that the values of the trigger signals are the same for the corresponding consumer processes because of (1)).

  Note that a message from P^* to a process in $R(P^*)$, which contains only a false fresh-bit, is essentially the same as a message from P' to the corresponding process in $R(P')$, which contains the same data as the last step.

  Consider a pair of corresponding (process.stage)'s in the two dependency graphs, denoted as ps^* and ps': if they are the sources of the dependency graphs, their output messages at the $(k+1)$-th step are the same since the outputs only depend on the internal states at step k, and the output data produced by some processes before or at the k-th step.

  Otherwise, given that (1)-(3) are true (and consequently (4)-(6)) for all the (process.stage)'s which are smaller (in terms of the dependency order which can be partial order) than ps^* and ps':

  - In the case that $ps^* = P^*.S_0$ and $ps' = P'.S_0$: they read the input trigger signals (which are the same), and send the signals to $P^*.S_1$ and $P'.S_1$.
  - In the case that $ps^* = P^*.S_1$ and $ps' = P'.S_1$: if the trigger signals (which come from $P^*.S_0$ and $P'.S_0$, and therefore are the same) are true, they read input data (which are the same according to the assumptions), execute (using the same output function), and consequently their states and output data are updated with

the same values; if the trigger signal is false, the states and output data in both of them stay the same. The output data are then sent to all the stages in $P^*.S_2$ and $P'.S_2$, as well as $B(P^*).S_0 \cup W(B(P^*)).S_2$ and $B(P').S_0 \cup W(B(P')).S_2$. So the input trigger signals for $B(P^*).S_0 \cup W(B(P^*)).S_2$ and $B(P').S_0 \cup W(B(P')).S_2$ are the same at the $(k+1)$-th step.

– In the case that $ps^* = P^*.S_2.RT(P^*)_i$ and $ps' = P'.S_2.RT(P')_i$: they reads the same input data from $P^*.S_1$ and $P'.S_1$ and the trigger signals $T(RT(P^*)).S_1$ and $T(RT(P')).S_1$. When the trigger signals (which are the same according to the assumptions) from $T(RT(P^*)_i).S_1$ and $T(RT(P')_i).S_1$ are true, they send outputs (which come from $P^*.S_1$ and $P'.S_1$, and therefore are the same) to $RT(P^*)_i.S_1$ and $RT(P')_i.S_1$. So the input data for $RT(P^*)_i.S_1$ and $RT(P')_i.S_1$ are the same at the $(k+1)$-th step.

By tracing down the dependency graphs, we can conclude that (1)-(3) are also true at the $(k+1)$-th step.

3.3 Communication Savings Analysis

Compared to the naive method, the communication savings achieved by the optimized method are, on the average (in bits per step)

$$\sum_{l:(W,R)\in L} P_W L_D + B^{RT}_{W,R} P^*_{W,R} L_D - (B^{RT}_{W,R}(1-P_R) + (1-B^{RT}_{W,R}) + B^{RT}_{W,R}) \cdot L_T \tag{1}$$

where: L is the set of all links in the Triggered SBD; W and R are the writer and reader blocks of a link l; P_W and P_R are the probabilities of W and R *not* being triggered at any given step; $B^{RT}_{W,R}$ is a boolean variable indicating whether R is in the set $RT(W)$ or not.

The first term of savings, $P_W L_D$, comes from the fact that, in the FFP, W only sends to R the new data which is produced when W is triggered. The second term is due to the fact that if $R \in RT(W)$, W only sends a message to R when R is triggered. Specifically, let $P^*_{W,R}(k)$ be the probability of savings due to the non-triggering of the reader R at step k. The savings are realized when the following two conditions are met: (1) R is not triggered while W is triggered at step k (the case where W is not triggered is already included in the first term of savings); (2) W is triggered at least once no later than the next time R is triggered. In this case, the output of W produced at step k need not be sent to R. $P^*_{W,R}(k)$ can be calculated as

$$P^*_{W,R}(k) = P_R \cdot (1-P_W)^2 \cdot \frac{P_W (P_R)^{N+1-k} - 1}{P_W P_R - 1} \tag{2}$$

for a finite number of steps N. As N goes to infinity, $P^*_{W,R}$ becomes independent of k and is equal to:

$$P^*_{W,R} = P_R \cdot (1-P_W)^2 \cdot \frac{1}{1 - P_W P_R} \tag{3}$$

Returning to Equation (1), L_T bits must be deducted (in the worst case) from the savings with probability $(1-P_R)$ if $R \in RT(W)$, due to the fact that an additional fresh message is sent from W to R at any step when R is triggered, and with probability 1 if $R \notin RT(W)$, since the fresh-bit needs to be sent at every step in this case. Finally, if $R \in RT(W)$, there is an additional backward signal sent from $T(R)$ to W at every step. These messages can often be merged but in the worst case will result in individual control messages whose size is approximately the same as the size of a trigger message, L_T.

4 Distribution of Timed SBDs

Since Timed SBDs are a special case of Triggered SBDs, we could simply use the mechanism described in Section 3. However, we can do better than that if we exploit the information about triggering patterns which is statically available in timed SBDs. In particular, let P be the FFP process corresponding to a block M with (period,initial phase) pair (τ_M, θ_M). Let $\tau_P = \tau_M$ and $\theta_P = \theta_M$.

```
P (inputs: ins; outputs: outs)
{
  initialize state, outs, and ins';
  k := 0;
  for (every R in R(P))
    if (theta_R < theta_P)
      put_outputs(outs[R]);
  while (true) {
    Wset, Rset := empty sets;
    for (every W in W(P))
      if (get?(W, P, k))
        add W to Wset;
    for (every R in R(P))
      if (put?(P, R, k))
        add R to Rset;
    wait until no queue from Wset is empty
      and no queue to Rset is full;
    get_inputs(ins[Wset]);
    for all i in Wset, ins'[i] := ins[i];
    (state, outs) := M.step(state, ins');
    put_outputs(outs[Rset]);
    k := k + 1;
  }
}
```

Figure 12: FFP process for a Timed SBD.

Let R be a process receiving data from P. To save communication load, P need only send a message to R when P is triggered and the message will be read by R, i.e., R will be triggered at least once before the next time P is triggered. More precisely, at its k-th triggered instant, P needs to send a message to R iff R is triggered at least once in the interval of synchronous steps between the k-th and $(k+1)$-th triggered instants of P. This is represented by the predicate $\mathsf{put?}(P, R, k)$, defined as follows:

$$\begin{aligned} &\mathsf{put?}(P, R, k) \\ &\widehat{=} \exists j : k\tau_P + \theta_P \leq j\tau_R + \theta_R < (k+1)\tau_P + \theta_P \\ &\equiv \lceil \frac{k\tau_P + \theta_P - \theta_R}{\tau_R} \rceil \tau_R + \theta_R < (k+1)\tau_P + \theta_P \end{aligned} \tag{4}$$

Note that if $\tau_P \geq \tau_R$, P sends a message to R at every instant when P is triggered.

Similarly, let W be a process sending data to P. At its k-th triggered instant, P must expect a new message from W iff W has been triggered between the $(k-1)$-th and k-th triggered instants of P. This is represented by the predicate $\mathsf{get?}(W, P, k)$, defined as follows:

$$\begin{aligned} &\mathsf{get?}(W, P, k) \\ &\widehat{=} \exists j : (k-1)\tau_P + \theta_P < j\tau_W + \theta_W \leq k\tau_P + \theta_P \\ &\equiv (\lfloor \frac{k\tau_P + \theta_P - \theta_W}{\tau_W} \rfloor - 1)\tau_P + \theta_P > (k-1)\tau_P + \theta_P \end{aligned} \tag{5}$$

Note that if $\tau_W \leq \tau_P$, P receives a message from W at every instant when P is triggered.

The above predicates are used in the code of a process P generated from a Timed SBD, and shown in Figure 12. At every iteration, P computes the sets `Wset` (resp. `Rset`) of processes that P needs to receive from (resp. send to). Then P waits for messages (resp. slots) to become available on the corresponding queues, and once this condition is satisfied, P fires. To compute `Wset` and `Rset`, P maintains a local counter `k`: notice that k does not count synchronous steps, but rather the times that P has fired. P has period τ_P and therefore fires every τ_P steps. P also maintains a state variable `ins'` which, similar to the code of Fig. 8, memorizes the last messages received at the inputs.

4.1 Semantical Preservation

Stream equivalence between a Timed SBD and the corresponding FF generated by our method (denoted as F^*) can be proven in three steps. As explained in Section. 2.3, the set of streams we consider here consists of the streams of input data sampled at the points in time where the consumers are triggered.

Step 1: A Timed SBD can be transformed to an equivalent pure SBD using the trigger elimination method of [16]. Specifically, a block M_i in a Timed SBD is transformed to a block M_i' in a pure SBD where M_i' is fired at every synchronous step.

Step 2: The pure SBD generated in step 1 can be mapped to an FFP, denoted as F', using the method proposed in [23], which guarantees stream equivalence.

Step 3: In this step, we prove that when triggered, any process in F' has the same stream of input data as the corresponding process in F^*.

F and F' have the same structure, i.e. there is a one-to-one mapping between the processes and the FIFO queues in the two FFPs. For any given pair of processes W^* and R^* in F^* that are connected by a FIFO queue, denoted as Q_{W^*,R^*}, we have their counterparts in F', denoted as W', R' and $Q_{W',R'}$. Note that the (period, initial phase) pairs (PPPs) of the corresponding processes in the two FFPs are the same. The PPPs are denoted as (τ_W, θ_W) for W' and W^*, and (τ_R, θ_R) for R' and R^*.

Next we need to prove that for any k-th triggering of R^*, the input data it reads from Q_{W^*,R^*} (or from its state varialbe `ins'`) has the same value as the input data being read at the $(k\tau_R + \theta_R)$-th synchronous step of R.

From the definition in step 1, it is obvious that the $(k\tau_R + \theta_R)$-th synchronous step of R' corresponds to the k-th triggering of the corresponding block in the original Timed SBD. Therefore, if we can prove the equivalence between the k-th triggering of R^* and the $(k\tau_R + \theta_R)$-th synchronous step of R', we can prove the equivalence between the FFP F^* and the original Timed SBD.

At the $(k\tau_R + \theta_R)$-th synchronous step, R' reads the $(k\tau_R + \theta_R)$-th data output by W'. Note that W' outputs "new" data at the synchronous steps $\{i\tau_W + \theta_W | i = 0, 1, 2, ...\}$. At other steps it outputs "old" data as in previous step. Assuming that at the $(k\tau_R + \theta_R)$-th synchronous step, R' reads the j-th "new" data produced by W', we can deduce the following relationship:

- the j-th "new" data must be produced by W' before or at the $(k\tau_R + \theta_R)$-th synchronous step, i.e. $j\tau_W + \theta_W \leq k\tau_R + \theta_R$,
- the $(j+1)$-th "new" data must be produced after the $(k\tau_R + \theta_R)$-th synchronous step, i.e. $(j+1)\tau_W + \theta_W > k\tau_R + \theta_R$.

On the other hand, assuming k-th triggering of R^* reads the l-th "new" data produced by W^*. Note that according to the definition of get?() and put?() functions, W^* puts the l-th new data to Q_{W^*,R^*} iff R^* is triggered at least once between the l-th and $(l+1)$-th triggering of W^*. The data will then be read and stored at the state varialbe `ins' [W*]` by R^* at its first triggering between the l-th and $(l+1)$-th triggering of W^*. While at any other triggering of R^* between l-th and $(l+1)$-th triggering of W^*, R^* reads from its state varialbe `ins' [W*]`, which is the l-th "new" data of W^*. Therefore, we can deduct the following relationship:

- the l-th "new" data must be produced by W^* before or at the k-th triggering of R^*, i.e. $l\tau_W + \theta_W \leq k\tau_R + \theta_R$,
- the $(l+1)$-th "new" data must be produced after the k-th triggering of R^*, i.e. $(l+1)\tau_W + \theta_W > k\tau_R + \theta_R$.

Clearly, j and l should be the same. Therefore, the k-th triggering of R^* and the $(k\tau_R + \theta_R)$-th triggering of R' reads the same data.

Based on the above reasoning, the FFP F^* is stream-equivalent to the original Timed SBD.

4.2 Communication Savings Analysis

In our method for Timed SBDs, the communication load for a link l is $max\{\tau_W, \tau_R\}^{-1} \cdot L_D$. Therefore, compared to the naive method, the savings achieved by our method are

$$\sum_{l:(W,R)\in L} (1 - \frac{1}{max\{\tau_W, \tau_R\}}) \cdot L_D \tag{6}$$

5 Discussion: an Alternative Implementation Method using Timestamps

In this section we briefly discuss an alternative method for distribution of Triggered SBDs on FFPs. We call this method the *timestamp method*. It is inspired by distributed discrete-event simulation methods [10, 17]. The idea is that a process P only sends messages when it is triggered, but it also "tags" each message m with a timestamp $k \in \mathbb{N}$ denoting the synchronous step on which m is produced. Receiving processes can use the tags to locally synchronize the messages, as well as to "fill-in the gaps". In particular, if a process P' receives two successive timestamped messages (m_1, k_1) and (m_2, k_2) on the same queue, such that $k_2 > k_1 + 1$, P' can conclude that the stream at that queue remains constant and equal to m_1 for all steps i with $k_1 \leq i < k_2$. The timestamp method reduces communication even further than the method we presented in the previous sections, which has the overhead of adding backward signals. However, as we illustrate in this section, the timestamp method may introduce a large, and in some cases unbounded, end-to-end latency in the system, as well as *bursty traffic* which results in high *jitter*. These characteristics are often unacceptable, especially in embedded applications where time predictability and low latency are essential. Therefore, we believe that the timestamp method is not appropriate for embedded applications.

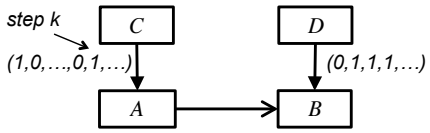

Figure 13: Example illustrating the problem of the timestamp method.

We illustrate the timestamp method and its problems with the example of Fig. 13. In this Triggered SBD, block A is triggered at step 0 and then at step k, say, $k = 10$, while block B is triggered at all steps except 0. Let P_A, P_B, P_C, P_D denote the FFP processes that blocks A, B, C, D are mapped to, respectively. Let $t_A, t_B, ...$ denote the local synchronous step counters of $P_A, P_B, ...$ According to the timestamp method, at its first iteration, $t_A = 0$, P_A sends message $(m, 0)$ to P_B, which signifies that a value m is produced by A at step 0. No messages are sent by P_A for the next $k - 1$ iterations. At iteration $t_A = k$, P_A sends message (m', k). Meanwhile, P_B does not consume m at its first iteration, $t_B = 0$, since B is not triggered at step 0. At $t_B = 1$, P_B cannot consume m either. This is because t_B is at that point greater than 0, the timestamp of m. P_B cannot know whether P_A sent a message at logical step 1 (in which case m is obsolete and should be discarded) or whether P_A will not send any message at step 1 (in which case m should be used). P_B will not know that the latter case applies until (m', k) arrives. Until then, P_B remains effectively idle. At that point, it can perform a series of computations to advance its local counter t_B to k, and emit a burst of messages at its output. Only then can P_B consume m and emit its first output message, for logical step 0. Therefore, the end-to-end latency from P_A to P_B is proportional to k, since it takes at least k iterations for P_B to emit even the first message. As k goes to infinity, the latency as well as the burstiness also go to infinity. Therefore, in the timestamp method, latency (and burstiness) cannot be bounded in general.

In comparison, our method works on this example as follows. P_A will transmit nothing to P_B at $t_A = 0$, since P_B is not triggered in that step: P_A receives this information via the backward link from P_D to P_A in the generated FFP. In subsequent steps $t_A = 1, ..., k - 1$, P_A transmits a message to P_B, even though P_A is not triggered. P_B consumes each of these messages at its own rate. The end-to-end latency in this implementation is therefore independent from k.

6 Conclusion and future work

We presented a method to optimize communication in asynchronous distributed implementations of triggered synchronous block diagrams, while preserving behavior equivalence. To the best of our knowledge, this paper is the first to address this problem.

Related work: There is a large body of research on distribution of synchronous models, and in particular synchronous languages [4]. The model of Triggered SBDs corresponds to a restricted class of synchronous programs. It

is directly inspired by tools such as Simulink[4] and SCADE.[5] SCADE can be seen as a subclass of Lustre [9]. Because we target a restricted class of synchronous models, we avoid many of the difficulties arising when considering more general models, such as the full Lustre, Signal or Esterel synchronous languages, for which there exists a wealth of techniques, e.g., see [12, 3, 21, 20, 1].

Our approach follows the one of [23] which is to map synchronous models on the FFP platform. The FFP platform makes no assumptions on clock synchronization. This has the advantage of providing implementations that are robust to various types of timing uncertainties such as clock drifts and network delays. Similar techniques are used in the design of digital circuits, in particular, latency-insensitive or elastic circuits [5, 6]. On the other hand, knowledge about the timing characteristics of the execution platform may sometimes be available, e.g., bounds on clock drifts and network delays. A number of works present implementation techniques that use such type of knowledge, e.g., see [21, 7, 2]. Other works target synchronous distributed execution platforms such as the Time-Triggered Architecture [15]: in that case, one of the main challenges is to synthesize time-triggered communication schedules so that semantics is preserved [8].

In this paper we assumed a 1-1 mapping between blocks of the synchronous model and processes of the distributed architecture. This simplifies the problem and allows focusing on semantical preservation. How to allocate functional blocks to processes is an important and difficult problem in embedded control systems, that often involves multi-criteria optimization and tradeoffs, e.g., see [22, 19, 18, 24].

While we consider the results of this paper a significant advance, we believe much remains to be done: one direction is to combine the method for general triggered diagrams and the one for timed diagrams into a single method that works for "hybrid" diagrams (those that contain both triggered and timed parts). Yet, another direction is examining alternatives in the implementation of Stage 2 of the execution of processes in the triggered method. As discussed at the end of Section 3.1, in this paper we opted for a *modular* code generation method, where the code of each process P is independent from the topology of the diagram. The downside is that execution time may increase, since a process in $RT(P)$ may be considered multiple times until it is marked as handled. On the other hand, choosing a fixed iteration order disregarding topology may result in deadlocks. It would be interesting to devise a method that uses a fixed order yet is guaranteed to avoid deadlocks. That method would most likely have to carefully choose the order by analyzing the block dependencies of the entire diagram. This approach would improve run-time performance, at the expense of compile time, but also at the expense of modularity. Devising a method that is both modular and has also good run-time performance is an interesting challenge.

References

[1] D. Baudisch, J. Brandt, and K. Schneider. Dependency-driven distribution of synchronous programs. In *DIPES/BICC*, 2010.

[2] A. Benveniste, A. Bouillard, and P. Caspi. A unifying view of loosely time-triggered architectures. In L. P. Carloni and S. Tripakis, editors, *EMSOFT*, pages 189–198. ACM, 2010.

[3] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation. *Information and Computation*, 163:125–171, 2000.

[4] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proc. IEEE*, 91(1):64–83, January 2003.

[5] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(9), 2001.

[6] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. Elastic circuits. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28:1437–1455, October 2009.

[4] http://www.mathworks.com/products/simulink/
[5] http://www.esterel-technologies.com/products/scade-suite/

[7] P. Caspi and A. Benveniste. Time-robust discrete control over networked loosely time-triggered architectures. In *CDC*, pages 3595–3600. IEEE, 2008.

[8] 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. In *LCTES'03*, 2003.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Symp. POPL*. ACM, 1987.

[10] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Soft. Eng.*, SE-5(5):440 – 452, September 1979.

[11] S. Edwards and E. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48:21–42(22), July 2003.

[12] T. Gautier and P. Le Guernic. Code generation in the SACRES project. In *Safety-critical Systems Symposium, SSS'99*, pages 127–149. Springer, 1999.

[13] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74, Proceedings of IFIP Congress 74*. North-Holland, 1974.

[14] Z. Kohavi. *Switching and finite automata theory, 2nd ed.* McGraw-Hill, 1978.

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

[16] R. Lublinerman and S. Tripakis. Modular code generation from triggered and timed block diagrams. In *RTAS*, 2008.

[17] J. Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18:39–65, March 1986.

[18] P. Pop, P. Eles, and Z. Peng. Schedulability-driven frame packing for multicluster distributed embedded systems. *Trans. on Embedded Computing Sys.*, 4(1):112–140, 2005.

[19] P. Pop, P. Eles, Z. Peng, and T. Pop. Scheduling and mapping in an incremental design methodology for distributed real-time embedded systems. *IEEE Trans. VLSI Syst.*, 12(8):793–811, 2004.

[20] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, 2006.

[21] J. Romberg and A. Bauer. Loose synchronization of event-triggered networks for distribution of synchronous programs. In *EMSOFT'04*, pages 193–202. ACM, 2004.

[22] M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Systems*, 14(3):219–250, 1998.

[23] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. Di Natale. Implementing synchronous models on loosely time triggered architectures. *IEEE Trans. Comput.*, 57(10):1300–1314, 2008.

[24] W. Zheng, Q. Zhu, M. Di Natale, and A.S. Vincentelli. Definition of task allocation and priority assignment in hard real-time distributed systems. In *RTSS*, pages 161 –170, 2007.