Efficient Distribution of Triggered Synchronous Block Diagrams on Asynchronous Platforms

Yang Yang (UC Berkeley), Stavros Tripakis (UC Berkeley and Aalto), and Alberto Sangiovanni-Vincentelli (UC Berkeley)

Abstract—As the complexity of embedded systems rapidly increases in terms of both scale and functionality, there has been a strong interest in design languages and methodologies that facilitate the use of formal methods. These languages and methodologies are mostly based on a synchronous paradigm that, while satisfying 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 propose an approach for efficient distribution of Triggered Synchronous Block Diagrams (SBDs) on asynchronous platforms while preserving the correct semantics. Compared to previous work that utilizes trigger elimination, our approach aims to reduce the unnecessary communication overhead and thus improve the efficiency of the implementation. We consider both general Triggered SBDs where the values of triggers are dynamically computed, as well as Timed SBDs where triggers are statically known and usually specified by (period, initial phase) pairs.

I. INTRODUCTION

Many of the modern embedded systems involve complex interactions between the physical environment and the electronic control systems (i.e. they are cyber-physical systems), and usually consist of heterogeneous spatially-distributed subsystems exchanging information over networks. Cars, airplanes, and power distribution grids are well-known examples. These systems are difficult to design and to verify given these characteristics, and there is a great deal of interest in developing formal 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 embedded systems, including SCADEn, Lustre [5], 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) [6] 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 systems given the predictability of synchronous designs.

However, the implementation of synchrony on distributed (and typically asynchronous) execution platforms incurs some overhead. We would like to maintain the benefits of synchronous designs while reducing the implementation overhead. Indeed, complex embedded systems being addressed today do pose significant pressure on verification and on the efficiency of the implementation platform. In this work, we are interested in methods that starting with a synchronous abstract design, derive an implementation that is both semantically equivalent and does not require adherence to the synchronous paradigm. Semantic equivalence guarantees any property valid in the abstraction is also valid in the implementation, while not adherence to synchronicity minimizes the 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 for cyber-physical systems.

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” in a certain order within a synchronous step, 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 trigger 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 Timed SBDs, a special case of Triggered SBDs, where triggering patterns are known statically (“at compile time”).

This work was partially supported by the NSF (awards #1329759 and #1139138) and by the Industrial Cyber-Physical Systems (iCyPhy) Research Center (supported by IBM and United Technologies).

1http://www.esterel-technologies.com/products/scade-suite/
2http://www.mathworks.com/products/simulink/
The problem we address in this paper is semantics-preserving and communication-efficient distribution of Triggered SBDs on asynchronous execution platforms. In particular, given a design specification described as a Triggered SBD, we address how to map it to a distributed and 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.

In [17], an approach is proposed for the semantics-preserving distribution of “pure” SBD models, where all blocks fire at every synchronous step. One of the key ideas is to define finite FIFO platforms (FFPs) to capture the distributed asynchronous execution platforms. An FFP is similar to a Kahn Process Network (KPN) [8], with the difference that while in a KPN queues are unbounded, in an FFP they are of fixed and finite sizes. 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) [17]. 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 speeds of the local clocks of distributed processes, hence the characterization asynchronous. We also heavily leverage the concept of FFP in our work.

Triggered SBDs can be translated into pure SBDs by a trigger elimination procedure that transforms triggers into standard inputs [11]. This provides a straightforward way to utilize the approach in [17] for Triggered SBDs. However, this simple method based on trigger elimination often results in unnecessary communication overhead, as a block always sends output messages even when its trigger is false. More specifically, when the trigger of a block is false, the block does not fire and its outputs remain the same values as in the previous step, so sending those outputs to downstream blocks is really unnecessary.

In this paper, we address Triggered SBDs directly and propose an efficient distribution method that aims to reduce the communication overhead, when compared with the trigger elimination based distribution method. We also study Timed SBDs as a special case, for which even more efficient implementations can be achieved. Our distribution method aims to optimize communication along the following two directions: first, data messages are not sent to processes that are not triggered; second, a process that 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. We believe reducing such communication overhead is important for the performance and robustness of many types of embedded systems, especially when the communication is expensive (e.g. in wireless applications where the channel capacity is limited, or in systems where energy savings are critical).

A. Motivating Example

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 trigger signals from standard inter-block communication signals, and we usually draw triggering signals as incoming to the top of a block.

![Fig. 1. A Triggered SBD.](image1)

Fig. 2. The FFP system resulting from the Triggered SBD of Fig. 1 after trigger elimination [11] and distribution [17].

In the trigger elimination based distribution method, after applying the trigger elimination method of [11] and the distribution method of [17], 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:

```plaintext
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 \( \text{true} \), \( P \) uses the output function of \( M.\text{step}() \), to update the outputs and the state. When the trigger is \( \text{false} \), 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 is waiting 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 synchronous 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 synchronous 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 as 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 trigger elimination based implementation is \( 6L_T + 7L_D \), measured in bits per synchronous step.

In the optimized distribution 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 synchronous step. Therefore, in the optimized implementation, at most 4 data messages are transmitted in each synchronous 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 \( 4/7 \approx 57\% \), 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 trigger elimination based implementation, process \( W \) would send a message to \( R \) at every synchronous step. \( R \) would execute to consume these messages and then do nothing 9 out of 10 times. 9/10 of those messages are unnecessary, and are eliminated by our method.

\[
\begin{array}{cc}
W & R \\
(1, 0) & (10, 0)
\end{array}
\]

Fig. 3. A Timed SBD.

II. BACKGROUND: TRIGGERED SBDs AND FFPs

A. 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 \( M' \) (in this case we say that \( M' \) has a trigger). A block can have zero or one incoming trigger. 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 [9]. We say that a block is “Moore” if its output function only depends on its state, but 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 \( \mathbb{N} = \{0, 1, 2, \ldots\} \) 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. 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 \( \tau \) represents a period and \( \theta \) represents an initial phase. For example, the pair \((2, 1)\) generates the stream false true false true ···. 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 distribution methods that we present in this paper, as well as the methods proposed in [17], are agnostic of the internals of blocks, that is, blocks are treated as black boxes whose internal state machines are not known.

### B. FFPs

We model the distributed, asynchronous execution platform as a Finite FIFO Platform (FFP) [17]. 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) [8], with the difference that in a KPN the queues are unbounded. Another difference is that, 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, respectively. An example of an FFP process is a process \( P \) executing the pseudo code shown in Section I-A (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. In fact, the distribution methods that we present in Sections III and IV rely on different FFP process structures and achieve better communication efficiency.

### C. 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 the code that each FFP process executes. The topology synthesis is straightforward, since we assume a one-to-one mapping of blocks to processes, as in [17]. In an FFP, a stream is essentially the sequence of values that are written in a given queue. In the trigger elimination based 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 our 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 show that the resulting FFPs are deadlock-free [17].

As mentioned in Section I, the straightforward, simple solution to the distribution problem is to combine the trigger elimination procedure of [11] with the distribution method of [17]. This method creates communication overhead, however, as illustrated by the examples of Section I-A. In the following sections we propose alternative implementations that eliminate this overhead while preserving the semantics.

### III. DISTRIBUTION OF GENERAL TRIGGERED SBDs

We first introduce some notations 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 \( t \). 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 to \( M \) is denoted as \( W(M) \).
- \( B(M) \) 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 into two disjoint subsets: \( RR(M) \), containing all blocks in \( R(M) \) that either have no trigger or have a trigger but are already in \( W(B(M)) \);

More generally, triggers in timed SBDs could be specified by firing time automata (FTA) [11]. Our distribution method can be directly extended to FTA, but for simplicity, we limit our discussion to PPPs.

\( ^3 \) More generally, triggers in timed SBDs could be specified by firing time automata (FTA) [11]. Our distribution method can be directly extended to FTA, but for simplicity, we limit our discussion to PPPs.

\( ^4 \) For a set \( W \), \( |W| \) denotes its cardinality. We use \( W_1, W_2 \), etc. to enumerate and denote its elements, so that \( W(M) = \{W(M)_1, \ldots, W(M)_{|W(M)|}\} \). Also, we define \( W(X) = \bigcup_{Q \in X} W(Q) \), for a set of processes \( X \).
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 either \(R(M)\) or \(B(M)\) (this would result in cyclic diagrams), but it may be an element of \(W(M)\).

A. Mapping Triggered SBDs on FFPs

A Triggered SBD is mapped onto an FFP in the following way. Every block \(M\) in the Triggered SBD is mapped to an FFP process \(P\). Every link from a block \(M\) to another block \(M'\) 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 [17]. In particular, if \(M\) is not Moore, a queue of size 1 is sufficient; otherwise a queue of size 2 is sufficient (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 the notation \(T(M), W(M)\), etc. for blocks, we introduce notation \(T(P), W(P)\), etc. for processes. 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 some 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, where \(P_3\) sends triggering information about \(P_2\) to \(P_1\). 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 processes in \(W(B(P))\).

We should note that additional queues are introduced by the optimized implementation, only if they do not exist after mapping links in the Triggered SBD to FIFO queues in the FFP. For example, the process \(T(P)\) may also be in \(T(RT(P))\). This is the case in Fig. 2, where \(T(P_{11}) = T(RT(P_3))\). Since there is already a queue from \(P_1\) to \(P_1\), 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, but are not needed for \(P_1\) to compute its outputs.

The code that each FFP process \(P\) executes is shown in Fig. 8. It follows the same general scheme as the trigger elimination based implementation described in Section 1-A: initialization of state variables, followed by execution of an infinite loop. Note that the code for the optimized implementation adds a small computation overhead, however we think this is usually negligible compared with the computation time for processing the data. 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 \( \text{trigger} \) is set to \( \text{true} \). Otherwise, \( P \) needs to consume a message from the input queue \( \text{trigger} \) 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 if and only if the trigger is \( \text{true} \), 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 the 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 in Fig. 7, \( P_1 \) will send a message to \( P_3 \) at Stage 1, since \( P_3 \in RR(P_1) \). Then, \( P_3 \) can execute and send back to \( P_1 \) the trigger information about \( P_2 \) via the backward queue. Once \( P_1 \) has this information, it can decide whether a message needs to be sent to \( P_3 \). If so, this will happen at Stage 2 of \( P_1 \). One can see how this careful ordering avoids dependency cycles and deadlocks in this example. More complicated cases exist, for instance, where \( P_3 \) has a trigger and belongs not to \( RR(P_1) \) but to \( RT(P_1) \). The proof of semantical preservation described in Section III-B argues how these cases are also handled correctly by our method.

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

\( P \) maintains state variables \( \text{ins'} \) and \( \text{outs'} \). For each input queue from a process \( i \), \( \text{ins'}[i] \) memorizes the last data message received from the queue. 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, \( \text{outs'} \) memorizes the latest message that \( P \) produced for that queue. Note that \( \text{get_inputs()} \) and \( \text{put_outputs()} \) use only \( \text{ins} \) and \( \text{outs} \) and do not affect \( \text{ins'} \) and \( \text{outs'} \).

Messages in \( \text{outs'} \) contain an extra Boolean flag \( \text{fresh} \), indicating that the corresponding output is newly produced, as opposed to the one that has already been sent. Initially all output data are fresh: this is because the initial data may need to be sent before the first time \( P \) is triggered. When \( \text{put_outputs()} \) takes \( \text{outs'} \) as argument, it first checks the \( \text{fresh} \) flag of each message: if it is \( \text{true} \), 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 that each message sent by \( \text{put_outputs()} \) contains all the information that must be transmitted from one process to another within a synchronous step. Such a message may therefore include, for example, both a trigger and a data part.

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

\( \text{RTunproc} \) represents the set of all processes in \( RT(P) \) that \( P \) needs to consider in Stage 2. For each \( rt \in RT(P) \),
This happens before at the same time, given the above fixed iteration order, signal from illustrated in Fig. 9 (the Triggered SBD is shown to the left because: absence of deadlocks while being diagram. In this paper, we opted for a method that guarantees such a static order generally depends on the topology of the and from to. If the trigger is true, P sends a message if space is available in the corresponding queue. In those 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 ∈ RT(P), wait for 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, assuming P₁, P2 ∈ RT(P) and we decide to wait first for a message from T(P₁) and then a message from T(P₂), there will be a deadlock if P₁ is itself triggered by P₂, i.e. T(P₁) = P₂. This situation is illustrated in Fig. 9 (the Triggered SBD is shown to the left and the corresponding FFP to the right). Links from P₃ to P and from P₂ to P are backward links. The deadlock happens because: P₂ waits at Stage 1 for a message from P; while at the same time, given the above fixed iteration order, P at Stage 2 first waits for a message from T(P₁), i.e. from P₃. This happens before P can wait for a message from T(P₂) to decide whether to send a message to P₂.

This deadlock is avoided in our method. Assuming P₁ is selected first in Stage 2 of P, P attempts to read the trigger signal from T(P₁) = P₂, but finds the backward queue from P₂ 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 P₂ before P₁.

![Diagram](image)

Fig. 9. Potential deadlock with a static iteration order over RT(P).

Note that the deadlock could be avoided with a static iteration order, where P handles P₂ before P₁. 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 VI).

**B. 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. Due to space limitation, we present only a sketch of these steps here. The full proof is shown in the technical report [18].

Step 1 and 2: G is transformed to an equivalent pure SBD Gₛ using the trigger elimination method from [11]. Gₛ is then mapped to an FFP Fₛ, using the method proposed in [17], which guarantees stream equivalence between Fₛ and Gₛ, and therefore also between Fₛ and G.

Step 3: We transform Fₛ to a new FFP, denoted as Fₛ′, by adding backward signals (and queues if needed), and restructuring every process in Fₛ into three stages, as with processes in F⁺. The difference between Fₛ′ and F⁺ is the following: although a process in Fₛ′ reads the backward signals, it does not use the information; instead, it always sends messages to all the output queues at every step.

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

Step 4: We prove that F⁺ is stream-equivalent to Fₛ′. 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. Consider a pair of corresponding processes in the two FFPs, P⁺ in F⁺ and P in Fₛ′. Because the structure of the two FFPs is the same, P⁺ has a trigger if and only if P has a trigger. Also, trigger messages are transmitted at every step and never omitted. Based on this, we use induction to show that the trigger signals of P⁺ and P have the same value at every step, and that the input data that are read by P⁺ and P are the same at every step when the trigger signals are true. The following facts are used in this proof: (1) state variables ins’ and outs’ of P⁺ memorize the latest inputs and outputs; (2) for every process W⁺ ∈ W(P⁺), W⁺ sends a message to P⁺ at a given step if and only if P⁺ reads the message from W⁺ at the same step. From (1) and (2) we can derive that P⁺ always gets the up-to-date inputs, either from a message from W⁺, or from its state variable ins’ when W⁺ sends a message with a fresh-bit being false.

**C. Communication Savings Analysis**

Compared to the trigger elimination based distribution method, the communication savings achieved by our distribution method are, on the average (in bits per synchronous step), at least as follows.

\[
\sum_{E(W,R) \in L} P_W L_D + B_{W,R}^{RT} P_{W,R} L_D + B_{W,R}^{RT} (1 - P_R) + (1 - B_{W,R}^{RT}) B_{W,R}^{RT} L_T
\]

where L is the set of data 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, respectively; B_{W,R}^{RT} is a Boolean variable indicating
whether \( R \) is in \( RT(W) \) or not; and \( L_T \) and \( L_D \) are the lengths of trigger and data messages, respectively. The size of a control message is approximately the same a trigger message.

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

\[
P^*_W, R(k) = P_R (1 - P_W) \left( P_{R}^{N-k} + \sum_{i=0}^{N-k-1} (1 - P_R) P^i_P (1 - P^i_W) \right)
\]

(2)

where \( N \) is the number of steps a system runs. As \( N \) goes to infinity, \( P^*_W, R \) becomes independent of \( k \) and is equal to

\[
P^*_W, R = \frac{P_R (1 - P_W)^2}{1 - P_RW}
\]

(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-bit is sent from \( W \) to \( R \) at any step when \( R \) is triggered; and with probability \( 1 - (1 - P_R) \) if \( R \not\in RT(W) \), 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. Note that in most systems, \( L_D \) is much larger than \( L_T \), therefore our savings could be significant. Moreover, some of the messages are often merged in our distribution method (e.g. a fresh-bit whose value is \( \text{true} \) is always merged with the data), which will produce even more savings than what was represented in Equation (1).

IV. DISTRIBUTION OF TIMED SBDs

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

Let \( R \) be a process receiving data from \( P \). To save communication load, \( P \) only needs to send a message to \( R \) when \( P \) is triggered \( and \) the message is read by \( R \), i.e. \( R \) is 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 \( R \) if and only if \( R \) is triggered at least once within the interval between the \( k \)-th and \( (k+1) \)-th triggered instants of \( P \). This is represented by the predicate \( \text{put?}(P, R, k) \), defined as follows:

\[
\text{put?}(P, R, k) \equiv \exists j: k \tau_P + \theta_P \leq j \tau_R + \theta_R < (k + 1) \tau_P + \theta_P \]

(4)

\[= \left[ k \tau_P + \theta_P - \theta_R \over \tau_R \right] \tau_R + \theta_R < (k + 1) \tau_P + \theta_P \]

where the \( j \)-th triggered instant of \( R \) is bounded between the \( k \)-th and the \( (k+1) \)-th instants of \( P \).

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

\[
\text{get?}(W, P, k) \equiv \exists j: -(k-1) \tau_W + \theta_W < j \tau_P + \theta_P \leq k \tau_P + \theta_P \]

\[= \left[ k \tau_P + \theta_P - \theta_W \over \tau_W \right] \tau_W + \theta_W > (k - 1) \tau_P + \theta_P \]

(5)

where the \( j \)-th triggered instant of \( W \) is bounded between the \( (k-1) \)-th and the \( k \)-th instants of \( P \).

The above predicates are used in the code of a process \( P \) generated from a Timed SBD, as shown in Fig. 10. At every iteration, \( P \) computes the sets \( \text{Wset} \left(\text{Rset}\right) \) of processes that \( P \) needs to receive from (send to). Then \( P \) waits for messages (slots) to become available on the corresponding queues before it fires. To compute \( \text{Wset} \) and \( \text{Rset} \), \( P \) maintains a local counter \( k \): notice that \( k \) does not count synchronous steps, but rather the times that \( P \) has fired. \( P \) has period \( \tau_P \) and therefore fires every \( \tau_P \) steps. \( P \) also maintains a state variable \( \text{ins}' \) which, similar to the code of Fig. 8, memorizes the last messages received at the inputs. The distribution method guarantees stream equivalence while mapping a Timed SBD to an FFP. The proof is shown in the technical report [18].

In our method for Timed SBDs, the communication load for a link \( l \) is max \( \{\tau_W, \tau_R\}^{-1} \cdot L_D \). Compared to the trigger
elimination based method, our method achieves a saving of
\[
\sum_{l_i(l_w,r) \in L} \left( 1 - \frac{1}{\max \{\tau_w, \tau_r \}} \right) L_D
\]  

(6)

V. CASE STUDIES
A. Communication Savings of Multi-Mode Models

The communication savings of a two-mode model shown in Fig. 1 is discussed in Section I-A. In this section, we extend it to a more general multi-mode model and analyze the communication savings when applying our distribution method. Specifically, a $k$-mode model is a special type of Triggered SBDs. It consists of $k$ sets of communicating blocks denoted as sets $M_1$ to $M_k$, and a set of blocks for mode control (denoted as $M_0$) that triggers only one of the $k$ sets at any given time. Note that there might be communications between different sets of blocks.

Let $L_T$ and $L_D$ denote the message lengths for trigger/control messages and data messages, respectively. When distributing the model by the trigger elimination based method, the average message load per synchronous step is:

\[
k \cdot L_T + \sum_{i=0}^{k} C_{M_i} L_D + \sum_{i=1}^{k} \sum_{j=1}^{k} C_{M_i,M_j} L_D
\]

(7)

where $C_{M_i}$ denotes the number of messages between the blocks in the set $M_i$ at a synchronous step, and $C_{M_i,M_j}$ denotes the number of messages between the blocks in $M_i$ and $M_j$.

On the other hand, when applying our distribution method, the average message load per synchronous step is as follows:

\[
(k + k') \cdot L_T + C_{M_0} L_D + \sum_{i=1}^{k} P_{M_i} C_{M_i} L_D
\]

\[\quad + \sum_{i=1}^{k} \sum_{j=1}^{k} P_{M_i,M_j} C_{M_i,M_j} L_D
\]

(8)

where $P_{M_i}$ denotes the probability of the set $M_i$ being triggered, and $P_{M_i,M_j}$ denotes the probability that $M_j$ is triggered at a synchronous step and $M_i$ is triggered at the step before (i.e., mode switch from $M_i$ to $M_j$). $k'$ is the number of additional backward trigger messages.

In the case that the data messages are much longer than trigger/control messages, and that the mode switch happens sporadically, the communication saving ratio accomplished by our method can be approximated to

\[
\frac{\sum_{i=1}^{k} C_{M_i} - \sum_{i=1}^{k} P_{M_i} C_{M_i}}{\sum_{i=0}^{k} C_{M_i}} \geq \frac{\sum_{i=1}^{k} C_{M_i} - \max_{i=1}^{k} \{C_{M_i}\}}{\sum_{i=0}^{k} C_{M_i}}
\]

In the case that the number of messages in each set is close to each other, the communication saving ratio approximates to

\[
k - \frac{1}{k + 1}
\]

(9)

Intuitively, the savings are due to the fact that only the data messages in set $M_0$ and the set that is triggered by $M_0$ are transmitted at a synchronous step in our approach.

B. Savings of Randomly Generated Triggered SBDs

Furthermore, to show the effectiveness of our distribution method for Triggered SBDs in general case, we conducted experiments with randomly generated Triggered SBDs. We use TGFF\(^3\) to generate random directed acyclic graphs consisted of blocks and links, and randomly pick some of the links as trigger links. We then assign a probability of not being triggered to every block that has a trigger. Specifically, the probability is assigned as a uniformly distributed random number in the range $[0,2x]$, where $x$ is the expectation of the random number and should be no larger than 0.5. The communication savings of the randomly generated Triggered SBDs can be calculated using Equation (1).

We generated Triggered SBDs with the number of blocks ranging from 100 to 1000. The experiment result in Fig. 11 shows the average communication savings for those Triggered SBDs when the average probability of blocks not being triggered goes from 0.1 to 0.5. We picked four communication protocols, CAN bus, ZigBee, Wi-Fi and TTP/C, which have different message overheads and maximum data payloads (and therefore result in different $L_D$ and $L_T$). The communication saving ratios achieved by our distribution method increase approximately in a linear relation with the probability of blocks not being triggered, and the saving ratios for the protocols Wi-Fi and TTP/C are larger than CAN bus and ZigBee due to the fact that the former two groups can have bigger ratio of $L_D$ to $L_T$ than the latter two.

\[\text{Fig. 11. Average Communication Savings with Different Probability of Blocks Not Being Triggered.}\]

We also conducted another experiment to show the impact on communication savings when the numbers of trigger links in Triggered SBDs are different. After a random graph is generated by TGFF, we randomly pick some of the non-source blocks to be ones with trigger inputs. Specifically, a non-source block is picked with a probability of $p$, and if a block is picked, one of its input links is randomly chosen as the trigger link. Therefore, the expected percentage of non-source blocks that have triggers is $p$. We generated such Triggered SBDs with the number of blocks ranging from 100 to 1000, and $p$ ranging from 0.1 to 1. The probability of not being triggered is assigned as 0.5 to every block with a trigger. The experiment

\[^3\text{http://ziyang.eecs.umich.edu/~dickrp/tgff/}\]
result in Fig. 12 shows the average communication saving ratios increase approximately linearly with the percentage of non-source blocks that have triggers (for ZigBee and CAN, the increasing trends saturate when $p$ gets larger than 0.9).

![Graph: Communication Savings](image)

Fig. 12. Average Communication Savings with Different Percentage of Non-Source Blocks with Triggers.

VI. CONCLUSIONS, RELATED, AND FUTURE WORK

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

There is a large body of research on distribution of synchronous control systems (e.g., see [15]), and in particular synchronous languages (e.g., [1]). The model of Triggered SBDs corresponds to a restricted class of synchronous programs. It is directly inspired by tools such as Simulink and SCADE. SCADE can be seen as a subclass of Lustre [5]. Because we target a restricted class of synchronous models, we avoid many of the difficulties encountered 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 [7], [14].

Our approach follows the one of [17] which maps pure SBDs 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 [2], [3]. Knowledge about delays, however, can lead to even more optimized implementations: this different problem is studied in [12]. Other works target synchronous distributed execution platforms such as the Time-Triggered Architecture [10]. In that case, one of the main challenges is to synthesize time-triggered communication schedules so that semantics is preserved [4].

In this paper we assumed a one-to-one 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 [16], [13].

Regarding future work, one direction is to combine the method for general triggered diagrams and the one for timed diagrams into an integrated method that works for diagrams that contain both triggered and timed parts. Another direction is to examine alternatives for the Stage 2 implementation of the process executions in the triggered method. As discussed in Section III-A, 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. Furthermore, 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 guarantees 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 and modularity. Devising a method that is both modular and has good run-time performance is an interesting challenge. Finally, the results obtained in this paper analytically and by simulation need to be validated on real case studies.

REFERENCES


