Modular Code Generation from Triggered and Timed Block Diagrams

Roberto Lublinerman*
Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA
rluble@psu.edu

Stavros Tripakis
Cadence Research Laboratories
2150 Shattuck Avenue, 10th Floor
Berkeley, CA 94704, USA
tripakis@cadence.com

Abstract

In previous work we have shown how modular code can be automatically generated from a synchronous block diagram notation where all blocks fire at all times. Here, we extend this work to triggered and timed diagrams, where some blocks fire only when their trigger is true, or at statically specified times. We show that, although triggers can be eliminated, this is not desirable since it destroys modularity and may also result in rejecting some diagrams that could be accepted. To avoid this we propose a modular code generation method that directly accounts for triggers. We also propose methods specialized to timed diagrams. Although timed diagrams are special cases of triggered diagrams, treating them directly allows us to obtain efficient code. We achieve this by enriching the interface of a macro block with firing time information and using this information to avoid firing the block unnecessarily. Existing firing time representations are generally conservative, in the sense that they cannot represent the exact set of firing times of a macro block, but a super-set. To remedy this, we devise a novel and accurate (exact) representation. This representation uses finite automata and is amenable to algebraic manipulation and generation of efficient code.

1 Introduction

Block diagrams are a popular graphical notation, implemented in a number of successful commercial products such as Simulink from The MathWorks[1] and SCADE from Esterel Technologies[2]. These notations and tools are used to design embedded software in multiple application domains and are especially widespread in the automotive and avionics domains. Automatic generation of code that implements the semantics of such diagrams is useful in different contexts, from simulation, to *model-based* development where embedded software is generated automatically or semi-automatically from high-level reference models.

To master complexity, but also to address intellectual property (IP) issues, designs are built in a *modular* manner. In block diagrams, modularity manifests as *hierarchy*, where a diagram of *atomic* blocks can be encapsulated into a *macro* block, which itself can be connected with other blocks and further encapsulated.

In such a context, *modular code generation* becomes a critical issue. By modular we mean two things. First, code for a macro block should be generated *independently from context*, that is, without knowing where (in which diagrams) this block is going to be used. Second, the macro block should have *minimal knowledge* about its sub-blocks. Ideally, sub-blocks should be seen as "black boxes" supplied with some interface information. The second requirement is very important for IP issues as explained above.

In this paper we are particularly interested in synchronous block diagrams. Current code generation practice for such diagrams is not modular: typically the diagram is *flattened*, that is, hierarchy is removed and only atomic blocks are left. Then a dependency analysis is performed to check for dependency cycles within a synchronous instant: if there are none, *static* code can be generated by executing blocks in any order that respects the dependencies. Clearly, flattening destroys modularity and results in IP issues. It also impacts performance since all methods compute on the entire flat diagram which can be very large. Moreover, the hierarchical structure of the diagram is not preserved in the code, which makes the code difficult to read and modify.

Turning the above method to a modular method that avoids flattening is not straightforward. To illustrate the problem, consider the example shown in Figure 1 (this example is not new [12]). To the left, a macro block P is shown containing sub-blocks A and B. We want to generate code for P in a modular way, without knowing how P

*This work was performed while the author was on an internship at Cadence Research Laboratories.
[1]www.mathworks.com/products/simulink/
[2]www.esterel-technologies.com/products/scade-suite/

is going to be connected. The code for P should execute sub-blocks A and B, once each. In what order should A and B be executed? If we choose to execute A before B, we find that when we connect P as shown to the right of Figure 1, we have a problem: A needs the output of B to execute, but B is to be called only after A. If we execute A after B, then we have a problem when we connect P as shown in the middle. The point is that no static execution order is correct for all possible embeddings of P.

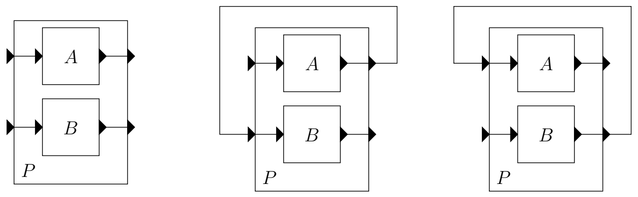


Figure 1. A hierarchical block diagram (left) and two possible ways to connect the macro block P (middle and right)

In [9] we proposed a general solution to this problem. The main idea is to generate, for a given block, not just one "monolithic" piece of sequential code that computes the outputs (and updates the state, if any) from the inputs, but a *set of interface functions*, each computing part of the block's outputs or updating its internal state. A *set of dependencies between these functions* is also exported: these specify the correct usage of the interface, that is, the order in which the functions should be called. For instance, in the example of Figure 1, we would generate not just one but two functions for P: one that executes sub-block A and another one that executes B. The two functions are independent in this case, thus they can be called in any order, depending on the context. If P is connected as shown in the middle, then A will be called before B. If P is connected as shown to the right, then B will be called before A.

The main limitation of our previous work [9] is that it considers a purely synchronous block diagram model, where all blocks are "fired" at every instant. In this paper we study two important extensions. First, we allow blocks to be *triggered* by Boolean signals produced from other blocks in the diagram (or directly from an input). This feature allows us to build *multi-rate* designs where different parts of the diagram are fired at different times. Second, we consider *timed* diagrams, where blocks are annotated with a specification of the times the block is to fire (*firing times*).

The first contribution of this paper is to show how the methods proposed in [9] can be extended to work with triggered and timed diagrams. As in [9], we equip each block with an interface consisting of a set of functions plus a dependency graph encoding the order in which these functions should be called.

For triggered blocks, an extra node is added to the graph, encoding the dependency of all interface functions of the block upon the trigger. For timed diagrams, a *firing time specification* (FTS) is added to the interface and is used to generate efficient code. Indeed, timed diagrams are special cases of triggered diagrams, in the sense that firing times are statically-defined triggers (i.e., we know at compile-time when they are going to fire). We can exploit this static information to generate code that fires a block only when necessary. We can also exploit this information in order to improve the dependency analysis. We propose an *activity analysis* step which allows us to discover that some dependencies between blocks are false, that is, these blocks are never active at the same time. False dependencies can be eliminated so that more diagrams are accepted.

Additional contributions of this paper are:

1. A method to *eliminate triggers* while preserving the semantics of a diagram. This shows that triggers do not add expressiveness to standard block diagrams. However, although triggers can be eliminated, this is not desirable for purposes of code generation, since it destroys modularity. To avoid this we also propose a modular code generation method that directly accounts for triggers.

2. A *novel representation of FTSs in terms of finite automata*, and a set of procedures to compose, multiply and divide these automata, for purposes of computing FTSs. This representation improves over the standard *(period, phase)* representation used in Simulink and other tools, since it allows to capture precisely the instants a block needs to fire, thus avoiding to fire a block unnecessarily (i.e., when none of its sub-blocks needs to fire).

2 Related Work

Code generation for notations with synchronous semantics has received great attention (e.g., see [2]). Modular code generation, however, has been relatively much less studied and "a unified treatment remains a research topic" as stated in [2].

A unifying approach has been proposed in our previous work [9]. Using this approach, modularity becomes a *quantifiable* notion, that can be measured by the size of the interface of a block (i.e., the number of interface functions and their dependencies): the smaller the interface, the more modular it is. This approach also clarifies the trade-off between modularity and *reusability* (i.e., the ability to embed a block in any context, without creating false dependencies) and presents different methods to explore this trade-off, among which: the so-called "dynamic" method that achieves maximal reusability with the minimal number of interface functions (at most $n+1$ where n is the number of outputs of the block); and the so-called "step-get" method that achieves a high degree of modularity (one or two interface functions per block) at the expense of reusability (some diagrams are rejected even though, once flattened, they are acyclic).

Although the need for multi-function interfaces has been also realized in [12, 3, 6], these works do not provide complete solutions. In particular, they do not address optimality and do not discuss modularity vs. reusability trade-offs. Also note that [3, 6] are focusing on the problem of *distribution* of synchronous programs. Distribution is related but not the same as modular code generation. For instance, one may look at the entire program (e.g., flatten it) in order to distribute it. More details about differences between our approach and other works can be found in the related work section of [9].

In this paper we focus on triggers and firing times. These mechanisms can be found in tools such as Simulink and SCADE, which have motivated our work.[3] *Clocks* is a construct similar to triggers, but more powerful, found in synchronous languages. The trigger elimination procedure is, to the best of our knowledge, new.

Related to our work on triggers are the works [10, 1, 3], which study the distribution of Signal programs. They use a dependency graph where dependencies are labeled with clocks. Therefore dependencies are "dynamic": they are valid only when their clock is true. Our framework with triggers is simpler: we do not use a clock calculus as in the works above. As a result, we are able to handle triggers in a "static" way, by adding one extra node to a (plain) dependency graph: this graph corresponds to the usage interface described above.

Regarding timed diagrams, a (period, phase) representation of Simulink sample times is used for purposes of type checking in [15]. A formal model for the distribution of Simulink diagrams is proposed in [16], however, triggers and firing times are not considered. To our knowledge, the representation of firing times as finite automata and the corresponding calculus are new.

Interfaces are of course a well established concept for modular software, and many "behavioral" variants have been recently proposed, including interface automata [4], timing interfaces [7], or causality interfaces [17]. These are all different from the firing time automata we propose here.

In this paper we consider diagrams that, if flattened, are acyclic. Cyclic diagrams can also be handled using the approach of [11, 13], which is to check at compile-time that, despite cyclic dependencies, the diagram still has well-defined semantics. This, however, requires knowledge of the function that the blocks compute, which is contrary to the idea of treating blocks as "black-boxes" for IP and modularity reasons. It is also possible to avoid such compile-time checks and rely on computing a fixpoint at run-time [5, 8], but this fixpoint may contain undefined values.

[3]SCADE and a discrete-time subset of Simulink such as the one considered in [15] can be mapped directly to our block diagram notation. The same holds for the subset of the synchronous language Lustre where every "when" operator is coupled with a "current" operator. Together these two operators correspond to a trigger.

3 Purely Synchronous Block Diagrams

This section contains a summary of our previous work [9].

The notation The purely synchronous block diagram notation is based on a set of *blocks* that can be connected to form *diagrams* (see Figure 2). Blocks are either *atomic* or *macro* (i.e. composite) blocks. Each block has a number of input *ports* (possibly zero) and a number of *output* ports (possibly zero). Diagrams are formed by connecting the output port of a block A to the input port of a block B (B can be the same as A). An output port can be connected to more than one input ports. However an input port can only be connected to a single output.

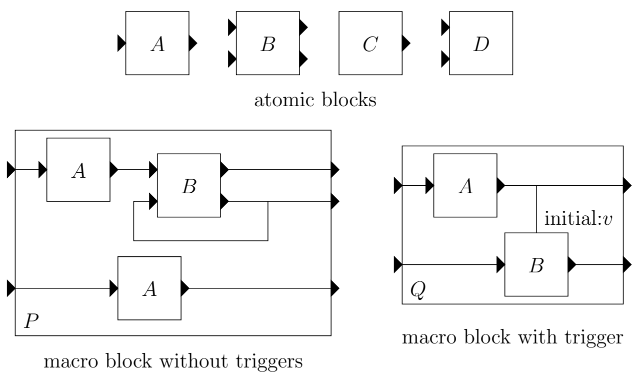


Figure 2. Examples of blocks and diagrams

A macro block encapsulates a block diagram into a block. The blocks forming the diagram are called the *internal* blocks of the macro block, or its *sub-blocks*. The connections between blocks ("wires") are called *signals*. Upon encapsulation: each input port of the macro block is connected to one or more inputs of its internal blocks, or to an output port of the macro block; and each output port of the macro block is connected to exactly one port, either an output port of an internal block, or an input of the macro block.

Combinational, sequential, Moore-sequential blocks Each atomic block A is pre-classified as either *combinational* (state-less) or *sequential* (having internal state). Some sequential blocks are *Moore-sequential*. Each output of a Moore-sequential block *only depends on the state,* *but not on the inputs*. For example a *unit-delay* block that stores the input and provides it as output in the next instant is a Moore-sequential block. On the other hand, an *integrator* block that outputs the sum of its input at all past instants is a non-Moore sequential block.

A macro block is combinational iff all its sub-blocks are combinational; otherwise it is sequential. A sequential macro block is Moore-sequential iff every path from an output port backwards towards the inputs eventually "meets" the output of a Moore-sequential sub-block. For example, in Figure 2, macro block P is Moore-sequential iff block A is Moore-sequential.

The concept of Moore-sequential blocks is used in the block-based dependency analysis and in the "step-get" modular code generation method. These are explained in the sequel.

Flattening A diagram is *flat* if it contains only atomic blocks. A *flattening* procedure can be used to transform a hierarchical block diagram into a flat one: (1) We start with the top-level diagram (which may consist of a single macro block). (2) We pick a macro block A and we replace it by its internal diagram. While doing so, we re-institute any connections that would be lost: If an input port p of A is connected externally to an output port q and internally to an input port r, then we connect q to r directly. Similarly for output ports of A. (3) If there are no more macro blocks left, we are done, otherwise, we repeat step (2).

Block-based dependency analysis and acyclic diagrams This type of dependency analysis is used only for the purpose of giving semantics to a diagram. We assume the diagram is flat. We construct a *block-based dependency graph*. The nodes of this graph are all the blocks in the diagram. For each block A that is not Moore-sequential, for each block B with some input connected to an output of A, we add a directed edge from A to B. We say that a diagram is *acyclic* if, once we flatten it and build its block-based dependency graph, we find that this graph has no cycles.

Semantics We only assign semantics to flat, acyclic diagrams. The semantics we use are standard synchronous semantics used also in languages like Lustre. Each signal x of the diagram is interpreted as a total function $x : N \to V_x$, where $N = \{0, 1, 2, 3, ...\}$ and V_x is a set of values: $x(k)$ denotes the value of signal x at time instant k. If x is an input this value is determined by the environment, otherwise it is determined by the (unique) block that produces x. Since the diagram is acyclic there exists a well-defined order of firing the blocks to compute the values of all signals in a given instant.

Modular code generation: inputs and outputs The code generation scheme proposed in [9] takes as inputs:

1. a macro block P and its internal block diagram; and
2. the *profile* of each sub-block of P (explained below).

It generates as outputs:

1. a profile for P.
2. the implementation (in a certain programming language such as C++, Java, etc.) of each of the interface functions listed in the profile of P.

Profiles The profile of a block is essentially its interface. Both atomic and macro blocks have profiles. The profile of a block A contains: (1) The *class* of A: whether A is combinational, Moore-sequential or non-Moore sequential. (2) A list of *interface functions* and their *signatures*. (3) A *profile dependency graph* (PDG) that describes the correct order in which these functions are to be called at every synchronous instant. The nodes of the PDG of A are the interface functions of A.

For example, Figure 3 shows the profiles of sub-blocks A, B, C of macro block P. Blocks A and C have a single interface function called "step()" which takes the input and returns the output of these blocks. Block B has two interface functions: B.step() and B.get(). B.get() takes no input and returns the output of B. B.step() takes the input of B and returns no output. This is a case where B is a Moore-sequential block: its get() method returns the outputs and its step() method updates the state, given the inputs. The PDG of B shown in the figure states that B.get() must be called before B.step(), at every synchronous instant.

Code generation steps Code generation is performed in three major steps:

(1) *Classification*: in this step the input macro block P is classified as combinational, Moore-sequential or non-Moore sequential, as explained above. It should be noted that the "dynamic" method does not use the class information [9]. Thus, for this method, the classification step is not necessary. On the other hand, class information is used in the "step-get" method, as explained below.

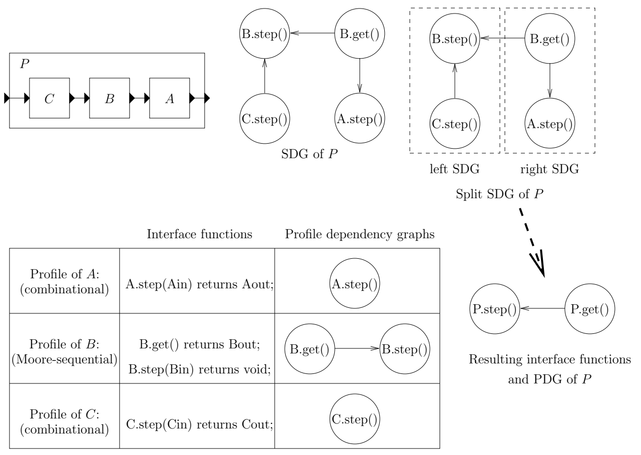
(2) *Dependency analysis*: this step determines whether there exists a valid execution order of the interface functions of the sub-blocks of P. It consists in building a *scheduling dependency graph* (SDG) for the given macro block P and then checking that this graph does not contain any cycles. The SDG for P is built by connecting the PDGs of all sub-blocks of P. If the SDG contains a cycle then P is *rejected*: this means that modular code generation fails and P needs to be flattened (one or more levels). Otherwise, we proceed to the code generation step.

(3) *Profile generation*: this step involves several sub-steps, including identifying the dependencies between inputs and outputs and then using these dependencies in order to *cluster* the SDG of P in a number of *sub-graphs*. For each sub-graph G, one interface function for P is generated by calling the functions included in G in the order specified by G. An init() function is also generated for sequential

blocks to initialize their state. Dependencies between sub-graphs are mapped into dependencies between the interface functions of the macro block, i.e., into a PDG for P.

Trade-offs Different clustering methods can be used, resulting in different code-generation methods. Two such methods are described in [9], the "dynamic" method and the "step-get" method. The dynamic method privileges reusability at the expense of modularity, while the step-get method privileges modularity at the expense of reusability. The dynamic method may result, in the worst case, in as many as $n + 1$ interface functions for a block, where n is the number of its outputs. However, this method is guaranteed to generate exactly two functions for Moore-sequential blocks: a "get" function that returns all outputs of such a block and a "get" function that updates the state of the block based on the inputs.

The step-get method, on the other hand, generates only one interface function for non-Moore-sequential blocks (and two functions for Moore-sequential blocks, like the dynamic method). The price to pay is that this adds extra dependencies between inputs and outputs, which may compromise the reusability of the block. We provide further details on this in Section 4 below (see discussion on Figure 6).



	Interface functions	Profile dependency graphs
Profile of A: (combinational)	A.step(Ain) returns Aout;	A.step()
Profile of B: (Moore-sequential)	B.get() returns Bout; B.step(Bin) returns void;	B.get() → B.step()
Profile of C: (combinational)	C.step(Cin) returns Cout;	C.step()

Figure 3. Example of modular code generation

Example Figure 3 shows a block diagram with macro block P and sub-blocks A, B, C, and the profiles of these sub-blocks. The SDG of P is built as shown in the figure: B.step() depends on C.step(), because the input of B is connected to the output of C. Likewise, A.step() depends on B.get(). Notice that the SDG of P is acyclic. Using the "step-get" method this SDG is clustered in two sub-graphs, called "left" and "right". Each of these sub-graphs gives rise to an interface function for P. The left SDG gives rise to P.step() and the right SDG to P.get(). The implementation of these functions is shown below:

```
P.get( ) {                    P.step( P_in ) {
  B_out := B.get();             C_out := C.step(P_in);
  P_out := A.step(B_out);       B.step( C_out );
  return P_out;               }
}
```

The PDG for P is obtained directly from the dependency of the left SDG upon the right SDG. As a result, P.step() depends on P.get(), as shown in the Figure.

4 Triggers

In this section we extend the synchronous block diagram notation with *triggers* and show how the modular code generation methods can be extended accordingly.

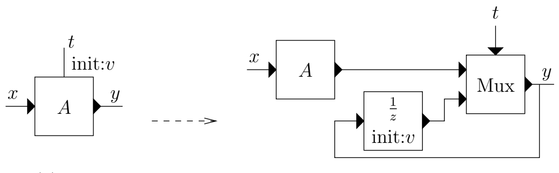
Any (atomic or macro) block A may be triggered by a Boolean signal x: x is then called the trigger of A. The intension is that A is to "fire" only when x is true. A block can have at most one trigger. In Figure 2, P has no triggers, whereas Q has one triggered sub-block B.

When a block is triggered, the user specifies initial values for each output of that block. These determine the values of the outputs during the initial period (possibly empty) until the block is triggered for the first time. In the example shown in Figure 2, an initial value v is specified for the (single) output of triggered block B.

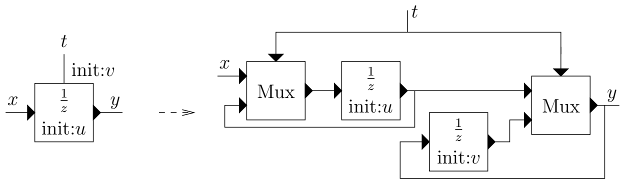
Classification of diagrams with triggers The classification step is similar to the case of purely synchronous diagrams, with some differences. First, macro blocks with triggered sub-blocks are sequential, since they require memory to store the initial value of the outputs of the triggered sub-blocks. Another difference is in how we decide that a block is Moore-sequential: when we follow a path from an output port backwards towards the inputs we make sure that we follow through both the inputs and the trigger (if any) of a non Moore-sequential block, and from the trigger (if any) of a Moore-sequential block.

For example, in Figure 2, Q is Moore-sequential iff *both* A and B are Moore-sequential. Block Q' of Figure 6 is not Moore-sequential because of the path from the output to input port t.

Trigger elimination Triggers do not add to the expressive power of block diagrams and can be eliminated by a structural transformation that preserves the semantics. Even though triggers can be eliminated, for purposes of code generation, it is not wise to do so: first, trigger elimination destroys modularity, and second, it results in more diagrams being rejected, as explained below. Therefore, the trigger elimination procedure we describe below is provided only to show that triggers do not add expressiveness to the model. We do not advocate the use of trigger elimination for purposes of code generation, since we can generate modular code that accounts for triggers, as explained later in this section.



(a) eliminating the trigger from a combinational atomic block



(b) eliminating the trigger from a unit-delay

Figure 4. Eliminating triggers from atomic blocks

The trigger elimination procedure essentially transforms triggers into inputs. The procedure is top-down. We start with the top-level macro block (which is not triggered but may contain triggered sub-blocks in its internal diagram). Call it P. For every triggered sub-block Q of P, we proceed as follows. If Q is a combinational atomic block, then we eliminate its trigger as shown in Figure 4(a). That is, we replace the triggered block A shown to the left by the diagram shown to the right of the figure: notice that there are no triggers in this diagram. If Q is the unit-delay atomic block, denoted $\frac{1}{z}$, then we eliminate its trigger as shown in Figure 4(b).

If Q is a macro block then we remove its trigger by adding it to its inputs, as illustrated in Figure 5. In particular, let t be the trigger of Q. We add an extra input port p for Q, remove the trigger and connect t to p. Then we trigger all internal blocks of Q by the new input p. Also, for each output y of Q, we insert an "Id block" ("identity-function block") right after the block producing y and set the initial value of this triggered Id block to the initial value of y (note that the Id block is also triggered by t). If some sub-block B of Q is already triggered by some other trigger t', then we add an atomic block computing the logical AND of t and t', say $t'' = t \wedge t'$, and then use t'' as the trigger for B. Triggers from internal blocks of Q can be eliminated recursively using the same procedure.

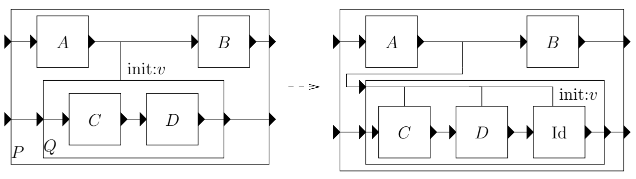


Figure 5. Eliminating the trigger from a macro block

Trigger elimination may turn a Moore-sequential macro block into a non-Moore-sequential block. This is illustrated in Figure 6: Q is Moore whereas Q' is not. As a result, trigger elimination may result in some diagrams being rejected, in particular when using the step-get method, although they could be accepted if we did not use the elimination procedure. For example, connecting output y to input x in macro block Q' of Figure 6 results in this block being rejected by the step-get method. This is because Q' is not a Moore-sequential block, therefore the step-get method generates only one interface function for that block: this introduces the (false) dependency from x to y. For these reasons, we present in the sequel a modular code generation method that directly handles triggered diagrams without the need for trigger elimination.

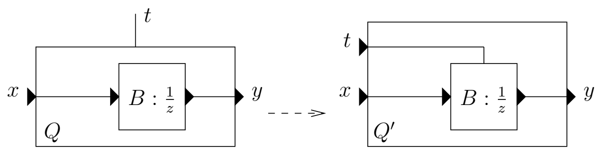


Figure 6. Trigger elimination may turn a Moore-sequential block into a non-Moore sequential block

Semantics of triggered diagrams We assign semantics to flat diagrams where only atomic blocks can be triggered. Before flattening a diagram we can eliminate triggers from all macro blocks, therefore, the flattening procedure for diagrams without triggers described in Section 3 applies.

The semantics are similar to those of diagrams without triggers. The only difference is that if signal x is an output of a block A triggered by some signal t, then for each instant k, in order to determine $x(k)$ we first need to determine $t(k)$: if $t(k)$ is false, then $x(k) = x(k-1)$ (if $k = 0$ then $x(k) = v$ where v is the initial value specified by the user); if $t(k)$ is true then $x(k)$ is defined by "firing" the block that produces x, just like in the case where A is not triggered.

Modular code generation with triggers The code generation scheme described in Section 3 can be extended to diagrams with triggers. The inputs and outputs of the scheme remain the same. Profiles of blocks also do not change. The Classification step is modified as explained above.

The Dependency Analysis step, and in particular, how to build the SDG of a macro block P, is modified as follows. For every sub-block A of P, if A is triggered, then we add to the profile dependency graph of A an extra node, called *A.trigger*. We add a dependency edge from A.trigger to every other node in the PDG of A: this means that every interface function of A depends on the trigger. Then, we use this new PDG of A in order to build the SDG of P.

An example is shown in Figure 7. Block C is triggered by the output of A. Therefore, a node C.trigger is added in

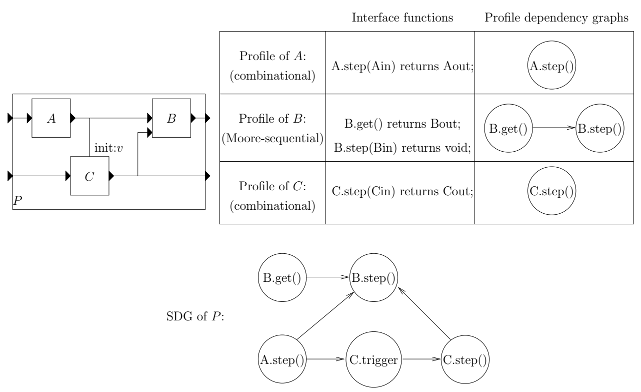

Interface functions
Profile dependency graphs
Profile of A: (combinational)
A.step(Ain) returns Aout;
A.step()
Profile of B: (Moore-sequential)
B.get() returns Bout;
B.step(Bin) returns void;
B.get()
B.step()
Profile of C: (combinational)
C.step(Cin) returns Cout;
C.step()
A
B
C
init:v
P
SDG of P:
B.get()
B.step()
A.step()
C.trigger
C.step()

Figure 7. Building a scheduling dependency graph with a triggered block C

the SDG of P: this node depends on A.step() which produces A's output. In turn, C.step() depends on C.trigger.

Again, once the modified SDG for P is built as described above, it is checked for dependency cycles. If it has a cycle, the diagram is rejected. Otherwise, we proceed with the clustering step. The same various clustering methods can be used for triggered diagrams as for purely synchronous diagrams. Each resulting sub-graph will give rise to a separate interface function for P, as in the purely synchronous case. For example, for the macro block P shown in Figure 7, we could use a clustering method that results in a single cluster that contains all nodes of the SDG of P. This means that P will have a single interface function, called P.step().

The implementation of the interface functions for P is modified in the case of triggered diagrams as follows. If A is a sub-block of P triggered by some signal t then the calls to all interface functions of A are placed within a conditional if-statement guarded by t. For example, consider again the macro block P shown in Figure 7 and suppose we generate a single function P.step() as described above. Then, a possible implementation of P.step() is:

```
P.step( P_in1, P_in2 ) returns ( P_out1, P_out2 ) {
  A_out := A.step( P_in1 );
  P_out1 := B.get();
  if (A_out) then
    P_out2 := C.step( P_in2 );
  end if;
  B.step( A_out, P_out2 );
  return( P_out1, P_out2 );
}
```

Outputs of triggered sub-blocks are *persistent* variables, meaning they maintain their value across successive calls to the interface functions.[4] In the example of Figure 7, P_out2 needs to be persistent, so that it maintains its value when block C is not triggered. P_out2 must be initialized by P.init(). Assuming A and B to be sequential and C combinational, a possible implementation of P.init() is:

```
P.init() {
  A.init();
  B.init();
  P_out2 := initial_value_v;
}
```

5 Timed Block Diagrams

Timed block diagrams are a sub-class of diagrams with triggers, where some triggers are *statically* defined, that is, the instants where the trigger is true are known at compile-time: such a trigger is called a *firing time specification* (FTS). The notation for timed diagrams is as described in the previous sections, with the addition that every *non-triggered* atomic block A has an associated FTS.[5] A block in a timed diagram is non-triggered if it is not triggered by a standard, "dynamic" trigger (i.e., not an FTS), and its parent is also non-triggered. Since FTS are themselves triggers, we do not allow a block to have both a trigger and a FTS. Thus only non-triggered blocks can have an FTS.

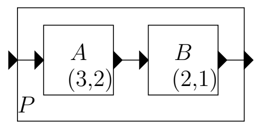


Figure 8. A timed block diagram

An example is shown in Figure 8. In this case, FTSs are represented as (period, initial phase) pairs: we abbreviate a (period, phase) pair as PPP.[6] The periods of blocks A and B are 3 and 2, and their initial phases are 2 and 1, respectively. This means that A is triggered at instants 2, 5, 8, ..., whereas B is triggered at instants 1, 3, 5, 7, ..., as shown in Figure 9.

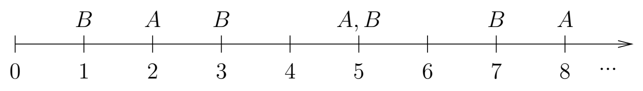


Figure 9. Times where blocks A and B of Figure 8 need to be executed

[4]One can think of the code for a block as a class in an object-oriented language. The interface functions correspond to public methods of this class, and the persistent variables to private variables of the class.

[5]In this paper, we will not be concerned with how the FTS is derived: it may be specified by the user, or it can be computed automatically by some *clock-inference* procedure as the one described in [15].

[6]In Simulink, PPPs are called "sample times". Simulink allows for both periods and initial phases to be specified. Initial phases are useful in designs where some blocks should start executing only after some set-up delay. The periods need not be harmonic (i.e., multiples of each other). Indeed, real applications often exhibit non-harmonic rates (e.g., see [14]).

Since timed diagrams are special cases of triggered diagrams, we could simply use the code generation scheme described in the previous sections. However, it is beneficial to take advantage of the extra information timed diagrams provide, namely, the statically defined FTSs, in order to generate more efficient code. To that end, in the case of timed diagrams, the profile of a block is extended to include an FTS. Every macro block has an FTS, computed from the FTSs of its sub-blocks. The FTS can be used to fire the macro block only when necessary, thus avoiding run-time overhead. This can be significant for macro blocks with sizable sub-block hierarchies. Moreover, the information included in FTS can be used to relax the dependencies between blocks, thus resulting into accepting more diagrams. In the sequel we show how FTSs can be represented and manipulated algorithmically, and how they are used for efficient code generation.

A PPP is not the only possible FTS representation. Indeed, we propose below a more "composable" representation in terms of finite automata. To see the motivation behind this, consider again the example shown in Figures 8 and 9. Observe that at times 0, 4, 6, and so on, neither A nor B is executed. This means that macro block P does not need to be executed at these times either.

Now, suppose we wanted to represent the times when P needs to be executed as a PPP. Then, the period of this PPP cannot be greater than 1, otherwise P will "miss" some executions of either A or B. Also, the initial phase of this PPP cannot be greater than 1, otherwise the initial execution of B will be missed. Such a PPP representation is clearly wasteful. It results in P being triggered also at times 4, 6, and so on, whereas it need not be. It should now be clear why (period, phase) representations are not optimal. The finite-automata representation we propose below remedies this.

Before we proceed, we must also note the following. If the initial phase specified by the FTS is greater than zero, the FTS must also specify an initial value for each output of the block. This is used, just as in the case of triggered blocks, during the initial interval before the block fires for the first time. For example, in Figure 8, initial values for the outputs of both blocks A and B need to be defined.

5.1 Firing Time Specifications

Semantically, an FTS is a set $T \subseteq N$. For an automated method, we need a finitary representation of such a set. We propose here two different representations for an FTS: as a PPP or as a finite automaton called a *firing time automaton* (FTA). A PPP (π, θ) represents the set $\{\pi \cdot n + \theta \mid n \in N\}$. Two FTA corresponding to PPPs $(3, 2)$ and $(2, 1)$ are shown in Figures 10(a) and (b), respectively. States drawn with double circles are the accepting states of the automata, corresponding to the instants where a block should fire. Transitions correspond to one time unit elapsing.

PPPs are more compact whereas FTA are strictly more expressive: every PPP can be translated into an equivalent FTA (e.g., see above figures) whereas the opposite translation is not always possible. For instance, the set corresponding to the union of PPPs $(3, 2)$ and $(2, 1)$ can be represented as the FTA shown in Figure 10(c), but not as a PPP.

Ideally, a macro block P should fire *if and only if* some of its sub-blocks needs to fire. This means that the FTS T of P should be equal to the union of the FTSs $T_1, T_2, ...$ of its sub-blocks. This can be achieved with FTA because they are closed under union, as is shown below. But it cannot be always achieved with PPPs, as explained above. Consequently, with PPPs, T is generally computed as an *over-approximation* of $\bigcup_i T_i$, i.e., $T \supseteq \bigcup_i T_i$. This is done using a generalization of the greatest common divisor (GCD) operator, described below. The over-approximation is correct, but may be inefficient, since it may result in P being fired even though none of its sub-blocks need to fire.

Once the FTS of P is defined, the FTSs of its sub-blocks need to be modified. This is because the original FTSs were defined under the assumption that P fires *at all times* (i.e., $T = N$). Now that a new T has been computed this may no longer be true and T_i need to be modified accordingly. To see this, consider an example (for simplicity we assume all FTSs have phase zero). Suppose P has two sub-blocks A_1 and A_2 with FTSs (i.e., periods) 2 and 4, respectively. Then, the FTS of P should be 2. This means P will now fire only at times $0, 2, 4, ...$ Consequently, the periods of A_1 and A_2 need to be updated to 1 and 2, respectively. Then, A_1 will be fired every $2 \cdot 1$ instants and A_2 every $2 \cdot 2$ instants, as specified originally.

We call updating the FTSs of sub-blocks *factoring*. In the simple case of periods factoring is just dividing the period of the sub-block by the period of its parent, e.g., $\frac{2}{2}$ and $\frac{4}{2}$, in the example above. The composition of the FTS of the parent with the FTS of a sub-block is in turn multiplication. These operations can be extended to PPPs and FTA as shown below. We denote them $\oslash$ (for division) and $\odot$ (for multiplication).

5.1.1 The (period, phase) representation

PPPs are not closed under union. Thus, instead of union, we use the *generalized greatest common divisor* (GGCD) operator defined in [15]:

$$GGCD\{(\pi_1, \theta_1), (\pi_2, \theta_2)\} = \begin{cases} (\gcd(\pi_1, \pi_2), \theta_1), & \text{if } \theta_1 = \theta_2, \\ (\gcd(\pi_1, \pi_2, \theta_1, \theta_2), 0), & \text{otherwise} \end{cases}$$

where gcd denotes the usual GCD operator. Note that $GGCD\{(\pi_1, \theta_1), (\pi_2, \theta_2)\} \supseteq (\pi_1, \theta_1) \cup (\pi_2, \theta_2)$, thus, we have a safe approximation for the FTS of a macro block.

5.1.2 The firing time automata representation

An FTA is a deterministic finite-state automaton A over the single-letter alphabet $\{1\}$. Such an automaton A defines a language $L(A) \subseteq \{1\}^*$. $L(A)$ encodes a subset $T(A) \subseteq N$, as follows:

$$T(A) = \{k \in N \mid 1^k \in L(A)\}.$$

Two automata A and B are equivalent, denoted $A \equiv B$, iff $L(A) = L(B)$. Examples are shown in Figure 10, where transitions are implicitly labeled with 1.

An FTA A is formally a tuple $(S_A, s_0^A, F_A, \delta_A)$, where S_A is the set of states, $s_0^A \in S_A$ is the initial state, $F_A \subseteq S_A$ is the set of accepting states and $\delta_A : S_A \to S_A$ is the transition function. Given FTA $A = (S_A, s_0^A, F_A, \delta_A)$ and $B = (S_B, s_0^B, F_B, \delta_B)$, the union of A and B, denoted $A \cup B$, is the standard construction on automata such that $L(A \cup B) = L(A) \cup L(B)$.

Every FTA A is equivalent to a finite union of FTA, $A = A' \cup A_1 \cup \cdots \cup A_n$, where $L(A')$ is finite and each A_i can be represented as a PPP. To see this, observe that, since an FTA is deterministic over a single-letter alphabet, A has the structure of a *lasso*, i.e., an initial finite sequence of transitions σ followed by a cycle ρ. We define A' to be the FTA consisting of only the initial segment σ of A. Clearly, $L(A')$ is finite (if σ is empty or has no accepting states, then $L(A')$ is empty). Let ρ have n accepting states $s_1, ..., s_n$. Let A_i be identical to A except that it has only one accepting state, namely s_i. Clearly, $A = A' \cup A_1 \cup \cdots \cup A_n$. Now, A_i can be expressed as (π_i, θ_i) where π_i is the length of the cycle ρ and θ_i is the number of transitions needed to reach s_i from the initial state.

The division and multiplication operators on FTA are defined as follows:

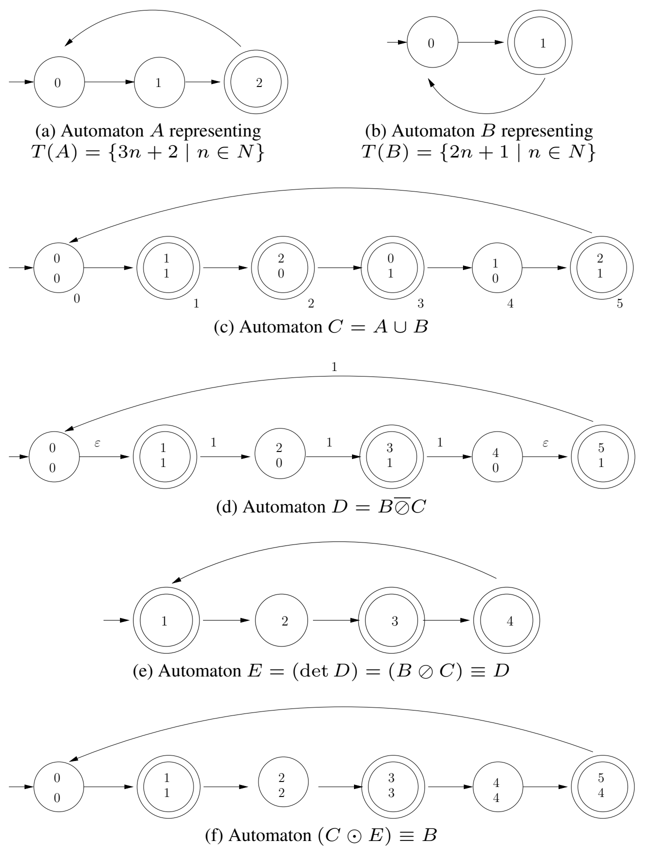
$$A \odot B \;=\; \big(S_A \times S_B, (s_0^A, s_0^B), F_A \times F_B, \delta_{A \odot B}\big)$$

with $\delta_{A \odot B}(s_A, s_B) = (s'_A, s'_B)$, where $s'_A = \delta_A(s_A)$ and, if $s_A \in F_A$ then $s'_B = \delta_B(s_B)$, otherwise $s'_B = s_B$.

$$\begin{aligned}
B \oslash A &= \det(B \overline{\oslash} A)\\
B \overline{\oslash} A &= \big(S_A \times S_B, (s_0^A, s_0^B), S_A \times F_B, \Delta_{B \,\overline{\oslash}\, A}\big)\\
\Delta_{B \,\overline{\oslash}\, A} &= \big\{ (s_A, s_B) \xrightarrow{1} (\delta_A(s_A), \delta_B(s_B)) \,|\, s_A \in F_A \big\}\\
&\cup \;\big\{ (s_A, s_B) \xrightarrow{\varepsilon} (\delta_A(s_A), \delta_B(s_B)) \,|\, s_A \notin F_A \big\}
\end{aligned}$$

The $\overline{\oslash}$ operator produces an automaton with ε-transitions and $\det(A)$ represents the equivalent deterministic automaton which can be produced by the usual determinization procedure that removes such transitions. The $\overline{\oslash}$ operator is illustrated in Figure 10(d). Automaton D is determinized to obtain the equivalent automaton E shown in Figure 10(e). The $\odot$ operator is illustrated in Figure 10(f).

The $\odot$ and $\oslash$ operators are not commutative. Here are some other interesting properties of these operators.


(a) Automaton A representing $T(A) = \{3n + 2 \mid n \in N\}$

(b) Automaton B representing $T(B) = \{2n + 1 \mid n \in N\}$

(c) Automaton $C = A \cup B$

(d) Automaton $D = B \overline{\oslash} C$

(e) Automaton $E = (\det D) = (B \oslash C) \equiv D$

(f) Automaton $(C \odot E) \equiv B$

Figure 10. Automata representing firing times.

Theorem 5.1. *For all FTA A, B:*

1. *$(A \cup B)$, $(A \odot B)$ and $(A \oslash B)$ are also FTA.*
2. *$\emptyset \odot A = A \odot \emptyset = \emptyset$ and $\{1\}^* \odot A = A \odot \{1\}^* = A$.*
3. *$\emptyset \oslash A = \emptyset$ and $A \oslash \{1\}^* = A$.*
4. *If $L(A) \supseteq L(B)$ then $A \odot (B \oslash A) \equiv B$.*

In the above theorem, $\emptyset$ and $\{1\}^*$ denote the FTA with languages empty and equal to $\{1\}^*$, respectively.

5.2 Activity and dependency analysis

The dependency analysis step for timed diagrams is the same as the one described in the previous sections, with the difference that it is preceded by an *activity analysis* step. The goal of activity analysis is to discover *false* dependencies, that is, dependencies among blocks that are never active (i.e., fire) at the same time. As an example, consider the timed diagram shown in Figure 11 and assume both blocks A and B are combinational. It may seem that this diagram contains a dependency cycle, however, this is not true because the two blocks never fire at the same time: A fires at times 0, 2, 4, ..., whereas B fires at times 1, 3, 5, ...

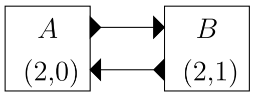


Figure 11. A timed diagram with false dependencies

A simple activity analysis method is to check, for every two sub-blocks A and B of the macro block P for which the SDG is to be built, whether $T_A \cap T_B$ is empty, where T_A and T_B are the FTSs of A and B, respectively. Then, when building the SDG of P, a dependency between two interface functions of A and B is added only if $T_A \cap T_B$ is non-empty.

Checking the condition $T_A \cap T_B = \emptyset$ can be easily done when T_A and T_B are represented as FTA: it amounts to checking whether the intersection of the languages accepted by two finite automata is empty. Standard algorithms can be used for this. If T_A and T_B are represented as PPPs (π_A, θ_A) and (π_B, θ_B), respectively, then checking $T_A \cap T_B = \emptyset$ amounts to finding a solution to the following linear Diophantine equation in two variables x, y:

$$\theta_A + \pi_A \cdot x = \theta_B + \pi_B \cdot y.$$

In this case this is equivalent to checking whether the difference $\theta = \theta_A - \theta_B$ is an integer multiple of the GCD π of π_A and π_B: if θ is a multiple of π then a solution exists, otherwise no solution exists (cf. *Bézout's lemma*).

We should note that the simple activity analysis method proposed above is *conservative*, in the sense that it cannot detect *all* false dependencies that may exist in a diagram. For example, consider a diagram that consists of three combinational blocks in a feedback loop: $A \to B \to C \to A$. We can easily construct a scenario where: (1) for any pair of blocks in this set, there exists a time where both blocks are active, but (2) there is no time when all three blocks are active. (1) means this is a false dependency cycle. (2) means that the analysis method presented above will not be able to detect this and will reject this diagram.

This can be improved by performing a separate activity analysis for each dependency cycle that is found in the SDG of P. This activity analysis would involve checking whether the intersection of the FTSs of *all* blocks involved in the cycle. This more accurate (but also more costly) method would discover the false dependency in the above example. Still, even this method cannot discover all false dependencies. In general, we need to associate an FTS not to the entire block or macro block, but to each individual interface function of that block. The details of how this can be done are beyond the scope of this paper and will be presented in future work.

5.3 Profile generation

The profile generation step for timed diagrams involves all steps involved in purely synchronous diagrams, plus generating the FTS for the macro block and factoring the FTSs of its sub-blocks.

FTS generation and factoring Let $T_1, ..., T_n$ be the firing time specifications of all sub-blocks of the macro block P. Then the firing time specification T of P is defined as

$$T = T_1 \oplus \cdots \oplus T_n,$$

where $\oplus$ is either the *GGCD* operator, in case T_i are represented as PPPs, or the union operator $\cup$, in case they are represented as FTA.

Once the FTS for P is computed, the FTSs of all sub-blocks of P are updated as follows:

$$T_i' = T_i \oslash T.$$

The updated T_i' are called the *factored* FTSs.

Factoring as above can be done in case T_i are represented as FTA, but also when they are represented as PPPs, by first translating them into FTA. In the latter case, it can be shown that the resulting factored FTS can always be represented as a PPP. In particular, let $T_i = (\pi_i, \theta_i)$ and let $T = (\pi, \theta)$. By definition, $(\pi, \theta) = GGCD\{(\pi_i, \theta_i)\}_{i=1,...,n}$. Then the following holds:

$$(\pi_i, \theta_i) \oslash (\pi, \theta) = \begin{cases} (\frac{\pi_i}{\pi}, \theta), & \text{if } \theta_i = \theta, \\ (\frac{\pi_i}{\pi}, \frac{\theta_i}{\pi}), & \text{otherwise} \end{cases}$$

The correctness of the factoring procedure is obtained as a corollary of Part 4 of Theorem 5.1. Observe that, by definition of T, $L(T) \supseteq \bigcup_i L(T_i)$, thus also $L(T) \supseteq L(T_i)$ for all i. Then, Part 4 of Theorem 5.1 applies and we get:

$$T \odot T_i' = T \odot (T_i \oslash T) \equiv T_i$$

This indeed means that each factored FTS T_i', when composed with the FTS T of the macro block, is equivalent to the original FTS T_i. Thus each sub-block will be fired exactly at the instants specified by its original FTS.

FTS periods An FTS represented as a PPP (π, θ) or an FTA A has a *period*: in the former case the period is simply π; in the latter case the period is the length of the unique cycle in the automaton A. For example, the period of the automaton shown in Figure 10(c) is 6.

Associating modulo counters to factored FTSs Every FTS$_i'$ with period greater than 1 is implemented by a persistent integer variable c_i functioning as a modulo counter. Counter c_i is modulo the period of FTS$_i'$. A counter c_i can be in one or more *accepting states*. A counter implementing a PPP has a single accepting state corresponding to the initial phase value. A counter implementing an FTA has as many accepting states as the accepting states of the automaton. For example, if FTS$'_i$ is $(3, 2)$ then it is implemented by a modulo-3 counter. The accepting state of the counter is 2. If FTS$'_i$ is the automaton shown in Figure 10(e), then it is implemented by a modulo-4 counter. The accepting states of this counter are 0, 2, and 3.

Generating code The classification, input-output dependency analysis, and SDG clustering steps remain as described in the previous sections. The only difference is with the interface function implementation step. When implementing these functions for timed diagrams, the activity of each sub-block, determined by its factored FTS, needs to be taken into account before calling the interface functions of the sub-block. In particular:

The call to any interface function A.f() of any sub-block A is guarded by a conditional if-statement that checks whether the modulo counter implementing the factored FTS of A is in one of its accepting states.

At every instant, the modulo counter for A is incremented after the *last* interface function of A has been called. How to determine which one is the last function of A depends on the code generation method used. It can sometimes be done statically, as in the case of the "step-get" method, the last function is always step(). In other times it needs to be done dynamically, as in the case of the "dynamic" method. Here, another counter can be used: this counter should be initialized at every instant to N_A, the total number of interface functions for A, and decremented every time a function of A is called. When the counter reaches 0 the last function has been called.

The modulo counters are initialized to zero by the init() method of the block. For blocks that have a phase greater than one, init() also initializes their outputs to initial values specified by the user, as in the case of triggered blocks.

Example Consider the timed diagram shown in Figure 12. B is assumed to be a Moore-sequential block, the other blocks are assumed to be combinational. Only periods are shown in the figure: it is assumed that all phases are zero. We use PPPs to represent FTSs: in this case they are just periods, thus we use standard GCD, division and multiplication operations.

The activity analysis step for this example concludes that blocks A and B may fire at the same time, and so can blocks C and D. Therefore, the dependency analysis step builds the same SDGs for P and Q as it would in the purely synchronous case. Both SDGs are acyclic, thus we can proceed with the profile generation step:

- We compute the FTS of P as the GCD of 2 and 4: this is equal to 2. We compute the FTS of Q as the GCD of 3 and 5: this is equal to 1.
- We compute the factored FTSs of A, B, C, D: these are equal to $\frac{2}{2} = 1$, $\frac{4}{2} = 2$, $\frac{5}{1} = 5$, $\frac{3}{1} = 3$, respectively.
- We assign a modulo-2 counter c_B to B. A does not need a modulo counter since its factored FTS is 1. We assign a modulo-5 counter c_C to C. We assign a modulo-3 counter c_D to D.

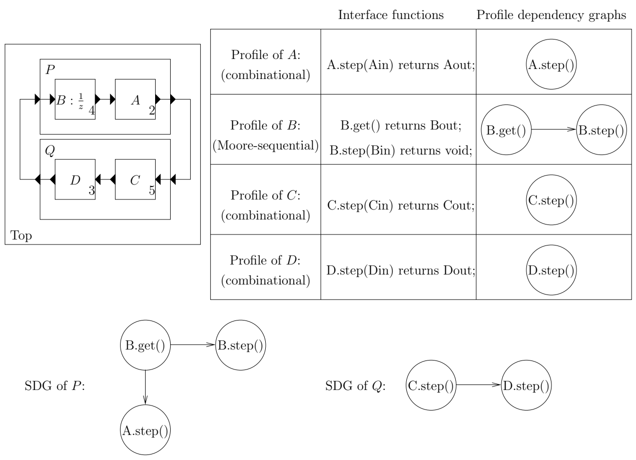

P
$B : \frac{1}{z}$
4
A
2
Q
D
3
C
5
Top
Interface functions
Profile dependency graphs
Profile of A: (combinational)
A.step(Ain) returns Aout;
A.step()
Profile of B: (Moore-sequential)
B.get() returns Bout;
B.step(Bin) returns void;
B.get()
B.step()
Profile of C: (combinational)
C.step(Cin) returns Cout;
C.step()
Profile of D: (combinational)
D.step(Din) returns Dout;
D.step()
SDG of P:
B.get()
B.step()
A.step()
SDG of Q:
C.step()
D.step()

Figure 12. A hierarchical timed diagram (top-left), profiles of its atomic blocks (top-right), and SDGs for macro blocks P and Q (bottom)

- We implement the interface functions of P and Q. Using the "step-get" method, we get:

```
Q.init() {          P.init() {
  c_C := 0;           c_B := 0;
  c_D := 0;         }
}

Q.step( Q_in ) returns Q_out {
  if (c_C = 0) then
    C_out := C.step( Q_in );
  end if;
  c_C := (c_C + 1) mod 5;
  if (c_D = 0) then
    Q_out := D.step( C_out );
  end if;
  c_D := (c_D + 1) mod 3;
  return Q_out;
}

P.step( P_in ) {
  if (c_B = 0) then
    B.step( P_in );
  end if;
  // c_B updated because B.step() is called last
  c_B := (c_B + 1) mod 2;
}

P.get( ) {
  if (c_B = 0) then
    B_out := B.get();
  end if;
  // c_B not updated because B.step()
  // is still to be called
  P_out := A.step( B_out );
  return P_out;
}
```

- We compute the FTS of Top as the GCD of 2 and 1: this is equal to 1.
- We compute the factored FTS of P: this is equal to $\frac{2}{1} = 2$. We compute the factored FTS of Q: this is equal to $\frac{1}{1} = 1$.
- We assign a modulo-2 counter to P. We assign no modulo counter to Q, since its factored FTS is 1.
- We implement the interface functions of Top. Using the "step-get" method, we get:

```
Top.step() {                            Top.init() {
  if (c_P = 0) then                       P.init();
    P_out := P.get();                     c_P := 0;
  end if;                                 Q.init();
  Q_out := Q.step( P_out );             }
  if (c_P = 0) then
    P.step( Q_out );
  end if;
  c_P := (c_P + 1) mod 2;
}
```

6 Conclusions

We have extended our previous work on modular code generation for synchronous block diagrams to triggered and timed diagrams. Although triggers can be eliminated, as we also showed, this is not desirable since it destroys modularity. To avoid this we have proposed a modular code generation method that directly accounts for triggers.

We have also proposed methods specialized to timed diagrams. Although timed diagrams are special cases of triggered diagrams, treating them directly allows us to obtain efficient code, that avoids firing the block unnecessarily. We achieve this by enriching the interface of macro blocks with firing time information. Existing firing time representations are generally conservative, resulting in non-optimal code. To remedy this, we have devised a novel and accurate (exact) representation of firing times. This novel representation uses finite automata and is amenable to algebraic manipulation and generation of efficient code.

Future work includes developing efficient methods for more accurate activity analysis. We would also like to evaluate the set of methods presented here and in [9] through experiments, in order to understand how the modularity/reusability trade-off, but also other trade-offs related to code quality such as size or execution time, manifest themselves in practice. Finally, we would like to extend our modular code generation approach to other high-level modeling formalisms.

References

[1] P. Aubry, P. Le Guernic, and S. Machard. Synchronous distribution of Signal programs. In *Proc. 29th Hawaii Intl. Conf. Sys. Sciences*, pages 656–665. IEEE, 1996.

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

[3] A. Benveniste, P. Le Guernic, and P. Aubry. Compositionality in dataflow synchronous languages: specification & code generation. Technical Report 3310, Irisa - Inria, 1997.

[4] L. de Alfaro and T. Henzinger. Interface theories for component-based design. In *EMSOFT'01: Proceedings of the First International Workshop on Embedded Software*, pages 148–165. Springer, 2001.

[5] 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.

[6] O. Hainque, L. Pautet, Y. L. Biannic, and E. Nassor. Cronos: A Separate Compilation Toolset for Modular Esterel Applications. In *World Congress on Formal Methods (FM'99)*, pages 1836–1853. Springer, 1999.

[7] T. Henzinger, C. Kirsch, and S. Matic. Composable code generation for distributed Giotto. In *Languages, Compilers and Tools for Embedded Systems (LCTES'05)*, pages 21–30. ACM, 2005.

[8] E. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT '07: Proc. 7th ACM & IEEE Intl. Conf. on Embedded software*, pages 114–123. ACM, 2007.

[9] R. Lublinerman and S. Tripakis. Modularity vs. Reusability: Code Generation from Synchronous Block Diagrams. In *Design, Automation, and Test in Europe (DATE'08)*. ACM, Mar. 2008.

[10] O. Maffeis and P. Le Guernic. Distributed Implementation of Signal: Scheduling & Graph Clustering. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 547–566. Springer, 1994.

[11] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, 1994.

[12] P. Raymond. Compilation séparée de programmes Lustre. Master's thesis, IMAG, 1988. In French.

[13] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference (EDTC'96)*. IEEE Computer Society, 1996.

[14] S. Tripakis. Description and Schedulability Analysis of the Software Architecture of an Automated Vehicle Control System. In *2nd Intl. Conf. on Embedded Software (EMSOFT'02)*, volume 2491 of *LNCS*, pages 123–137. Springer, Oct. 2002.

[15] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating Discrete-Time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems*, 4(4):779–818, 2006.

[16] M. Zennaro and R. Sengupta. Distributing synchronous programs using bounded queues. In *EMSOFT '05: 5th ACM Intl. Conf. on Embedded Software*, pages 325–334. ACM, 2005.

[17] Y. Zhou and E. Lee. Causality interfaces for actor networks. Technical Report UCB/EECS-2006-148, EECS Department, University of California, Berkeley, Nov 2006.