Modular Code Generation from Synchronous Block Diagrams*

Roberto Lublinerman** and Stavros Tripakis***

Abstract. We study modular, automatic code generation from hierarchical block diagrams with synchronous semantics. Such diagrams are the fundamental model behind widespread tools such as Simulink and SCADE. Modularity means code is generated for a given composite block independently from context, that is, without knowing in which diagrams this block is going to be used. This can be achieved by abstracting a given macro (i.e., composite) block into a set of interface functions plus a set of dependencies between these functions. These two pieces of information form the exported interface for a block.

This approach allows modularity to be quantified, in terms of the size of the interface, that is, the number of interface functions that are generated per block. The larger this number, the less modular the code is. This definition reveals a fundamental trade-off between modularity and reusability (set of diagrams the block can be used in): using an abstraction that is too coarse (i.e., too few interface functions) may create false input-output dependencies and result in dependency cycles due to feedback loops. In this paper we explore this and other trade-offs. We also show how the method can be extended from a purely synchronous block diagram model to triggered and timed block diagrams, which allow for modeling multi-rate systems.

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.

* This paper gives a brief summary of the work described in [1–3].

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

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

[1] www.mathworks.com/products/simulink/

[2] www.esterel-technologies.com/products/scade-suite/

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.[3]

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.

2 The problem

Turning the above method to a modular method that avoids flattening is not straightforward. In particular, the approach of generating a *monolithic* piece of code for each block does not work. To illustrate the problem, consider the example shown in Figure 1 (similar examples can be found in [8]). 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 is going to be connected. Suppose we generate a single "step" function for P, of the following form:

```
P.step( x1, x2 ) returns ( y1, y2 )
{
  y1 := A.step( x1 );
  y2 := B.step( x2 );
  return ( y1, y2 );
}
```

[3] In this paper we consider diagrams that, if flattened, are acyclic. Cyclic diagrams can also be handled using the approach of [4, 5], 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 [6, 7], but this fixpoint may contain undefined values.

where x1 and x2 denote the inputs of P, y1 and y2 denote its outputs, and A.step() and B.step() are the step functions for blocks A and B (these functions could have been generated previously in an automatic manner, or could have been written "manually", for instance, if A and B are atomic blocks).

The problem with the above monolithic code generation approach is that we have created additional, *false* dependencies between the inputs and outputs of P: in order to call P.step() we need to have a value for both x1 and x2. This implies that both outputs y1 and y2 now depend on both inputs. However, in the original diagram, y2 does not depend on x1 and y1 does not depend on x2. We will discover problems with our code generation scheme when we attempt to connect the block P as shown to the middle or right of Figure 1: in both these cases we will find a dependency cycle between the inputs and outputs of P.step(). However, there is no dependency cycle between the inputs and outputs of P. Thus, whereas flattening succeeds in generating code for this example, the above monolithic code generation approach fails.[4]

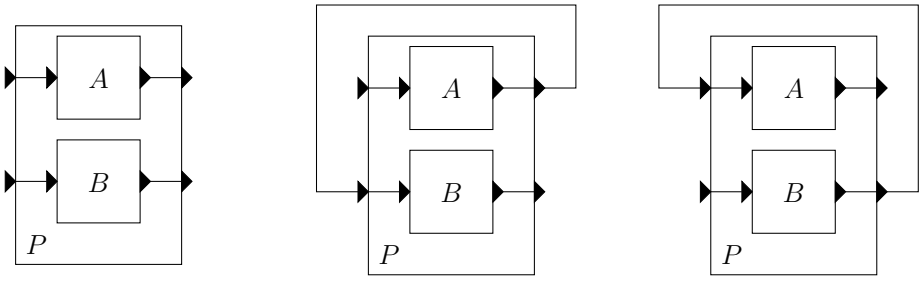


Fig. 1. A hierarchical block diagram (left) and two possible ways to connect the macro block P (middle and right).

3 The solution

In [1] 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. In the above example, we would generate two interface functions for P, as shown below:

[4] As we found out in some recent experiments [3], this type of examples are not just academic examples. Indeed, they arise often in realistic case studies, such as models from the Simulink demo suite.

```
P.get1(x1) returns y1
{
  y1 := A.step(x1);
  return y1;
}
```

```
P.get2(x2) returns y2
{
  y2 := B.step(x2);
  return y2;
}
```

In general, we need to export along with the set of interface functions a *set of dependencies between these functions*: these specify the correct usage of the interface, that is, the order in which the functions should be called within a synchronous instant. This set of dependencies is represented by a *profile dependency graph* (PDG), a directed acyclic graph, the nodes of which are interface functions and the edges of which are dependencies between these functions. Together the set of interface functions (with their implementation in a given programming language such as C, Java, ...) form the *profile* of a block. In the case of the above example, the two functions P.get1() and P.get2() are independent, thus the PDG for P is trivial: it consists of two independent nodes and no edges. In other cases, however, a PDG is needed. A simple example is shown in Figure 2.

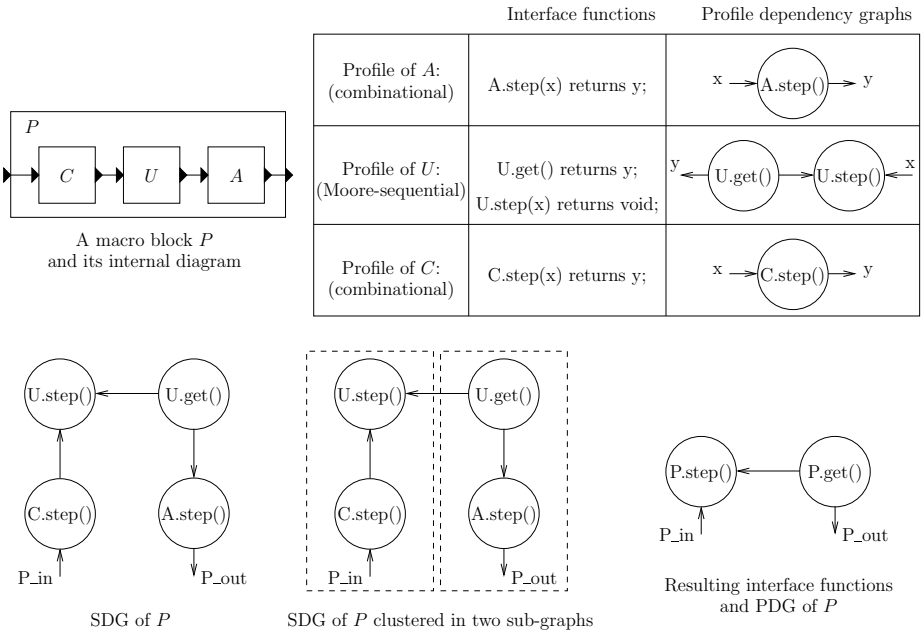


Fig. 2. Example of modular code generation.

This example shows a macro (i.e., composite) block P with three sub-blocks, C, U and A. The profiles of these sub-blocks are shown to the top right. A and C

are classified as *combinational* blocks, that is, they have no internal state (given the same input, they always produce the same output). U, on the other hand, is a *sequential* block, that is, it has internal state. Moreover, U is a special sequential block, called *Moore-sequential*, because its output at a given synchronous instant depends only on its internal state and not on the value of its input at that instant. For example, U could be the *unit-delay* block, where the value of the output at time k is equal to the value of its input at time $k-1$ (there is an initial value for $k=0$).

Figure 2 also shows the signatures of each interface function of the sub-blocks of P, plus the PDG of each sub-block. The PDG for block U is interesting: U has two interface functions, U.get() and U.step(). Its PDG states that U.get() must be called before U.step(), at every clock instant. Assuming that U is the unit-delay block, we interpret this constraint as follows: U.get() returns the current value of the internal state, while U.step() updates the internal state with the input. The implementation of these two functions could be as follows:

```
U.get() returns y
{
  y := state;
  return y;
}
```

```
P.step( x ) returns void
{
  state := x;
}
```

where "state" is the internal variable that represents the internal state of the block.

Figure 2 also illustrates the essential steps in our code generation method. The first step, shown to the bottom-left of the figure, is to construct the *scheduling dependency graph* (SDG) of the block P for which we want to generate code. The SDG is built by connecting the PDGs of the sub-blocks of P according to the internal diagram of P. For example, since the output of C is connected to the input of U, we insert an edge from node C.step() (which produces the output of C) to node U.step() (which consumes the input of U).

Once the SDG of P is formed, we check whether it is *acyclic*, that is, whether there are no dependency cycles between the interface functions of the sub-blocks. If there are no cycles, then a well-defined order for executing these functions exists. Otherwise, the diagram is rejected and code generation fails. This would happen, for instance, in the case of block P shown in Figure 1, if a single function was generated for this block and the block was connected as shown to the middle or right of Figure 1.

If the SDG is acyclic, the next step is to perform SDG *clustering*. The clustering operation "splits" the SDG of P into a set of *sub-graphs*. In the case of the example of Figure 2, clustering results in two sub-graphs. Clustering is at the heart of our code generation method. Exactly how clustering is performed depends on the clustering algorithm used. There is not one, but many different clustering algorithms that can be used. Different clustering algorithms will result in different trade-offs, as explained in [1, 3] and briefly discussed below.

Once the SDG of P is clustered, code is generated. For each sub-graph of P produced by clustering, one interface function is generated. Each sub-graph G_i is guaranteed to be acyclic, since the SDG itself is acyclic. The interface function generated from G_i consists essentially in calling all sub-block interface functions included in G_i in a given order that respects the dependencies of G_i.[5]

In the case of the example of Figure 2, the SDG of P is clustered in two sub-graphs as shown in the figure. This means that two interface functions for P will be generated. The implementation of these functions is shown below:

```
P.get( ) returns y
{
  U_out := U.get();
  y := A.step(U_out);
  return y;
}
```

```
P.step( x ) returns void
{
  C_out := C.step( x );
  U.step( C_out );
}
```

After the generation of the interface functions, all that remains is to synthesize a PDG for P. This PDG can be derived directly from the clustering: the nodes of the PDG are the interface functions for P generated above. The edges of the PDG are computed as follows. If sub-graph G_1 depends on sub-graph G_2 (that is, there exists a node v_1 in G_1 and a node v_2 in G_2 such that v_1 depends on v_2 in the SDG of P) then interface function f_{G_1} depends on f_{G_2}. Clustering is done in such a way so as to guarantee that no cyclic dependencies are introduced between sub-graphs, therefore, the SDG is guaranteed to be acyclic. As an example, the SDG of block P of Figure 2 is shown at the bottom-right of the figure. P.step() depends on P.get(), since U.step() depends on U.get().

Apart from the interface functions generated above, for sequential blocks, an init() function is also generated, in order to initialize the state of these blocks. The init() function is not included in the PDG of a block, and is implicitly only called at initialization.

4 Trade-offs

As we mentioned above, by choosing different clustering algorithms, we can explore different trade-offs. A most important trade-off is between *modularity* and *reusability*. Modularity is a quantitative notion in our framework: it can be measured in terms of the number of interface functions generated for a given

[5] Clustering may sometimes result in *overlapping* clusters, that is, sub-graphs that share some nodes. This means that the same sub-block interface function will be "callable" from more than one interface functions of the parent block. In this case, care must be taken to ensure that the sub-block function is only called once at every synchronous instant. This can be done using special flag mechanisms, as explained in [1].

block. The smaller this number, the higher the degree of modularity. Ideally, the most modular code is one that has just one interface function.

The price to pay for modularity is reusability. If we generate too few interface functions for a given block P, we may create additional input-output dependencies that were not in the original internal diagram of P. These dependencies may result in false dependency cycles when we later embed P as a sub-block in a certain higher-level diagram. We already gave an example that illustrates this phenomenon: see Figure 1 and corresponding discussion.

We say that *maximal reusability* is achieved when no false input-output dependencies are added during clustering. In [1] we have proposed the so-called *dynamic* clustering method that achieves maximal reusability with a minimal number of interface functions. In that sense, this method is optimal. Moreover, the dynamic method is guaranteed to generate no more than $n + 1$ interface functions for a block with n outputs (no more than n functions if the block is combinational). In [1] we have also proposed another clustering method, called the *step-get* method, which generates at most two (and often just one) interface functions per block. The step-get method privileges modularity but obviously cannot guarantee maximal reusability.

In more recent work [3], we explore another trade-off: modularity vs. code size. We illustrate this trade-off through an example, shown in Figure 3. The figure shows a macro block P with 3 sub-blocks, A, B, C. We suppose that each of these blocks has a single interface function step(), abbreviated by s (so $A.s$ stands for A.step(), etc). The PDG of each of the sub-blocks then clearly consists of a single node.

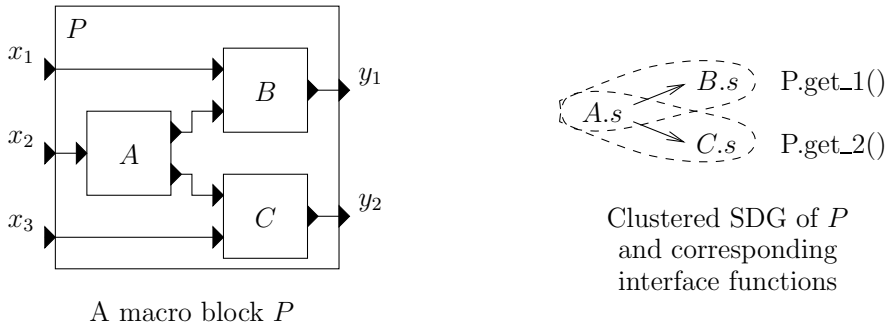


Fig. 3. Illustration of the dynamic method.

For this example, the dynamic method performs the clustering shown to the right of Figure 3: the SDG of P is clustered in two overlapping sub-graphs. Node $A.s$ is shared by the two sub-graphs. This sharing results in replication of code across the interface functions generated for P, plus additional overhead to ensure that the shared sub-block functions (A.step() in this case) are called only

once at each synchronous instant. This overhead results in increased code size. In [3] we explore how to generate code of minimal size, by avoiding overlapping during clustering. The price to pay is modularity: in general, if clustering is restricted to be disjoint (i.e., without overlapping) then more interface functions may need to be generated in order to achieve maximal reusability. This is the case for the example of Figure 3: instead of two overlapping sub-graphs, we need three disjoint sub-graphs (one for each node in the SDG). Moreover, the optimal disjoint clustering problem is NP-hard, as shown in [3].

5 Extension to multi-rate synchronous models

So far we have discussed modular code generation for a purely synchronous block diagram model, where all blocks are "fired" at every instant. Our method can be extended to a *multi-rate* synchronous model, by considering block diagrams with *triggers* [2]. In this extended model, any block may be *triggered* by a Boolean signal produced from other blocks in the diagram (or directly from an input signal). This feature allows us to build multi-rate designs where different parts of the diagram are fired at different times.

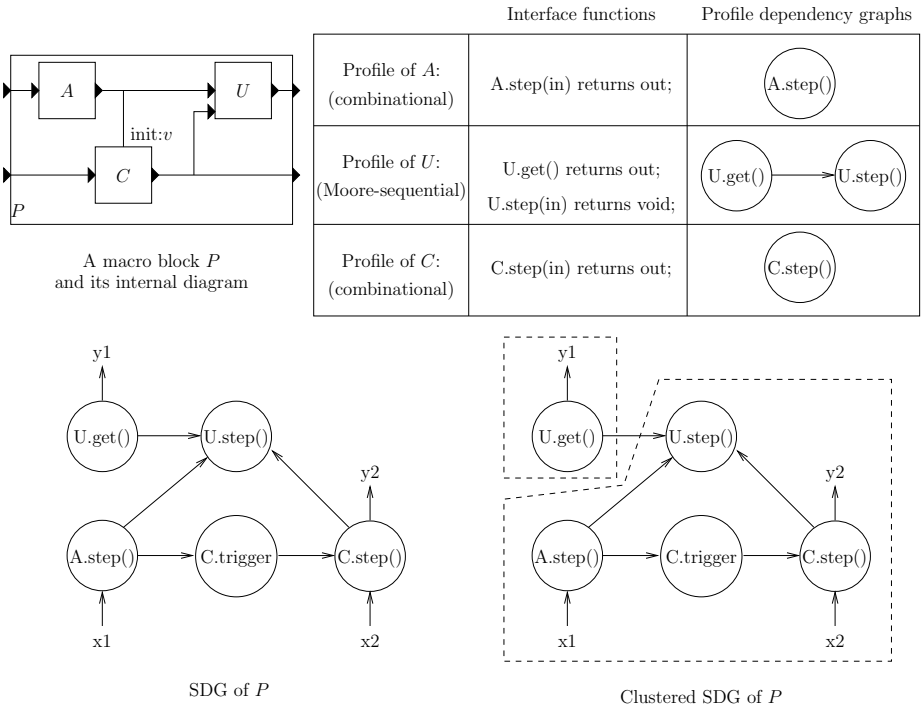


Fig. 4. Modular code generation for diagrams with triggers.

An example of a block diagram with triggers is shown in Figure 4. Block C is triggered by the output of A. This means that C is "fired" only at those synchronous instants where the output of A (which must be of type Boolean) is true. Otherwise, C is not fired and its output remains constant (i.e., keeps its value at the previous instant). The notation "init:v" is a specification of an initial value v for the output of C, for the case when the trigger is initially false.

Triggers do not increase the expressive power of the model: indeed, it is shown in [2] how triggers can be eliminated while preserving the semantics of the model. However, this elimination procedure is done in a top-down, recursive manner, where all blocks of the hierarchy are visited, up to the atomic blocks. Therefore, this elimination procedure does not preserve the "black-box" abstraction of a block, which is important for IP reasons as explained in the introduction. For this reason, we opt for handling triggers directly instead of eliminating them.

It turns out that the code generation steps discussed above can be easily extended to handle triggers. Essentially, it suffices to add extra nodes and dependencies in the SDG of a macro block for its triggered sub-blocks; and then add conditional if-statements in the generated code. We illustrate these extensions in the case of the example shown in Figure 4. Because block C is a triggered block, we add the node C.trigger to the SDG of P: this node depends on A.step() which produces A's output. In turn, C.step() depends on C.trigger, since the value of the trigger needs to be known in order to decide whether to fire C or not.

After clustering this SDG in two sub-graphs as shown to the bottom-right of Figure 4, two interface functions are generated for P as shown below:

```
P.get1( ) returns y1
{
  y1 := U.get( );
  return y1;
}
```

```
P.get2( x1, x2 ) returns y2
{
  A_out := A.step( x1 );
  if (A_out) then
    y2 := C.step( x2 );
  end if;
  U.step( A_out, x2 );
  return y2;
}
```

Another extension considered in [2] is *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).

An example is shown in Figure 5. In this case, FTSs are represented as (period, initial phase) pairs: we abbreviate a (period, phase) pair as PPP. 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, ...

Since timed diagrams are special cases of triggered diagrams, we could simply use the code generation scheme described in the previous sections. However, it

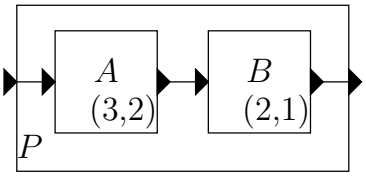


Fig. 5. A timed block diagram

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.

A PPP is not the only possible FTS representation. Indeed, in [2] we proposed an alternative representation in terms of finite automata. To see the motivation behind this, consider again the example shown in Figure 5. 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.

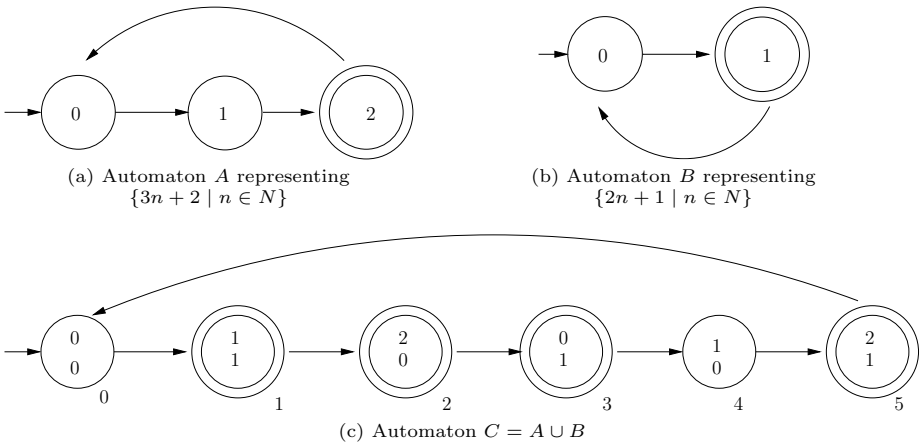
Now, suppose we wanted to represent the times when P needs to be executed as a PPP. Then, the period of this PPP cannot be greater than 1, otherwise P will "miss" some executions of either A or B. Also, the initial phase of this PPP cannot be greater than 1, otherwise the initial execution of B will 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.

To remedy this, we can use a representation in terms of finite automata, called *firing time automata* (FTA). A PPP (π, θ) represents the set of times $\{\pi \cdot n + \theta \mid n \in N\}$. Two FTA corresponding to PPPs $(3, 2)$ and $(2, 1)$ are shown in Figures 6(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 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 6(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. For a more detailed description of FTA and their properties, see [2].



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

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

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

Fig. 6. Automata representing firing times.

Having an explicit representation of the firing times of a block allows for a more refined dependency analysis. In particular, when constructing the SDG for a macro block P, some dependencies in this SDG can be removed by considering the firing times of the corresponding sub-blocks of P. To illustrate this, consider the example shown in Figure 7. Suppose 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, ... This can be discovered automatically by checking whether the intersection of the firing times of A and B is empty. When these firing times are represented as FTA the check can be implemented by computing the intersection of the automata and checking for emptiness.

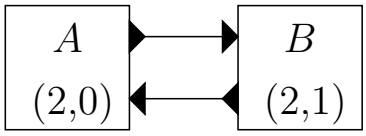


Fig. 7. A timed diagram with false dependencies

6 Conclusions

We have proposed methods for the automatic generation of modular code from synchronous block diagram notations. Modularity means code is generated independently from context, that is, independently from the diagram in which the block is going to be used. The essential mechanisms to achieve modular code generation are: (a) an abstraction of each block in terms of a profile, which includes the interface of the block and a set of constraints on the usage of this interface, and (b) a set of clustering algorithms that allow to construct the interface of a parent block based on the interfaces of its sub-blocks. Different clustering algorithms can be used to explore different trade-offs in terms of modularity, reusability, code size, and so on.

References

1. R. Lublinerman and S. Tripakis. Modularity vs. Reusability: Code Generation from Synchronous Block Diagrams. In *Design, Automation, and Test in Europe (DATE'08)*. ACM, March 2008.
2. R. Lublinerman and S. Tripakis. Modular Code Generation from Triggered and Timed Block Diagrams. In *14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*. IEEE CS Press, April 2008.
3. R. Lublinerman, C. Szegedy, and S. Tripakis. Modular Code Generation from Synchronous Block Diagrams: Modularity vs. Code Size. Submitted.
4. S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, 1994.
5. T.R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference (EDTC'96)*. IEEE Computer Society, 1996.
6. 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.
7. E.A. 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.
8. P. Raymond. Compilation séparée de programmes Lustre. Master's thesis, IMAG, 1988. In French.
9. A. Benveniste, P. Le Guernic, and P. Aubry. Compositionality in dataflow synchronous languages: specification & code generation. Technical Report 3310, Irisa - Inria, 1997.
10. O. Hainque, L. Pautet, Y. Le 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.
11. P. Mosterman and J. Ciolfi. Interleaved execution to resolve cyclic dependencies in time-based block diagrams. In *43rd IEEE Conf. on Decision and Control (CDC'04)*, 2004.
12. 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.
13. P. Aubry, P. Le Guernic, and S. Machard. Synchronous distribution of Signal programs. In *29th Intl. Conf. Sys. Sciences*, pages 656–665. IEEE, 1996.