Modularity vs. Reusability: Code Generation from Synchronous Block Diagrams*

Roberto Lublinerman†
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
rluble@psu.edu

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

Abstract

We present several methods to generate modular code from synchronous hierarchical block diagrams. Modularity means code is generated for a given macro (i.e., composite) block independently from context, that is, without knowing where this block is to be used, and also with minimal knowledge about its sub-blocks. We achieve this by generating a set of interface functions for each block and a set of dependencies between these functions that is exported along with the interface. The main trade-off is the degree of modularity (number of interface functions) vs. reusability (the set of diagrams that the block can be used in without creating dependency cycles).

1 Introduction

Block diagrams are a popular notation, implemented in a number of successful commercial products such as Simulink from The MathWorks [1] or 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. For IP issues, the internal structure of a macro block may be "hidden" from its user.

In such a context, *modular* code generation becomes a critical issue, and this is the goal of this paper. By modular we mean: first, code for a macro block should be generated *independently from context*, that is, without knowing where (i.e., 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.

Current code generation practice 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, code can be generated by statically ordering the execution of atomic blocks so that all dependencies are respected.[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.

To remedy this, we propose several methods to regain modularity. The main idea is to generate, for a given block, not just one "step" function that computes the outputs (and updates the state, if any) from the inputs, but a *set of interface functions*, each evaluating part of the block and/or computing part of the outputs. A *set of dependencies between these functions* is also exported: these specify the correct usage of the interface, that is, the order in which

*We would like to thank Allen Goldberg, Yaron Kashai and Guang Yang from Cadence for motivating this work and providing valuable feedback. We also thank Paul Caspi from Verimag and Reinhard von Hanxleden from Kiel University for providing references to related work.

†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/

[3] Cyclic diagrams can also be handled, e.g., see [4, 8, 6]. This is useful in some applications (e.g., digital circuits) but avoided in others (e.g., embedded control). In this paper we consider diagrams that, if flattened, are acyclic.

the functions should be called.

Using this approach, modularity becomes a *quantifiable* notion, that can be measured by the *size of the interface* of a block (number of interface functions and their dependencies): the smaller the interface, the more modular it is. In the best case the block has just one step function (plus "init" if the block has memory). The price to pay for modularity is *reusability*: the smaller the interface, the greater the chances to create dependency cycles when attempting to use the block in a certain diagram, and thus reject the diagram.[4] Nevertheless, in this paper we propose the so-called "dynamic" method which allows to achieve *maximal reusability* (i.e., accept all diagrams that are acyclic when flattened) while generating a minimal number of interface functions per block. Moreover, the dynamic method generates at most $n + 1$ interface functions per block, where n is the number of outputs of the block. The dynamic method has polynomial worst-case time complexity.

Related work

Code generation for notations with synchronous semantics has received great attention, especially from the synchronous language community (e.g., see [2, 8]). Modular code generation, however, has been much less studied, in fact, it is often considered to be impossible in the general case. This is true if we restrict ourselves to single-function interfaces, but not if we allow multi-function interfaces, as we show here.

The need for multi-function interfaces has been realized in [9, 3, 5]. [9, 5] start from a very fine-grain interface where every atomic operator is mapped to an interface function, and then use methods to reduce the number of functions by "clustering" operators together. This approach generates a larger number of interface functions than our dynamic approach. It is also unclear how expensive their grouping algorithms are. Our algorithms are polynomial in time. [3] discuss a method that clusters operators depending on the same set of inputs. Their discussion is rather informal and does not address the issues of maximal reusability, number of generated interface functions, and complexity of algorithms. None of the above works use the concept of *Moore-sequential* blocks (see Section 2) which allows to extend the class of diagrams that can be handled by single-function interfaces, while maintaining a high degree of modularity.

Multi-function interfaces are also used in the simulation environment Ptolemy, where each "actor" (essentially block) must implement a set of functions such as "prefire", "fire", "post-fire" and so on [6]. However, the set of interface functions is fixed and does not depend on the internal structure of the macro block, as in our approach. Also, in our case, interface functions of sub-blocks are called at most once per instant, in a statically determined order, whereas in Ptolemy they can be called multiple times until a fix-point is reached: this allows Ptolemy to handle dependency cycles whereas we don't.

Less related are the works [7, 1] which consider the problem of *distribution* of synchronous programs. Distribution does not necessarily imply modularity: for instance, one may look at the entire program (e.g., flatten it) in order to distribute it, which is the approach taken in the works above. A formal model for the distribution of Simulink programs is proposed in [12], however, multi-function interfaces are not considered.

Different notions of "modular" compilation are studied in [11, 10]. [11] consider the partial evaluation of Esterel programs: they generate code that tries to compute as many outputs as possible, while some inputs may be still unknown. [10] consider a language similar to Esterel called Quarz and its compilation to a target "job language".

To our knowledge, commercial code generators, such as the Real Time Workshop and Embedded Coder for Simulink and the DO-178B certified code generator for SCADE, offer limited, if any, modular code generation capabilities. For instance, RTWEC for Simulink provides a feature called "Function with separate data" but does not seem to generate multiple interface functions per block, neither a dependency interface, both of which are essential in our methods.

2 Synchronous block diagrams

The notation is based on a set of *blocks* that can be connected to form *diagrams*, as illustrated in Figure 1. 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.

A macro block encapsulates a block diagram into a block. The blocks forming the block diagram are called the *internal* blocks of the macro block, or its *sub-blocks*. The connections between blocks ("wires") are called *signals*. Like an atomic block, a macro block has a set of input and output ports. 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 and Moore-sequential blocks

Each atomic block A is pre-classified as either *combi-*

[4] Contrasting modularity to reusability may appear shocking. However, it should be clear that reusability requires information about the internals of a block. In the extreme case, the most reusable block is a "white box" which reveals all the information about it. This is clearly not modular.

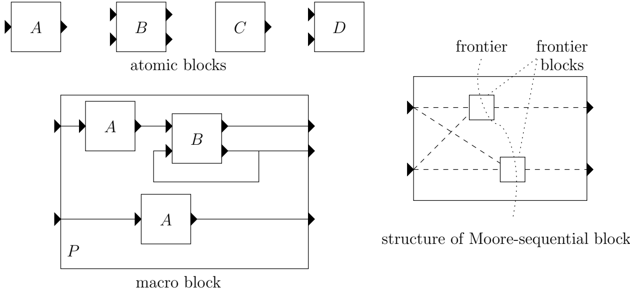

Figure 1. Atomic and macro blocks (left); structure of Moore-sequential block (right).

national (state-less) or *sequential* (having internal state). Some sequential blocks are *Moore-sequential*. Every output of a Moore-sequential block *depends only on the state, 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 1, if block A is Moore-sequential then macro block P is also Moore-sequential.

Every Moore-sequential macro block A has a structure as shown in Figure 1. There is a *frontier* dividing the internal diagram of A in two parts, a *left* and a *right* part. This frontier contains all Moore-sequential blocks (called *frontier blocks*) that are "met" by the backwards paths described above. The right part contains all blocks that are visited in one of these paths. The left part contains all remaining blocks.

Notice that a block with no outputs is by definition Moore-sequential. Such a block has no paths from the outputs, thus it has no frontier blocks and no right blocks either: all its sub-blocks are left blocks. Similarly a block with no inputs is also Moore-sequential.

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.

Block-based dependency analysis and acyclic diagrams We use different types of dependency analysis in this paper. The one described here is standard: we use it 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 which are all 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. Another type of dependency analysis, used for code generation, is described in Section 3.1.

Semantics We only assign semantics to flat, acyclic diagrams. We use the standard synchronous semantics. Each signal x of the diagram is interpreted as a total function $x : N \to V_x$, where $N = \{1, 2, 3, ...\}$ and V_x is a set of values: $x(k)$ denotes the value of signal k 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 computing the values of all signals in a given instant based on the current inputs and possibly the values of signals in the previous instants, encoded in the state of blocks.

3 Modular code generation

We describe several methods. They all fit into a generic code generation scheme. The inputs to this scheme are: (1) A macro block P and its internal block diagram; (2) The *profile* of each sub-block of P (explained below). The outputs of the code generation scheme are: (1) A profile for P. (2) The implementation of each function listed in the profile of P. This implementation can be done in a certain programming language such as C++, Java, etc. For simplicity, we use pseudo-code in this paper.

Profile of a block The block profile contains the necessary information for the user of a block. Both atomic and macro blocks have profiles. The profile of a block A contains: (1) The *type* 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. We give several examples of block profiles in the sequel.

Object-oriented code The code we generate is object-oriented: for each block P we build a *class* in a certain object-oriented language (C++, Java, ...). The public methods of this class correspond to the interface functions of P. The class of P also contains a set of internal (private) variables: there is an internal variable for each output port of P; there is also an internal variable for each internal signal

of P. All these variables are *persistent* meaning they maintain their values across execution of the different interface functions.

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 in Section 2. (2) *Dependency analysis*: this step (described in Section 3.1) determines whether there exists a valid execution order of the interface functions of the sub-blocks of P. (3) *Profile generation*: this is the main step, described in Section 3.2.

3.1 Dependency analysis

Dependency analysis 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. If the SDG contains a cycle then P is *rejected*: this means that modular code generation fails and P needs to be flattened (the flat diagram may or may not contain cycles). Otherwise, we proceed to the code generation step.

The SDG for P is built essentially by connecting the PDGs of all sub-blocks of P. In particular:

For each sub-block A of P, the SDG of P contains all nodes and edges of the PDG of A. The SDG of P has the following additional edges: If A and B are sub-blocks of P, such that an output port y of A is connected to an input port x of B, then: Let $A.f()$ be the interface function of A producing output y: this function is guaranteed to be unique. Let $B.g()$ be an interface function of B having as input x: in general there can be more than one such functions of B. For each such function $B.g()$, we add an edge $A.f() \to B.g()$ to the SDG of P.

Example Figure 2 shows a block diagram with several blocks and the profiles of blocks A, B (given as input) and P (generated by our methods). Block A has only one interface function A.step() while B has two functions, B.step() and B.get(). B.step() returns no output, while B.get() takes no input. The SDG for macro block P, built as described above, is shown at the bottom of the figure.

3.2 Profile generation

This step involves several sub-steps. The most important one is SDG *clustering*. Let G be the SDG of the macro block P (built during the dependency analysis step). G is clustered into a set of sub-graphs. For example: in Figure 2 the SDG is clustered in two sub-graphs, called "left" and "right"; in Figure 3 the SDG is clustered in two sub-graphs (note that in this case the sub-graphs "overlap").

Each of the sub-graphs is going to be mapped into an interface function for P: this function calls all sub-block

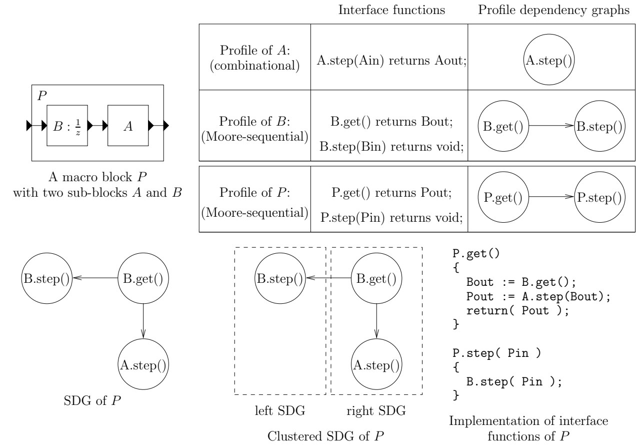

Figure 2. Diagram, profiles and SDGs

interface functions that belong to its sub-graph. Dependencies between nodes of G induce dependencies between its sub-graphs: a sub-graph G_i depends on G_j, denoted $G_j \to G_i$, if there are two nodes v_i in G_i and v_j in G_j such that $v_i \to v_j$ in G. A dependency between sub-graphs is mapped into a dependency between the corresponding interface functions, and this is how the PDG of P is generated. For example, in Figure 2 the two sub-graphs "left" and "right" are mapped into P.step() and P.get(), respectively. The PDG of P is shown in the figure: it contains a dependency P.get() → P.step() induced by the dependency B.get() → B.step().

Clustering must meet certain requirements: (1) it must not create cyclic dependencies between sub-graphs; (2) it must not create new input-output dependencies, that were not already induced by G: we call these *false* dependencies. Requirement (1) must be satisfied in all cases, otherwise the method is obviously invalid (since the PDG of P contains cycles, thus P is unusable). Requirement (2) is essential for achieving maximal reusability: false dependencies may result in dependency cycles when using the block, and consequently rejections of the corresponding diagrams. Requirement (2) may be sometimes violated, as a trade-off of reusability for modularity, that is, in order to reduce the number of interface functions that are generated for a given block. We give examples of this later in this paper.

In what follows, we describe different methods for performing the steps of clustering in particular and profile generation in general. These methods result in different trade-offs between modularity and reusability.

3.2.1 The "dynamic" method

This method achieves *maximal reusability*: it uses a clustering that is guaranteed to create no false dependencies.

As a result, this method accepts all acyclic diagrams. The method is also *optimal* with respect to modularity (given the constraint that no false dependencies be created): this means no more interface functions than strictly necessary are generated. Finally, the method is guaranteed to generate no more than $n + 1$ interface functions in the worst case, where n is the number of outputs of the block.

Clustering in the dynamic method is performed according to the following steps:

Input-output dependency analysis Let P be the macro block for which we need to generate code. We first build the *bipartite input-output dependency graph* (BIODG) for P. Let the set of outputs of P be $Y = \{y_1, ..., y_n\}$. For each y_i the BIODG of P captures the set of inputs of P that y_i depends upon: y_i depends on a given input x if there is a path in the SDG of P starting at some node f that takes x as input and ending in the (unique) node that produces y_i as output. Notice that some outputs may not depend upon any input: this is the case in Moore-sequential blocks, for instance. This analysis has polynomial worst-case complexity $O(m \cdot n \cdot b \cdot l)$ where m is the number of inputs of P, n the number of outputs, b is the total number of interface functions of P's sub-blocks, and l is the number of links in the diagram.

Output partitioning The set of outputs Y is partitioned into the minimal number of k disjoint subsets $Y_1, ..., Y_k$, such that for each $i = 1, ..., k$, all outputs in Y_i depend on the same set of inputs. This can also be done in polynomial time $O(m \cdot n^2)$. We denote by X_i the set of inputs that Y_i depends upon.

Clustering in the dynamic method The SDG G of P is clustered into a number of sub-graphs: For each $i \in \{1, ..., k\}$, we build a sub-graph G_i of G. We add to G_i all nodes needed to produce an output in Y_i, even if these nodes have also been included in another sub-graph. If, at the end of the above procedure, there are still nodes of G not included in any of the k sub-graphs, we build an additional sub-graph G_{k+1} and add all remaining nodes to it. Building each graph G_i can be done in time $O(b \cdot l)$, so the total complexity of this step is $O(n \cdot b \cdot l)$.

As an example, the SDG of block P of Figure 2 is clustered by the above method in exactly two sub-graphs, "left" and "right", as shown in the figure. In this example, $k = 1$. The right sub-graph corresponds to G_1 and the left sub-graph to G_2. In fact, it can be seen that for any Moore-sequential block, the above clustering procedure results in exactly two sub-graphs: $k = 1$ since no output depends upon any input; and an additional sub-graph G_2 is need to update the state of the block.

Another example of clustering is shown in Figure 3. We assume all internal blocks A, B, C are combinational and each has one interface function, denoted in the figure as $A.s$, $B.s$, $C.s$, for A.step(), B.step() and C.step(). In this example, the SDG of P is clustered in two sub-graphs.

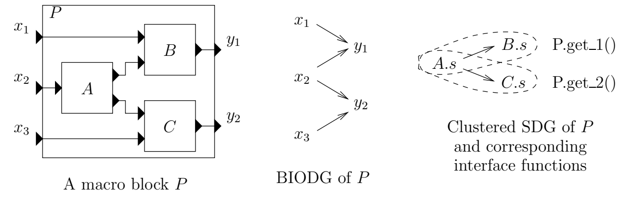

Figure 3. Illustration of the dynamic method.

As shown by this example, the dynamic clustering method may result in a node f being included in more than one sub-graphs. This means that the interface function corresponding to f (f is an interface function of some sub-block of P) can be called by more than one interface functions of P. However, an interface function should be called exactly once per synchronous instant. To achieve this, we use a set of counters defined below.

Assigning modulo counters to internal interface functions For each interface function f of a sub-block of P, let N_f be the number of sub-graphs G_i that the node corresponding to f is included in. If $N_f > 1$ then we create a modulo-N_f counter for f: the counter "wraps" to zero when its value reaches N_f. Each such counter is part of the persistent internal variables of the class of P. The counter for f serves to indicate whether f has already been called in the current instant: *f has been called iff its modulo counter is greater than zero.*

Interface function implementation At the end of the SDG clustering step we are left with k or $k + 1$ sub-graphs of G. Then:

(1) For each sub-graph G_i, for $i = 1, ..., k$, we generate an interface function

```
P.get_i( X_i ) returns ( Y_i );
```

If sub-graph G_{k+1} exists, we generate an interface function

```
P.step( Xs ) returns void;
```

where Xs is the set of inputs that the nodes in G_{k+1} depend upon.

(2) The implementation of interface function P.get_i() or P.step() is obtained by serializing the corresponding sub-graph G_i using a topological sort algorithm. We then call all functions in G_i in the order determined by this serialization and store results in internal signal variables. Every call to a method f that has $N_f > 1$ is guarded by the condition $c_f = 0$, where c_f is the modulo counter associated with f.

PDG generation The nodes of the PDG of P are all interface functions for P generated above. If there is no function P.step() then the PDG of P contains no edges. If there is such a function, then the dependency graph contains an edge P.get_i() $\rightarrow$ P.step() for each interface function P.get_i() of P.

Consider again the example of Figure 3. Function A.step() is included in both sub-graphs that are generated by clustering, thus this function has an associated modulo-2 counter c_A_step. Two interface functions for P are generated, as shown below:

```
P.get_1( x1, x2 ) returns y_1 {
  if (c_A_step = 0) {
    (z1, z2) := A.step( x2 );
  }
  c_A_step := (c_A_step + 1) modulo 2;
  y1 := B.step( x1, z1 );
  return y1;
}

P.get_2( x2, x3 ) returns y_2 {
  if (c_A_step = 0) {
    (z1, z2) := A.step( x2 );
  }
  c_A_step := (c_A_step + 1) modulo 2;
  y2 := C.step( z2, x3 );
  return y2;
}
```

The PDG of P contains two independent nodes, for P.get_1() and P.get_2(), respectively.

The init() function For a sequential block P, an additional function, P.init(), is generated to initialize the state of P. P.init() calls A.init() for every sub-block A of P that is sequential (therefore must also have init() in its interface). The init() functions of sub-blocks of P are called in an arbitrary order. P.init() is not included in the PDG of P since it is called only upon initialization.

3.2.2 The "step-get" method

This method achieves a high degree of modularity: it generates two interface functions for Moore-sequential blocks, like the dynamic method; but only a single interface function for other blocks. This means that the clustering of non-Moore-sequential blocks is trivial: it generates a single sub-graph identical to the original SDG. In fact, this method does not need clustering at all, since Moore-sequential blocks can always be split into "left" and "right" sub-graphs.

This method obviously cannot guarantee maximal reusability, since it may create false dependencies in non-Moore-sequential blocks. However, it still allows to extend the class of blocks that can be handled accurately, for example, the one shown in Figure 2.

3.2.3 Other methods

Other methods could be developed following the principles described above. For example, [9] and [5] start with a fine-grain clustering where every node of the SDG is a sub-graph, and then apply techniques to group nodes into coarser classes. As long as the classes remain disjoint, this approach may result in a larger number of interface functions than our approach. For example, to achieve maximal reusability for the example of Figure 3 using disjoint classes of nodes, one would need three interface functions, one for each sub-block of P.

4 Conclusions and future work

We have presented several modular code generation methods for synchronous block diagrams. The main trade-off is modularity in terms of the number of interface functions per block vs. reusability in terms of the class of diagrams that can be accepted. We are currently evaluating this and other trade-offs in terms of code quality (size, execution time, etc.). We are also studying extensions to other types of diagrams and further modular code generation techniques.

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] D. Potop-Butucaru, S. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.

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

[10] K. Schneider, J. Brandt, and E. Vecchié. Modular compilation of synchronous programs. In *Distr. and Parallel Emb. Sys. (DIPES'06)*, pages 75–84. Springer, 2006.

[11] J. Zeng and S. Edwards. Separate compilation of synchronous modules. In *Embedded Software and Systems (ICESS 2005)*, volume 3820 of *LNCS*. Springer, 2005.

[12] 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.