

Modular Code Generation from Synchronous Block Diagrams

Modularity vs. Code Size

Roberto Lublinerman

Computer Science and Engineering
Pennsylvania State University
rluble@psu.edu

Christian Szegedy

Cadence Research Laboratories
Cadence Design Systems
szegedy@cadence.com

Stavros Tripakis

Cadence Research Laboratories
and Verimag Laboratory – CNRS
tripakis@imag.fr

Abstract

We study modular, automatic code generation from hierarchical block diagrams with synchronous semantics. Such diagrams are the fundamental model behind widespread tools in the embedded software domain, such as Simulink and SCADE. Code is modular in the sense that it is generated for a given composite block independently from context (i.e., without knowing in which diagrams the block is to be used) and using minimal information about the internals of the block. In previous work, we have shown how modular code can be generated by computing a set of interface functions for each block and a set of dependencies between these functions that is exported along with the interface. We have also introduced a quantified notion of modularity in terms of the number of interface functions generated per block, and showed how to minimize this number, which is essential for scalability. Finally, we have exposed the fundamental trade-off between modularity and reusability (set of diagrams the block can be used in).

In this paper we explore another trade-off: modularity vs. code size. We show that our previous technique, although it achieves maximal reusability and is optimal in terms of modularity, may result in code replication and therefore large code sizes, something often unacceptable in an embedded system context. We propose to remedy this by generating code with no replication, and show that this generally results in some loss of modularity. We show that optimizing modularity while maintaining maximal reusability and zero replication is an intractable problem (NP-complete). We also show that this problem can be solved using a simple iterative procedure that checks satisfiability of a sequence of propositional formulas. We report on a new prototype implementation and experimental results. The latter demonstrate the practical interest in our methods.

Categories and Subject Descriptors D.2.3 [Software Engineering]: Coding Tools and Techniques; D.3.2 [Programming Languages]: Language Classifications—Concurrent, distributed, and parallel languages, Data-flow languages; D.3.4 [Programming Languages]: Processors—Code generation

General Terms Algorithms, Design, Languages

Keywords Embedded software, Block diagrams, Synchronous languages, Code generation, Clustering, NP-complete

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM 978-1-60558-379-2/09/01...\$5.00

1. Introduction

Programming may be art, craft or science. In any case, the tools used today have greatly advanced since the first days of programming. By tools we mean programming languages and compilers, but also “support” tools such as debuggers, static analyzers, etc. The most important class of tools are probably the programming languages themselves, since they are the primary means of capturing the intent of the designer (or programmer). Understandably, as these means become more powerful, by raising the level of abstraction, the designer’s productivity increases.

In the field of embedded, real-time systems, like in many other fields, specialized (sometimes called “domain-specific”) languages are used. Simulink from The MathWorks¹ and SCADE from Esterel Technologies² are two successful commercial products in this field. They are especially widespread in the automotive and avionics domains.

These tools offer a mix of modeling/programming model for the design and implementation of embedded software. They provide environments which include graphical model editors, simulators and code generators. Automatic generation of code that implements the semantics of a model is useful in different contexts: the code can be used for simulation; but it can also be embedded in a real-time digital control system (*X-by-wire*). In fact, usages of the latter type are increasingly adopted by the industry. Thus, these tools can be seen as programming languages, debuggers and compilers for the embedded software domain.

The fundamental model behind notations such as the above is that of *synchronous block diagrams*. The latter are hierarchical dataflow diagrams with a synchronous semantics, similar to that of synchronous languages such as Lustre (Caspi et al. 1987) or Esterel (Berry and Gonthier 1992).

Hierarchy is a powerful concept, used extensively in graphical notations. Hierarchy allows to build designs in a *modular* manner, which is crucial for mastering complexity but also in order to address intellectual property (IP) issues. In a hierarchical block diagram, a set of *atomic* blocks can be connected to form a diagram, and this diagram can be then *encapsulated* into a *macro* (i.e., composite) block. The macro block can itself be connected with other blocks and further encapsulated. An example of a hierarchical block diagram is shown in Figure 1.

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

¹ www.mathworks.com/products/simulink/

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

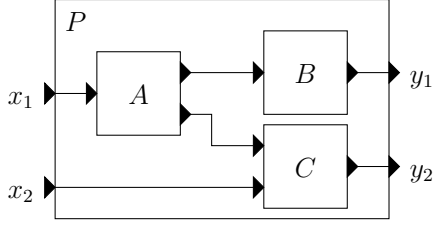


Figure 1. A hierarchical block diagram consisting of a macro block P with three sub-blocks A, B, C .

boxes” supplied with some interface information. The second requirement is very important for IP issues as mentioned above.

Current code generation practice for synchronous block 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 the algorithms 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.

Why is it difficult to avoid flattening and have a truly modular code generation method? Let us illustrate the problem with an example. Consider the macro block P shown in Figure 1. An straightforward way to generate code for P is to generate a *monolithic* piece of code, in the form of a “step” function, as follows:

```
P.step(x1, x2) returns (y1, y2) {
  (z1,z2) := A.step(x1);
  y1 := B.step(z1);
  y2 := C.step(z2, x2);
  return (y1, y2);
}
```

$P.step()$ calls the step functions of the sub-blocks of P , to compute the outputs y_1, y_2 of P , from the inputs x_1, x_2 . Now, suppose P is used as shown to the left of Figure 2: that is, we connect its output y_1 to its input x_2 . Then we find that the function $P.step()$ generated above cannot be used: indeed, in order to call this function we need both x_1 and x_2 , but $x_2 = y_1$, which is an output of the function! In other words, we have a cyclic dependency between inputs and outputs. This dependency, however, is a *false* dependency: it is a result of the above code generation method, and not a property of the diagram. Indeed, flattening P as shown to the right of Figure 2 reveals this: there are no cyclic dependencies in this flattened diagram, and we can compute the output y_2 from the input x_1 by calling the step functions of blocks A, B , and C , in that order.

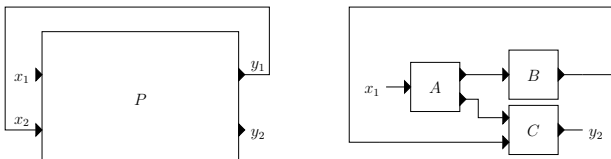


Figure 2. Using block P of Figure 1.

In (Lublinerman and Tripakis 2008b) we have solved this problem, by proposing a general framework for modular code generation. This framework relies on the concept of a *profile* for each block. The profile includes a set of *interface functions* that evaluate

different outputs of the block and may also update the state of the block (if the block has state). The fundamental difference with the monolithic method described above is that the profile generally includes not one, but several interface functions. For instance, in the case of macro block P of Figure 1, we can generate the following two interface functions:

<pre>P.get1(x1) returns y1 { (z1,z2) := A.step(x1); y1 := B.step(z1); return y1; }</pre>	<pre>P.get2(x2) returns y2 { y2 := C.step(z2,x2); return y2; }</pre>
--	--

The profile also includes a *profile dependency graph* that specifies constraints on the order in which these functions need to be called. For the above example, this graph states that $P.get1()$ must be executed before $P.get2()$ (thus making the second output of A available to C via internal variable $z2$).

Our framework allows to quantify modularity of the generated code, in terms of the number of interface functions that are produced: the fewer these functions, the more modular the code is. Our framework also allows to explore the trade-off between modularity and *reusability* (the ability to embed generated code in any context). As the above example illustrates, increasing modularity sometimes means sacrificing reusability.

Our notion of modularity has similarities but also differences from what is typically understood as modularity in programming. It is similar in the sense that it captures the principle of abstracting a component by hiding all information, except what is necessary for the users of the component: block profiles aim at capturing exactly this necessary information. On the other hand, contrasting modularity to reusability may appear surprising. It follows, however, from our choice to quantify modularity as inversely proportional to profile size. This choice is not as arbitrary as it may seem. Indeed, minimizing the size of profiles is important for scalability. In a hierarchical notation such as ours, the complexity of algorithms applied to a given block is a direct function of the size of the profiles of its sub-blocks. Therefore, minimizing the number of interface functions is crucial in managing complexity as we move up in the hierarchy of blocks.

In our framework, adjusting the degree of modularity is done by choosing a certain *clustering* algorithm. This clustering algorithm operates on the so-called *scheduling dependency graph* (SDG) of a given macro block P . The SDG of P consists of all interface functions of sub-blocks of P connected together according to the internal diagram of P . Clustering splits the SDG into a number of sub-graphs, and each sub-graph gives rise to an interface function for P . Therefore, choosing different clustering methods results in exploring different trade-offs. In (Lublinerman and Tripakis 2008b) we presented two clustering methods, the so-called *dynamic* method that achieves maximum reusability and is optimal with respect to modularity (that is, it generates the minimal number of interface functions needed to achieve maximal reusability), and the so-called *step-get* method that generates at most two interface functions, at the expense of reusability.

The first contribution of this paper is the exploration of another trade-off: between modularity and code size. We first show that the dynamic method may result in larger code sizes. This is because the dynamic clustering algorithm allows sub-graphs to overlap, which in turn results in code replication across interface functions. This leads us to study the problem of achieving maximal reusability without code replication, that is, non-overlapping clustering techniques. Our main theoretical result is that *optimal disjoint clustering*, that is, clustering into a minimal number of non-overlapping

sub-graphs without introducing false input-output dependencies, is an intractable problem (NP-complete).

We also show that disjoint clustering generally results in loss of modularity, that is, an increase in the number of interface methods that need to be generated in order to achieve maximal reusability. Finally, we show how the optimal disjoint clustering problem can be solved using a SAT solver. We propose a reduction of checking whether a disjoint clustering exists with a given number of clusters K , to checking whether a given propositional formula is satisfiable. Then, optimal disjoint clustering can be solved by iteratively increasing K .

The second major contribution of this work is a prototype implementation of our framework and a set of experimental results. This implementation is new and includes the optimal disjoint clustering method proposed in this paper as well as the dynamic and step-get methods proposed in our previous work. Using this implementation we experiment on a set of real Simulink examples coming from the Simulink demo suite and from industrial partners from the automotive domain. We find out that many of the problems that our modular code generation framework attempts to solve are not just theoretical problems, but also arise in practice. In particular, we find examples that justify the interest in all three clustering and code generation methods mentioned above.

The rest of this paper is organized as follows. In Section 2 we discuss related work. In Sections 3 and 4 we present the synchronous block diagram notation and our modular code generation framework: this is a summary of our previous work. In Section 5 we illustrate the trade-off between modularity and code size. In Section 6 we study the optimal disjoint clustering problem and show that it is NP-complete. In Section 7 we show how to solve this problem using a SAT solver. In Section 8 we report on our implementation and experiments. Section 9 concludes this paper.

2. Related work

There exists a large body of work on automatic code generation for languages with synchronous semantics, e.g., see (Benveniste et al. 2003). Modular code generation, however, has been not as extensively studied. (Benveniste et al. 2003) stated for this problem that “a unified treatment remains a research topic” and this claim largely remains true today.

The need to depart from monolithic code generation into multiple functions for each module or block has been acknowledged in previous works (Raymond 1988; Benveniste et al. 1997; Hainque et al. 1999), mainly in the context of distribution of synchronous programs. These works, however, provide incomplete solutions. In particular, they omit the concept of profile dependency graph, which is essential to reduce the number of interface functions by expressing dependencies among them. Also, optimality and maximal reusability are not addressed in these works. Finally, clustering with overlapping, which is fundamental to achieve optimality, is not discussed in any of these works. For instance, (Hainque et al. 1999) 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.

Industrial tools such as Simulink or SCADE offer limited modular code generation capabilities. In SCADE the main problem with modular code generation, namely cyclic dependencies, is avoided by requiring that every feedback loop be “broken” by a unit-delay block *at every level of the hierarchy*. This is a major modeling restriction, as most diagrams in practice exhibit feedback loops with no such delays at higher levels of the hierarchy. The same restriction is imposed in (Biernacki et al. 2008). Indeed, a single

“step” function is generated per block in these works, but this is not enough to guarantee maximal reusability as explained above.

Real-Time Workshop Embedded Coder, a code generator for Simulink by The MathWorks, 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 as mentioned above.

In (Mosterman and Ciolfi 2004) a solution reminiscent of our step-get method is presented and reported to be implemented in Simulink. This solution generates only two interface functions per block (an “output” and an “update” function), thus cannot achieve maximal reusability. Indeed, maximal reusability requires in general as many as $n + 1$ functions, where n is the number of outputs of a block (Lubliner and Tripakis 2008b).

Less related are the works (Maffeis and Le Guernic 1994; Aubry et al. 1996) which consider the problem of *distribution* of synchronous programs. Distribution does not necessarily imply that the generated code is modular: 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.

Trade-offs between modularity and reusability or modularity and code size are not discussed in any of the aforementioned works. To our knowledge, this paper is the first to study the problem of minimizing code size in the context of modular code generation for block diagrams.

In this paper we use a simple, “purely synchronous” model of block diagrams, where all blocks run at the same rate. This is done for reasons of simplicity. In (Lubliner and Tripakis 2008a) we have shown how to extend our modular code generation framework to *triggered* and *timed* block diagrams: these features allow one to describe *multi-rate* models. The optimal disjoint clustering method proposed in this paper can be readily used in triggered and timed block diagrams as well.

In this paper we consider diagrams that, if flattened, are acyclic. Cyclic diagrams can be handled using the approach of (Malik 1994; Shiple et al. 1996), which is to check at compile-time that, despite cyclic dependencies, the diagram still has well-defined semantics. This, however, requires semantic knowledge about each block, namely, what is the function that each block computes. Having such semantic knowledge is contrary to the requirement of treating blocks as “black-boxes” due to IP reasons. It is also possible to avoid such compile-time checks and rely on computing a fixpoint at run-time (Edwards and Lee July 2003; Lee and Zheng 2007). However, this fixpoint may contain undefined values, which means such code cannot be used in safety-critical applications.

3. Synchronous block diagrams

In this section we describe our notation which is a simple form of hierarchical block diagrams, and its semantics, which is synchronous.

The notation: We consider a notation based on a set of *blocks* that can be connected to form *diagrams* (see Figure 1). Blocks are either *atomic* or *macro* (i.e. composite). 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 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: Every atomic block A is pre-classified as either *combinational* (stateless) or *sequential* (having internal state). Some sequential blocks are *Moore-sequential*. Each output of a Moore-sequential block *only depends on the current state, but not on the current inputs*. For example a *unit-delay* block that stores the input and provides it as output in the next synchronous instant is a Moore-sequential block. On the other hand, a block implementing an equation of the form $y_k = ax_k + bx_{k-1}$, where y is the output stream, x is the input stream, a, b are constants and k is the index of the current synchronous instant, is a non-Moore sequential block. The above definitions extend to macro blocks in a natural way.

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. Notice that since the diagram is flat, all these blocks are atomic. 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 assign semantics only 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 $x : N \rightarrow V_x$, where $N = \{0, 1, 2, 3, \dots\}$ 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 at a given synchronous instant.

4. Modular code generation

In (Lublinerman and Tripakis 2008b) we have proposed a modular code generation scheme for the diagrams described in the previous section. Here, we present a summary of this scheme and show the trade-offs that can be explored with it.

Modular code generation — inputs and outputs: Our code generation scheme takes as inputs:

1. a macro block P with 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: a list of *interface functions* and their *signatures*; and 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, U, 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 U has two interface functions: $U.step()$ and $U.get()$. $U.get()$ takes no input and returns the output of U . $U.step()$ takes the input of U and returns no output. This is a case where U is a Moore-sequential block: its $get()$ method returns the outputs and its $step()$ method updates the state, given the inputs. The PDG of U shown in the figure states that $U.get()$ must be called before $U.step()$, at every synchronous instant.

Code generation steps: Our modular code generation scheme contains two major steps, explained briefly and illustrated through the example of Figure 3:

(1) *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*³ (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 . For example, the SDG for the block P shown in Figure 3 is shown in the bottom left of the figure. 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.

(2) *Profile generation:* this step involves several sub-steps, the most important of which is *clustering* the SDG into a number of *sub-graphs*. How exactly to cluster the SDG is at the heart of our modular code generation scheme: choosing different clustering methods results in different specializations of the scheme, with different trade-offs as discussed below. An example of clustering is shown in Figure 3. The SDG of P is clustered in two sub-graphs, one containing functions $U.step()$ and $C.step()$, and the other containing $U.get()$ and $A.step()$.

Once the SDG is clustered into a set of sub-graphs, for each sub-graph G , we generate one interface function f_G for P . Function f_G calls the functions included in G in the order specified by G . This is a partial order, and any serialization of it into a total order is legal. An $init()$ function is also generated for sequential blocks to initialize their state. In the example of Figure 3, the SDG of P is clustered in two sub-graphs. Each of these sub-graphs gives rise to an interface function. The left-most sub-graph gives rise to a function called $P.step()$ and the right-most one to a function called $P.get()$. The implementation of these functions is shown below:

<pre> P.get() returns P_out { U_out := U.get(); P_out := A.step(U_out); return P_out; } </pre>	<pre> P.step(P_in) { C_out := C.step(P_in); U.step(C_out); } </pre>
---	---

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

³Note that the scheduling dependency graph (SDG) and the profile dependency graph (PDG) are two different objects.

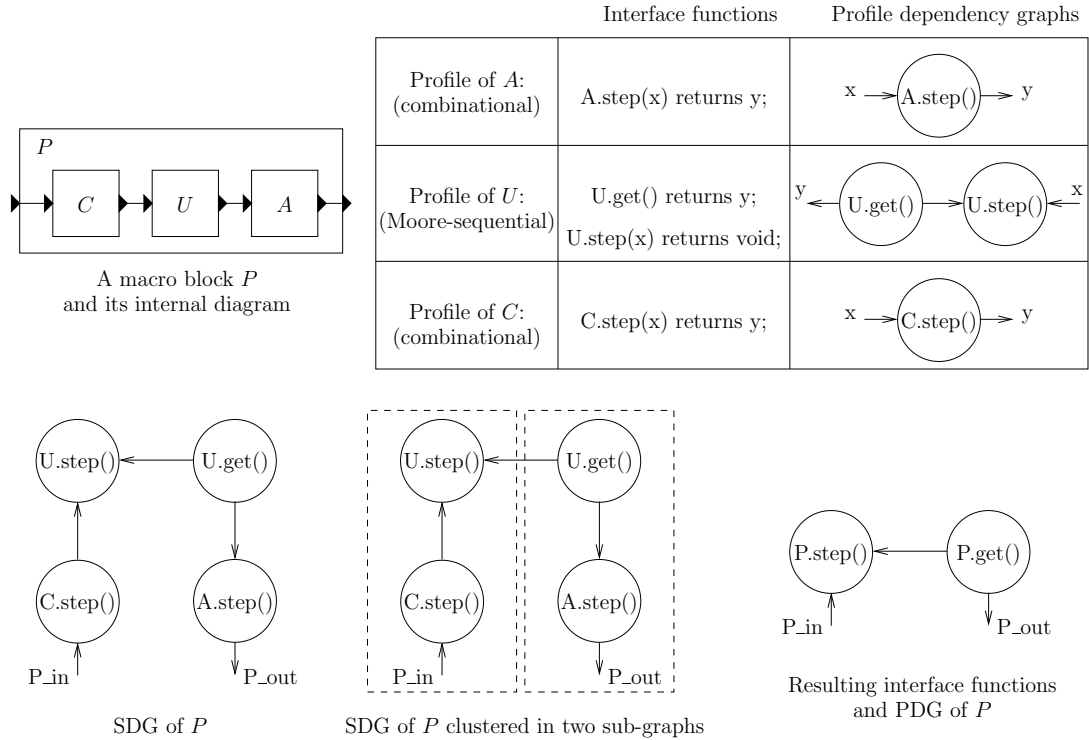


Figure 3. Example of modular code generation.

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 3 is shown at the bottom-right of the figure. $P.step()$ depends on $P.get()$, since $U.step()$ depends on $U.get()$.

Modularity vs. reusability trade-off: 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 can be made a quantitative notion in our framework: it can be measured in terms of the number of interface functions generated for a given block. The smaller this number, the higher the degree of modularity. In that sense, the most modular code is one that has just one interface function. This definition is justified by complexity considerations. The complexity of algorithms such as cycle detection or clustering, that our method uses, is a direct function of the sizes of the PDGs of the sub-blocks of a given macro block. In turn, the size of the PDG of a block is a function of the number of interface functions generated for this block (these functions are the nodes of the PDG). Therefore, minimizing the number of generating functions is essentially for scalability.

The price to pay for modularity is reusability.⁴ If we generate too few interface functions, we may create additional, false input-output dependencies, that were not in the original diagram. These dependencies may result in false dependency cycles when we later embed macro block P in a certain diagram. An example illustrating this phenomenon has been given in the Introduction (see Figures 1 and 2). Another example is given here. Suppose we generate only

one, instead of two, interface function for block P of Figure 3. In terms of clustering, this means we cluster the SDG of P into a single sub-graph that contains all nodes. We would then generate a single interface function for P , say $P.step()$, as follows:

```

P.step( P_in ) returns P_out {
  U_out := U.get();
  C_out := C.step(P_in);
  P_out := A.step(U_out);
  U.step( C_out );
  return P_out;
}

```

The problem is that we have created a *false dependency* between the output of P and its input: this dependency did not exist in the original diagram because U is Moore-sequential. False input-output dependencies can be harmful in terms of reusability. For example, if a user of P attempts to connect the output of P to its input, to form a feedback loop, then the above single-function interface cannot be used. We say that *maximal reusability* is achieved when no false input-output dependencies are added during clustering.

In (Lublinerman and Tripakis 2008b) 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 (Lublinerman and Tripakis 2008b) 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. For sequential blocks, an *init()* function is added to initialize the state.

⁴ Note that profiles and PDGs are essentially *abstractions* of the internals of a block. As with most abstractions, they can be more or less refined, that is, lose less or more information. The more information is lost, the less usable the abstraction becomes.

5. Modularity vs. code size

In this paper, we explore another trade-off: modularity vs. code size. We illustrate this trade-off through an example, shown in Figure 4. Part (a) of the figure shows a macro block P with $n + 2$ sub-blocks, named A_1, \dots, A_n, B, C . We suppose that each of these blocks has a single interface function $\text{step}()$, abbreviated by s . So $A_1.s$ stands for $A_1.\text{step}()$, etc. The PDG of each of the sub-blocks then clearly consists of a single node. By connecting the PDGs of the sub-blocks we form the SDG for P , as shown in part (b) of Figure 4.

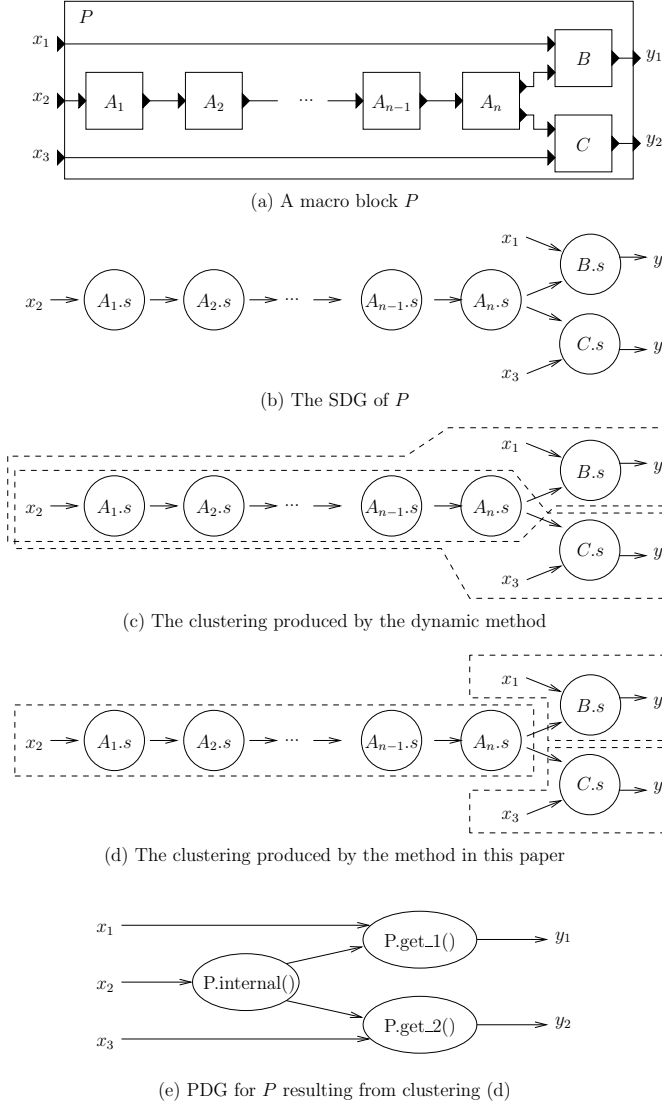


Figure 4. Example illustrating the trade-off between modularity and code size.

Using the dynamic method, the SDG is clustered in two sub-graphs, as shown in part (c) of Figure 4. Notice that this is optimal. Indeed, clustering all nodes into a single sub-graph would create false dependencies from input x_1 to output y_2 and from input x_3 to output y_1 , and that would result in loss of reusability. Also note that the dynamic clustering method may result in sub-graphs that “overlap”, that is, that share some nodes, as is the case with this example. This is essential in order to achieve a minimal number of sub-graphs.

<pre> P.get_1(x_1,x_2) returns y_1 { if (c = 0) { z_1 := A_1.step(x_2); z_2 := A_2.step(z_1); ... z_{n-1}:=A_{n-1}.step(z_{n-2}); (z_B, z_C):=A_n.step(z_{n-1}); } c := (c + 1) modulo 2; y_1 := B.step(x_1, z_B); return y_1; } </pre>	<pre> P.get_2(x_2,x_3) returns y_2 { if (c = 0) { z_1 := A_1.step(x_2); z_2 := A_2.step(z_1); ... z_{n-1}:=A_{n-1}.step(z_{n-2}); (z_B, z_C):=A_n.step(z_{n-1}); } c := (c + 1) modulo 2; y_2 := C.step(z_C, x_3); return y_2; } </pre>
---	---

Figure 5. Code generated with the dynamic method for the example of Figure 3.

<pre> P.internal(x_2) { z_1 := A_1.step(x_2); z_2 := A_2.step(z_1); ... z_{n-1}:=A_{n-1}.step(z_{n-2}); (z_B, z_C):=A_n.step(z_{n-1}); } </pre>	<pre> P.get_1(x_1) returns y_1 { y_1 := B.step(x_1, z_B); return y_1; } P.get_2(x_3) returns y_2 { y_2 := C.step(z_C, x_3); return y_2; } </pre>
---	--

Figure 6. Code generated with the optimal disjoint clustering method (this paper) for the example of Figure 3.

The code that is generated from this clustering is shown in Figure 5. Two interface functions are generated for P , one for each sub-graph. The bodies of the two functions are identical up to their last two statements: this is because both functions may need to evaluate the chain of blocks A_1 to A_n . Internal persistent variables z_1, z_2 , and so on, hold the values of internal signals between blocks A_1 and A_2, A_2 and A_3 , and so on. Variable c is a modulo-2 counter, initialized to zero, that serves to flag whether sub-blocks A_1 to A_n have or have not been fired in the current synchronous instant: if $c=0$ these blocks have not been fired; if $c=1$ they have. All these are automatically generated using the dynamic method: we refer the reader to (Lublinerman and Tripakis 2008b) for the details.

The problem with the code of Figure 5 is that it contains a lot of replication: the first $n + 3$ lines of the bodies of functions $P.\text{get}_1()$ and $P.\text{get}_2()$ are identical. In the worst case, this phenomenon can result in code that is replicated m times, where m is the number of outputs of the block. This clearly results in code size blow-up that may be unacceptable for embedded applications where small footprint is often crucial.

We can eliminate this replication phenomenon, by giving up some modularity: instead of two sub-graphs, we can cluster the graph in three sub-graphs, as shown in part (d) of Figure 4. This clustering produces non-overlapping sub-graphs, thus resulting into interface functions that share no code. The code generated from this alternative clustering is shown in Figure 6. Note that the code of Figure 6 is not only smaller than the code of Figure 5, it is also more efficient. Indeed, it avoids the use of counter c , which results in savings of memory, as well as time (to test the condition $c=0$).

The PDG for P generated for the clustering in three sub-graphs is shown in part (e) of Figure 4.

Note that clustering (d) does not result in any loss of reusability. It does not create any false input-output dependencies, and thus achieves maximal reusability, as does clustering (c). The different between the two clusterings is that (c) privileges modularity instead of code size, while clustering (d) opts for code size at the expense of some modularity.

In the rest of the paper we explore the idea of optimizing code size in depth. It is worth pointing out that the motivating example of Figure 4 is not just an academic example but actually occurs in practice: see Figure 10 and related discussion in Section 8.

6. Optimal disjoint clustering

From the discussion in the previous section, it becomes clear that the trade-off between modularity and code size can be explored by “tuning” the clustering algorithm to use more or less overlapping. The dynamic clustering method proposed in (Lublinerman and Tripakis 2008b) uses as much overlapping as necessary in order to optimize modularity while maintaining maximal reusability. On the other hand, if we want code of minimal size then we should use a clustering that results in *disjoint* (i.e., non-overlapping) sub-graphs. In this section we explore this option. We show that, unfortunately, optimizing modularity while maintaining maximal reusability and disjoint sub-graphs is an intractable problem (NP-hard).

We formalize a Scheduling Dependency Graph (SDG) as a finite directed acyclic graph $G = (V, \rightarrow)$, where V is the set of nodes and $\rightarrow \subset V \times V$ is the set of edges. Both sets are assumed to be non-empty. If $(v, v') \in \rightarrow$ we write $v \rightarrow v'$. V is partitioned in three disjoint non-empty sets

$$V = V_{\text{in}} \cup V_{\text{out}} \cup V_{\text{int}} \quad (1)$$

V_{in} contains the *input* nodes that have no incoming edges: $\forall v \in V_{\text{in}}, \nexists u \in V : u \rightarrow v$. V_{out} contains the *output* nodes that have no outgoing edges: $\forall v \in V_{\text{out}}, \nexists u \in V : v \rightarrow u$. We also assume that output nodes have a unique “writer”, that is, for any $v \in V_{\text{out}}$, the set $\{u \in V \mid u \rightarrow v\}$ is a singleton (i.e., contains a single element). V_{int} contains the *internal* nodes. We assume that there is no direct edge between an input node and an output node (if there is, we can add a new, “dummy”, internal node in-between). Notice that some internal nodes may have no incoming or no outgoing edges. This is necessary in order to model, for instance, SDGs like the one shown in Figure 3. In that example, there are four internal nodes, plus the input node P.in and the output node P.out.

Definition 1 (Clustering). Let $G = (V, \rightarrow)$ be a directed graph. A clustering of G is defined by a partition of V into k disjoint sets of nodes $\mathcal{V} = \{V_1, \dots, V_k\}$ such that $\bigcup_{i=1}^k V_i = V$. Each V_i is called a cluster. We “lift” the relation \rightarrow from nodes to clusters as follows: $V_i \rightarrow_{\mathcal{V}} V_j$ iff there exist $v \in V_i$ and $v' \in V_j$ such that $v \rightarrow v'$. Given $v \in V$, we denote by $[v]$ the (unique) cluster $V_i \in \mathcal{V}$ such that $v \in V_i$. We denote by \rightarrow^* (resp., $\rightarrow_{\mathcal{V}}^*$) the transitive closure of \rightarrow (resp., $\rightarrow_{\mathcal{V}}$). We say that the clustering defined by \mathcal{V} is valid if the following conditions hold:

1. For any $v \in V_{\text{in}} \cup V_{\text{out}}$, $[v]$ is a singleton, that is, input and output nodes are clustered independently.
2. For any $v \in V_{\text{in}}$ and $v' \in V_{\text{out}}$, if $[v] \rightarrow_{\mathcal{V}}^* [v']$ then $v \rightarrow^* v'$. That is, \mathcal{V} does not introduce any input-output dependencies that were not already in G (note that $v \rightarrow^* v'$ implies $[v] \rightarrow_{\mathcal{V}}^* [v']$ by definition).
3. The relation $\rightarrow_{\mathcal{V}}$ on clusters is acyclic (after dropping all self-loops).

We say that the clustering defined by \mathcal{V} is almost valid if Conditions 1 and 2 above hold, while Condition 3 (acyclicity) may not necessarily hold.

The quotient of G w.r.t. \mathcal{V} , denoted G/\mathcal{V} , is the directed graph that has clusters as nodes and whose edge relation is the relation $\rightarrow_{\mathcal{V}}$ on clusters.

Validity captures the fact that a clustering achieves maximal reusability, by not introducing false input-output dependencies. The acyclicity condition ensures that the resulting interface functions do not have cyclic dependencies. Notice that there always exists a

valid clustering for any SDG: the one where every cluster contains a single node.

Proposition 1. *Validity of a clustering can be checked in polynomial time in the number of nodes of the SDG.*

Proof. Validity Condition 1 is trivial to check in linear time in the number of nodes. Condition 2 can be checked by computing the transitive closure of \rightarrow on the nodes of the original graph, then the transitive closure of the “lifted” relation $\rightarrow_{\mathcal{V}}$ on clusters, and then checking that the two relations are the same between inputs and outputs. Transitive closure can be computed using a simple algorithm such as the all-pair shortest-path algorithm (Floyd 1962) which runs in $O(|V|^3)$ time, or other techniques with better worst-case complexity, e.g., see (Fischer and Meyer 1971) and (Coppersmith and Winograd 1990). The acyclicity condition can be verified by explicitly computing the quotient graph and then checking for acyclicity using Tarjan’s algorithm (Tarjan 1972) which runs in linear time. \square

The *optimal disjoint clustering problem* is, given an SDG G , find a valid clustering of G with minimum number of clusters. The decision version of this problem is, given G and an integer k , decide whether there exists a valid clustering of G with at most k clusters. We will show that the optimal disjoint clustering problem is NP-hard.

Before proving the result, we introduce some useful concepts. Notice that because of validity Condition 1, input and output nodes are clustered in isolation in a valid clustering. Therefore, a valid clustering can be defined by a partition of V_{int} instead of V . We will be using this alternative definition in what follows.

Definition 2 (Mergeability). Let $G = (V, \rightarrow)$ be an SDG and let $V' = \{v_1, \dots, v_k\} \subseteq V_{\text{int}} = \{v_1, \dots, v_n\}$ be a subset of internal nodes. We call V' mergeable if the partition $\{V', \{v_{k+1}\}, \dots, \{v_n\}\}$ defines an almost valid clustering. Two nodes $v_1, v_2 \in V_{\text{int}}$ are called mergeable, if the subset $\{v_1, v_2\}$ is mergeable. The mergeability graph of G , denoted $M(G)$, is an undirected graph on the node set V_{int} , where we add an edge between v_1 and v_2 iff v_1 and v_2 are mergeable.

Lemma 1. Let $G = (V, \rightarrow)$ be an SDG and let $\mathcal{V} = \{V_1, \dots, V_k\}$ be a partition of V_{int} defining an almost valid clustering. Let $\mathcal{V}' = \{V_1, \dots, V'_l, V''_l, \dots, V_k\}$ be a refinement of \mathcal{V} , where V_l is split in two non-empty, disjoint subsets $V'_l \cup V''_l = V_l$. Then \mathcal{V}' defines an almost valid clustering.

Proof. Suppose \mathcal{V}' is not almost valid. Then, \mathcal{V}' violates validity Condition 2 (Condition 1 holds by definition). This means there must exist nodes $v \in V_{\text{in}}$ and $v' \in V_{\text{out}}$ such that $[v] \rightarrow_{\mathcal{V}'}^* [v']$ in G/\mathcal{V}' , but $v \not\rightarrow^* v'$ in G . That is, there must exist a path

$$\sigma = [v] \rightarrow_{\mathcal{V}'} U_1 \rightarrow_{\mathcal{V}'} \dots \rightarrow_{\mathcal{V}'} U_l \rightarrow_{\mathcal{V}'} [v']$$

in G/\mathcal{V}' . This means in turn that there exist nodes $u_j, u'_j \in U_j$, for $j = 1, \dots, l$, such that

$$v \rightarrow u_1, u'_1 \rightarrow u_2, \dots, u'_l \rightarrow v'$$

We distinguish the following cases:

1) None of V'_l, V''_l appear in σ , that is, $\forall j = 1, \dots, l : U_j \neq V'_l \wedge U_j \neq V''_l$. Then σ is also a path of G/\mathcal{V} , which implies that \mathcal{V} is not almost valid, which is a contradiction.

2) Only one of V'_l, V''_l appear in σ . Suppose, w.l.o.g., that only V'_l appears in σ . Then, by replacing V'_l by V_l we obtain a new path σ' which is a path of G/\mathcal{V} : this holds because $V'_l \subset V_l$. This again implies that \mathcal{V} is not almost valid, a contradiction.

3) Both V'_l, V''_l appear in σ , that is, σ has the form:

$$[v] \rightarrow_{\mathcal{V}'} U_1 \rightarrow_{\mathcal{V}'} \dots \rightarrow_{\mathcal{V}'} U_{l_1} \rightarrow_{\mathcal{V}'} V'_l \rightarrow_{\mathcal{V}'} \dots$$

$\rightarrow_{\mathcal{V}'} V_i'' \rightarrow_{\mathcal{V}'} U_{i_2} \rightarrow_{\mathcal{V}'} \dots \rightarrow_{\mathcal{V}'} U_i \rightarrow_{\mathcal{V}'} [v']$.

Then the path

$\sigma' = [v] \rightarrow_{\mathcal{V}} U_1 \rightarrow_{\mathcal{V}} \dots \rightarrow_{\mathcal{V}} U_{i_1} \rightarrow_{\mathcal{V}} V_i \rightarrow_{\mathcal{V}} U_{i_2} \rightarrow_{\mathcal{V}} \dots \rightarrow_{\mathcal{V}} [v']$

is a path of G/\mathcal{V} : this holds because both V_i', V_i'' are subsets of V_i . This again implies that \mathcal{V} is not almost valid, a contradiction. \square

Lemma 2. *Let $G = (V, \rightarrow)$ be an SDG and let $\mathcal{V} = \{V_1, \dots, V_k\}$ be a partition of V_{int} defining an almost valid clustering. Then for all $i = 1, \dots, k$, for all $v, u \in V_i$, v and u are mergeable.*

Proof. By Lemma 1, since \mathcal{V} is almost valid, any refinement of \mathcal{V} into a finer partition is also almost valid. In particular the refinement

$$(\{v_1\}, \dots, \{v, u\}, \dots, \{v_n\})$$

that groups only nodes v, u together and leaves all others isolated must be almost valid. Therefore, by Definition 2, v and u are mergeable. \square

An SDG $G = (V, \rightarrow)$ is called *flat* if there are no dependencies between its internal nodes, that is

$$\nexists v, v' \in V_{\text{int}} : v \rightarrow v' \quad (2)$$

Notice that in a flat SDG G , any clustering of G that satisfies the first two validity conditions also satisfies the acyclicity condition, that is, it is a valid clustering. This is because the only edges allowed in a flat SDG are from input nodes to internal nodes and from internal nodes to output nodes.

Lemma 3. *Let $G = (V, \rightarrow)$ be a flat SDG and let $\mathcal{V} = \{V_1, \dots, V_k\}$ be a partition of V_{int} into mergeable cliques, that is, for all $i = 1, \dots, k$, for all $v, u \in V_i$, v and u are mergeable. Then \mathcal{V} defines a valid clustering.*

Proof. Suppose \mathcal{V} is not valid. Then, validity Condition 2 must be violated (Condition 1 holds by definition and Condition 3 cannot be violated in a flat graph). This means there must exist nodes $v \in V_{\text{in}}$ and $v' \in V_{\text{out}}$ such that $[v] \rightarrow_{\mathcal{V}}^* [v']$ (in G/\mathcal{V}) but $v \not\rightarrow^* v'$ (in G). Since v is an input, $[v] \rightarrow_{\mathcal{V}}^* [v']$ implies there exists $v_1 \in V_i$, for some $i = 1, \dots, k$, such that: (1) $v \rightarrow v_1$ and (2) $V_i \rightarrow_{\mathcal{V}}^* [v']$. (2) and the fact that G is flat together imply that (3) there exists $v_2 \in V_j$ such that $v_2 \rightarrow v'$. (1) and $v \not\rightarrow^* v'$ together imply (4) $v_2 \neq v_1$. (1), (4) and $v \not\rightarrow^* v'$ together imply that v_1 and v_2 are not mergeable, which contradicts the hypothesis that V_i is a mergeable clique. \square

We are now ready to prove our main theoretical result. We show that the optimal disjoint clustering problem is NP-hard by reducing the *partition-into-cliques* problem to the former problem. Let us first recall the problem of partitioning a graph into cliques. We are given an undirected graph $G = (V, E)$, where V is the set of nodes and E is the set of edges, and a positive integer $K \leq |V|$. We want to decide whether V can be partitioned into $k \leq K$ disjoint sets, $V = V_1 \cup \dots \cup V_k$, such that each V_i is a clique, that is, for any $v, u \in V_i$, $(v, u) \in E$. This problem is NP-hard (Garey and Johnson 1978).

Proposition 2. *Optimal disjoint clustering is NP-hard.*

Proof. Let $G = (V, E)$ be an undirected graph. We construct a flat SDG $G_{\mathcal{F}}$, using the construction technique illustrated in Figure 7. For each node v of G we create three nodes in $G_{\mathcal{F}}$: an internal node $v \in V_{\text{int}}$, an input node $v^i \in V_{\text{in}}$ and an output node $v^o \in V_{\text{out}}$. We add in $G_{\mathcal{F}}$ the edges $v^i \rightarrow v$ and $v \rightarrow v^o$.

For each edge $(u, v) \in E$, we create the following six nodes in $G_{\mathcal{F}}$: $e_u, e_v \in V_{\text{int}}$, $e'_u, e'_v \in V_{\text{in}}$, $e''_u, e''_v \in V_{\text{out}}$. We also insert in $G_{\mathcal{F}}$ the edges $e'_u \rightarrow e_u$, $e_u \rightarrow e''_u$, $e'_v \rightarrow e_v$, $e_v \rightarrow e''_v$. Additionally, we create the following four edges: $u^i \rightarrow e_u$, $e_u \rightarrow$

v^o , $v^i \rightarrow e_v$, $e_v \rightarrow v^o$. $G_{\mathcal{F}}$ can be constructed in linear time in the size of G .

Let us study $M(G_{\mathcal{F}})$. First, note that a node e_u produced by edge e of G is not mergeable with any other internal node. For instance, e_u is not mergeable with u , since this would create the dependency $e'_u \rightarrow^* u^o$; e_u is not mergeable with e_v , since this would create the dependency $e'_u \rightarrow^* e''_v$; e_u is not mergeable with v , since this would create the dependency $v^i \rightarrow^* e''_u$. Now, let us focus on two nodes u and v of G . Suppose, first, that $(u, v) \notin E$. Then u and v are not mergeable in $G_{\mathcal{F}}$, since merging them would create the dependency $u^i \rightarrow^* v^o$ (or $v^i \rightarrow^* u^o$). Next, suppose that $(u, v) \in E$. Then u and v are mergeable in $G_{\mathcal{F}}$. Indeed, in this case we have the picture shown in Figure 7. Merging u and v does not add new dependencies, since u^o already depends on v^i and v^o already depends on u^i .

It follows that two nodes u and v of G are mergeable in $G_{\mathcal{F}}$ iff they are adjacent in G . This, and the fact that e_u -type nodes are not mergeable with any other node, implies that $M(G_{\mathcal{F}})$ is the disjoint union of G and a set of $2|E|$ independent nodes e_u, e_v, \dots , that must be clustered in isolation. We claim that G can be partitioned into k cliques iff $G_{\mathcal{F}}$ admits a valid clustering of $k + 2|E|$ clusters (we are not counting the clusters for the input and output nodes). Suppose $G_{\mathcal{F}}$ can be clustered into $k + 2|E|$ clusters. Then the nodes of G (that are internal nodes of $G_{\mathcal{F}}$) must be clustered into k clusters V_1, \dots, V_k , since the rest $2|E|$ clusters are reserved for the e_u -type nodes. By Lemma 2, each V_i can only contain nodes that are pairwise mergeable. Therefore, each V_i corresponds to a clique of $M(G_{\mathcal{F}})$, thus to a clique of G . Conversely, suppose that G can be partitioned into k cliques. Thus, $M(G_{\mathcal{F}})$ can be partitioned into $k + 2|E|$ (mergeable) cliques. Since $G_{\mathcal{F}}$ is flat, by Lemma 3, this partition defines a valid clustering for $G_{\mathcal{F}}$. \square

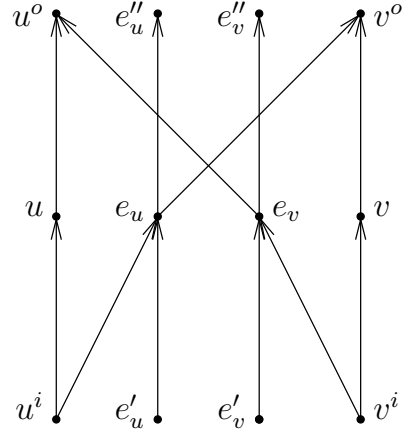


Figure 7. The SDG sub-graph representing edge (u, v) of G in the proof of Proposition 2.

Corollary 1. *Optimal disjoint clustering is NP-complete.*

Proof. The problem is NP-hard as shown in Proposition 2. It is also in NP: non-deterministically choose a clustering, then check whether it is valid. The latter can be done in polynomial time as shown in Proposition 1. \square

7. Reduction to SAT

In this section, we show how the optimal disjoint clustering problem can be solved by reducing it to a propositional satisfiability problem. The latter can be solved in practice today using a SAT

solver. Before we present our method, we need some technical results.

Let $G = (V, \rightarrow)$ be an SDG with sets of input and output nodes V_{in} and V_{out} , respectively. For a subset $V' \subseteq V$ of nodes, we define

$$\text{In}(V') = \{v \in V_{\text{in}} \mid \exists v' \in V' : v \rightarrow^* v'\} \quad (3)$$

$$\text{Out}(V') = \{v \in V_{\text{out}} \mid \exists v' \in V' : v' \rightarrow^* v\} \quad (4)$$

the *set of dependent inputs and outputs of V'* , respectively. We write $\text{In}(v)$ and $\text{Out}(v)$ instead of $\text{In}(\{v\})$ and $\text{Out}(\{v\})$, respectively.

Let \mathcal{V} be a partition of V_{int} . For any $V' \in \mathcal{V}$, we define

$$\text{In}_{\mathcal{V}}(V') = \{v \in V_{\text{in}} \mid [v] \rightarrow_{\mathcal{V}}^* V'\} \quad (5)$$

$$\text{Out}_{\mathcal{V}}(V') = \{v \in V_{\text{out}} \mid V' \rightarrow_{\mathcal{V}}^* [v]\} \quad (6)$$

If $v \in \text{In}_{\mathcal{V}}(V')$ then we say that cluster V' depends on input node v . If $v \in \text{Out}_{\mathcal{V}}(V')$ then we say that output node v depends on cluster V' . Notice that, by definition, $\text{In}(V') \subseteq \text{In}_{\mathcal{V}}(V')$ and $\text{Out}(V') \subseteq \text{Out}_{\mathcal{V}}(V')$. The inverse inclusions do not generally hold, however. Also note that, by definition, if $V' \rightarrow_{\mathcal{V}}^* V''$ then $\text{In}_{\mathcal{V}}(V') \subseteq \text{In}_{\mathcal{V}}(V'')$ and $\text{Out}_{\mathcal{V}}(V') \supseteq \text{Out}_{\mathcal{V}}(V'')$. Finally, note that for $v \in V_{\text{in}}$ and $u \in V_{\text{out}}$, $[v] \rightarrow_{\mathcal{V}}^* [u]$ iff there exists $V' \in \mathcal{V}$ such that $v \in \text{In}_{\mathcal{V}}(V')$ and $u \in \text{Out}_{\mathcal{V}}(V')$.

Lemma 4. *Let $G = (V, \rightarrow)$ be an SDG and let $\mathcal{V} = \{V_1, V_2, \dots, V_k\}$ be a partition of V_{int} defining an almost valid clustering. Suppose that $\text{In}_{\mathcal{V}}(V_1) = \text{In}_{\mathcal{V}}(V_2)$ and $\text{Out}_{\mathcal{V}}(V_1) = \text{Out}_{\mathcal{V}}(V_2)$ and let $\mathcal{V}' = \{V_1 \cup V_2, V_3, \dots, V_k\}$. \mathcal{V}' defines an almost valid clustering.*

Proof. \mathcal{V}' is a partition of internal nodes only, thus the clustering it defines satisfies validity Condition 1 by definition. Suppose there exist $v \in V_{\text{in}}$ and $v' \in V_{\text{out}}$, such that $[v] \rightarrow_{\mathcal{V}'}^* [v']$ but $v \not\rightarrow^* v'$. Then, there must exist $V', V'' \in \mathcal{V}'$, such that: (1) $[v] \rightarrow_{\mathcal{V}'}^* V'$, (2) $V' \rightarrow_{\mathcal{V}'}^* V''$, and (3) $V'' \rightarrow_{\mathcal{V}'}^* [v']$. If both $V' \neq V_1 \cup V_2$ and $V'' \neq V_1 \cup V_2$ then the clustering induced by \mathcal{V} does not satisfy Condition 2, which contradicts the hypothesis that it is almost valid. Thus, $V' = V_1 \cup V_2$ or $V'' = V_1 \cup V_2$.

Case (a): $V' = V_1 \cup V_2$. (2) implies that either $V_1 \rightarrow_{\mathcal{V}'}^* V''$ or $V_2 \rightarrow_{\mathcal{V}'}^* V''$. Without loss of generality, we consider the latter case. (1) and the fact that $\text{In}_{\mathcal{V}}(V_1) = \text{In}_{\mathcal{V}}(V_2)$ imply $[v] \rightarrow_{\mathcal{V}} V_2$. (3) implies $V'' \rightarrow_{\mathcal{V}} [v']$. Thus, we have $[v] \rightarrow_{\mathcal{V}} V_2 \rightarrow_{\mathcal{V}}^* V'' \rightarrow_{\mathcal{V}} [v']$, i.e., $[v] \rightarrow_{\mathcal{V}}^* [v']$, which contradicts the hypothesis that \mathcal{V} defines an almost valid clustering.

Case (b): $V'' = V_1 \cup V_2$. (2) implies that either $V' \rightarrow_{\mathcal{V}'}^* V_1$ or $V' \rightarrow_{\mathcal{V}'}^* V_2$. Without loss of generality, we consider the latter case. (3) and the fact that $\text{Out}_{\mathcal{V}}(V_1) = \text{Out}_{\mathcal{V}}(V_2)$ imply $V_2 \rightarrow_{\mathcal{V}} [v']$. (1) implies $[v] \rightarrow_{\mathcal{V}} V'$. Thus, we have $[v] \rightarrow_{\mathcal{V}} V' \rightarrow_{\mathcal{V}}^* V_2 \rightarrow_{\mathcal{V}} [v']$, i.e., $[v] \rightarrow_{\mathcal{V}}^* [v']$, which again contradicts the hypothesis that \mathcal{V} defines an almost valid clustering. \square

Lemma 5. *Let $G = (V, \rightarrow)$ be an SDG and let \mathcal{V} be a partition of V_{int} defining an almost valid clustering. Suppose \mathcal{V} is optimal w.r.t. the number of clusters, that is, there is no partition \mathcal{V}' that has fewer elements than \mathcal{V} and also defines an almost valid clustering. Then \mathcal{V} defines a valid clustering.*

Proof. \mathcal{V} defines an almost valid clustering, therefore, validity Conditions 1 and 2 are already satisfied (Definition 1). We need to prove that Condition 3 (acyclicity) is also satisfied. Suppose not. Then there exist $V_1, V_2 \in \mathcal{V}$, $i \neq j$, such that $V_1 \rightarrow_{\mathcal{V}}^* V_2$ and $V_2 \rightarrow_{\mathcal{V}}^* V_1$. This implies that $\text{In}_{\mathcal{V}}(V_1) = \text{In}_{\mathcal{V}}(V_2)$ and $\text{Out}_{\mathcal{V}}(V_1) = \text{Out}_{\mathcal{V}}(V_2)$. But this means that we can merge V_1 and V_2 and still obtain an almost valid clustering, with fewer elements (Lemma 4). This contradicts the hypothesis that \mathcal{V} is optimal. \square

We now propose the reduction to SAT. Let $G = (V, \rightarrow)$ be a Scheduling Dependency Graph as before. Let $k \in \mathbb{N}$ be an arbitrary

$$\begin{aligned} \mathcal{F}_k \equiv & \bigwedge_{1 \leq j \leq k} \bigvee_{b \in V_{\text{int}}} X_{bj} \\ & \wedge \bigwedge_{\substack{b \in V_{\text{int}} \\ 1 \leq j \leq k}} \left(X_{bj} \Leftrightarrow \bigwedge_{\substack{1 \leq \ell \leq k \\ \ell \neq j}} \neg X_{b\ell} \right) \\ & \wedge \bigwedge_{\substack{b \in V_{\text{int}}, o \in V_{\text{out}} \\ b \rightarrow o \\ 1 \leq j \leq k}} (X_{bj} \Rightarrow Y_{oj}) \\ & \wedge \bigwedge_{\substack{b \in V_{\text{int}}, i \in V_{\text{in}} \\ i \rightarrow b \\ 1 \leq j \leq k}} (X_{bj} \Rightarrow Z_{ij}) \\ & \wedge \bigwedge_{\substack{b_1, b_2 \in V_{\text{int}}, o \in V_{\text{out}} \\ b_1 \rightarrow b_2 \\ 1 \leq j \neq \ell \leq k}} (X_{b_1 j} \wedge X_{b_2} \Rightarrow (Y_o \Rightarrow Y_{oj})) \\ & \wedge \bigwedge_{\substack{b_1, b_2 \in V_{\text{int}}, i \in V_{\text{in}} \\ b_1 \rightarrow b_2 \\ 1 \leq j \neq \ell \leq k}} (X_{b_1 j} \wedge X_{b_2} \Rightarrow (Z_{ij} \Rightarrow Z_{i\ell})) \\ & \wedge \bigwedge_{\substack{i \in V_{\text{int}}, o \in V_{\text{out}} \\ i \not\rightarrow^* o \\ 1 \leq j \leq k}} \neg(Y_{oj} \wedge Z_{ij}) \end{aligned}$$

Figure 8. Encoding almost valid clusterings as propositional formulas.

positive integer. We define the propositional formula \mathcal{F}_k as shown in Figure 8, where X_{bj}, Y_{oj}, Z_{ij} are boolean variables denoting that internal node b belongs to cluster j , that output node o depends on cluster j , and that cluster j depends on input node i , respectively.

\mathcal{F}_k encodes the following conditions (conjunctions appearing from top to bottom in Figure 8):

1. every cluster must contain at least one internal node,
2. an internal node belongs to exactly only one cluster,
3. if an output o directly depends on an internal node b then o also depends on the cluster containing b ,
4. if an internal node b directly depends on an input node i then the cluster containing b also depends on input node i ,
5. if an internal node b_2 directly depends on another internal node b_1 , then the cluster containing b_2 will depend on every input node that the cluster containing b_1 depends upon (i.e., $b_1 \rightarrow b_2 \Rightarrow \text{In}_{\mathcal{V}}([b_1]) \subseteq \text{In}_{\mathcal{V}}([b_2])$),
6. if an internal node b_2 directly depends on another internal node b_1 then every output that depends on the cluster containing b_2 will also depend on the cluster containing b_1 (i.e., $b_1 \rightarrow b_2 \Rightarrow \text{Out}_{\mathcal{V}}([b_1]) \supseteq \text{Out}_{\mathcal{V}}([b_2])$), and,
7. if an output o does not depend on an input i then it is not the case that a cluster j depends on i and o depends on cluster j .

Lemma 6. *Let $G = (V, \rightarrow)$ be an SDG and $k \in \mathbb{N}$ such that \mathcal{F}_k is satisfiable. Let ϕ be a satisfying assignment and \mathcal{V} the clustering*

induced by $\mathcal{V}' = \{V_1, \dots, V_k\}$ where for $b \in V_{int}$, $b \in V_j$ if and only if X_{bj} is true in ϕ . Then:

1. $\forall 1 \leq j \leq k. \forall o \in V_{out}. V_j \rightarrow_{\mathcal{V}'}^* [o]$ implies that Y_{oj} is true in ϕ ,
2. $\forall 1 \leq j \leq k. \forall i \in V_{in}. [i] \rightarrow_{\mathcal{V}'}^* V_j$ implies that Z_{ij} is true in ϕ .

Proof. Follows by induction on the length of the path on the dependence relation $\rightarrow_{\mathcal{V}}$. \square

Lemma 7. Let $G = (V, \rightarrow)$ be an SDG and $k \in \mathbb{N}$. \mathcal{F}_k is satisfiable if and only if there exists an almost valid clustering defined by \mathcal{V}' , a partition of V_{int} , such that $|\mathcal{V}'| = k$.

Proof. (\Leftarrow) Let \mathcal{V}' be a partition defining an almost valid clustering \mathcal{V} such that $|\mathcal{V}'| = k$ and let $b \in V$ and $\forall 1 \leq j \leq k. V_j \in \mathcal{V}'$. Set each boolean variable X_{bj} to true if $b \in V_j$ and to false otherwise. Note that the first two conjunction terms in \mathcal{F}_k hold for any proper partition of size k . Set each variable Y_{oj} to true if $o \in \text{Out}_{\mathcal{V}}(V_j)$ and to false otherwise. Set each variable Z_{ij} to true if $i \in \text{In}_{\mathcal{V}}(V_j)$ and to false otherwise. Observe now that the third to sixth conjunction terms in \mathcal{F}_k encode the following facts: $\text{In}(V_j) \subseteq \text{In}_{\mathcal{V}}(V_j)$, $\text{Out}(V_j) \subseteq \text{Out}_{\mathcal{V}}(V_j)$, and $\forall j, \ell$, if $V_j \rightarrow_{\mathcal{V}'}^* V_\ell$ then $\text{In}_{\mathcal{V}}(V_j) \subseteq \text{In}_{\mathcal{V}}(V_\ell)$ and $\text{Out}_{\mathcal{V}}(V_j) \supseteq \text{Out}_{\mathcal{V}}(V_\ell)$. Finally, the last term encodes the fact that $\forall i \in V_{in}, o \in V_{out}, i \not\rightarrow^* o \Rightarrow \exists j : i \in \text{In}_{\mathcal{V}}(V_j) \wedge o \in \text{Out}_{\mathcal{V}}(V_j)$. Thus the above is a satisfying assignment for \mathcal{F}_k .

(\Rightarrow) Consider a satisfying assignment ϕ to \mathcal{F}_k and let $\mathcal{V}' = \{V_1, \dots, V_k\}$ where $b \in V_j$ if and only if X_{bj} is true in ϕ . The first two terms in \mathcal{F}_k impose \mathcal{V}' to be a partition of size k . Let $\mathcal{V} = \mathcal{V}' \cup V_{int} \cup V_{out}$ as before and note that \mathcal{V} satisfies validity Condition 1. Suppose that validity Condition 2 does not hold. Hence $\exists i \in V_{in}, o \in V_{out}$ such that $i \not\rightarrow^* o$ and $[i] \rightarrow_{\mathcal{V}'}^* [o]$. Moreover there exists j such that $[i] \rightarrow_{\mathcal{V}'}^* V_j$ and $V_j \rightarrow_{\mathcal{V}'}^* [o]$. By Lemma 6 we have that Y_{oj} and Z_{ij} are true in ϕ but then ϕ would not be a satisfying assignment of \mathcal{F}_k since this assignment falsifies the last term in the conjunction. \square

Thus, we can solve the optimal disjoint clustering problem using the following procedure:

1. initialize $k := 1$;
2. build formula \mathcal{F}_k ;
3. check whether \mathcal{F}_k is satisfiable;
4. if \mathcal{F}_k is satisfiable, terminate and announce that the optimal disjoint clustering has k clusters; the clustering can be directly obtained by the X_{bj} variables: node b is in cluster j iff in the solution of \mathcal{F}_k , X_{bj} is true;
5. otherwise increment $k := k + 1$ and go to step 2.

The procedure is guaranteed to terminate since there always exists a valid clustering with at most as many nodes as the nodes of the SDG. By Lemma 7, the procedure is guaranteed to produce an almost valid clustering, which is also of minimal size. By Lemma 5, this almost valid clustering is also valid.

8. Tool and experiments

We have implemented the code generation techniques proposed in (Lublinerman and Tripakis 2008b) and in this paper in a prototype tool written in Java. The tool takes as input a Simulink model (.mdl file) as well as profiles of blocks (in our own ASCII format). The tool generates as output profiles for the macro blocks in the Simulink model as well as Java code that implements the interface functions for each of the macro blocks. The tool also generates various statistics such as those that are presented below.

Three clustering and corresponding code generation methods are currently implemented in the tool: the step-get and the dynamic methods proposed in (Lublinerman and Tripakis 2008b) and the optimal disjoint clustering method proposed in this paper. To solve the latter problem, the tool employs the SAT-based method described in Section 7. In our implementation we used the *SAT4J* SAT solver (see <http://www.sat4j.org>).

We have experimented using our tool on a set of Simulink examples. Some of these examples are described in The Mathworks' web-site for Simulink under "Demos" and they are available with the Simulink tool. Two other examples are real-world models provided by industrial partners from the automotive domain. For confidentiality reasons we cannot reveal the source neither the nature of these examples. We will refer to them as "X1" and "X2" in the rest of this section.

Our experiments had multiple objectives. First, we wanted to validate the need for multiple interface functions per block, which is the fundamental hypothesis justifying this and our previous work. Indeed, we found that this need arises in practice and it is not just a theoretical problem. In most of the examples we tried, there are feedback loops between macro blocks, at different levels of the hierarchy. One such example is an engine timing model coming from the Simulink demo suite. The top-level diagram for this example is shown in Figure 9 (picture copyright The Mathworks).⁵ The top-level diagram contains a number of macro blocks: "Throttle & Manifold", "Compression" and so on. Notice that there are multiple feedback loops between these blocks. However, when the hierarchy is flattened these loops are "broken" by blocks such as unit-delays. A monolithic code generation method that generates a single interface function per block would fail on all these examples, because of these feedback loops. Thus, multiple interface functions per block are needed in practice as well as in theory.

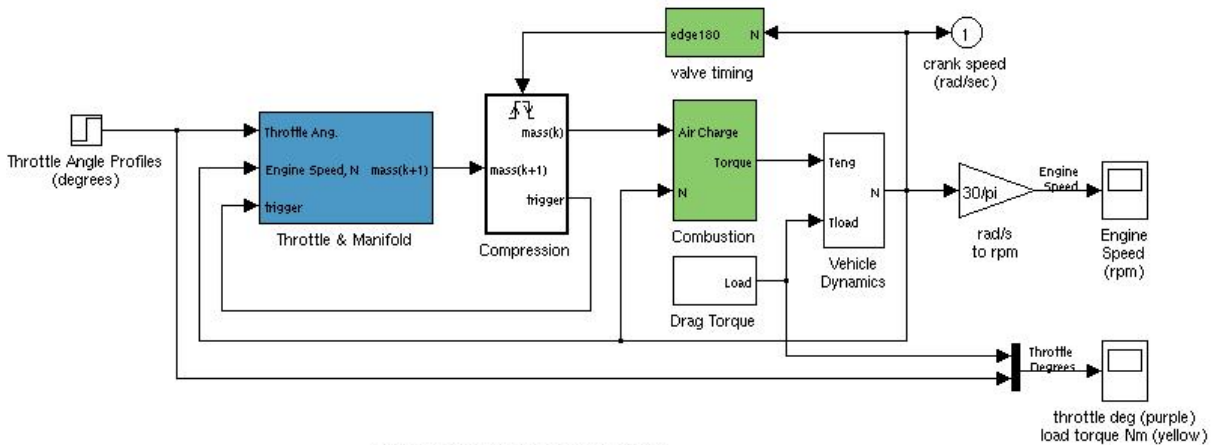
As a second objective for the experiments, we wanted to compare the three clustering and code generation methods implemented in the tool. Finally, we wanted to see how the SAT-based method performs in practice in terms of execution time.

The results of our experiments are summarized in Table 1. Although the optimal disjoint clustering problem and our SAT-based method have exponential worst-case complexity, in every example we have tried, execution took only a few seconds on a standard laptop computer (IBM T43p with a 2.26 GHz Intel Pentium processor and 2 GB of RAM running Windows XP and Java 6). Notice that this is the time it takes to process and generate code for *all* blocks of each example. The greatest SAT problem submitted to the SAT solver among all the examples that we tried was a formula with 851 boolean variables and 773 clauses. This formula was produced for the top-level block of example "X2" when checking whether a disjoint clustering with 4 clusters exists. Notice that modern SAT solvers can easily handle problems of much greater size.

Table 1 should be read as follows. Every row contains the information for one example. In the columns, we list: the name of the example; the total number of blocks in the example (including atomic and macro blocks); the number of macro blocks; the number of blocks that are of type Combinational (C), Non-Moore-Sequential (NS), and Moore-Sequential (MS); the maximum number of output ports that some block from the entire model has; the maximum number of sub-blocks that some (macro) block has; and the results we obtained by running the three code generation methods on these examples.

The three methods are: step-get (S-G), dynamic (Dyn) and optimal disjoint clustering (ODC). For each method, and each example, we report the total number of interface functions generated for

⁵ This model corresponds to the "Engine 1" row in Table 1.



Copyright 1990-2005 The MathWorks Inc.

Figure 9. An engine timing model in Simulink

model name	no. blocks			max no. outputs	max no. sub-blocks	total no. intf. func.			total code size (ELOC)			
	total	macro	C,NS,MS			S-G	Dyn	ODC	S-G	Dyn	ODC	max red.
ABS	27	3	1,0,2	1	13	4	4	4	57	57	57	—
Autotrans	42	9	4,0,5	2	11	fails	13	14	fails	108	101	14:6
Climate	65	10	4,0,6	4	29	12	14	14	144	165	144	42:26
Engine1	55	11	2,1,8	2	12	18	18	18	132	140	132	19:11
Engine2	73	13	3,2,8	2	13	20	20	20	180	188	180	19:11
Power window	75	14	6,2,6	3	11	20	21	21	180	199	183	32:16
X1	82	16	2,5,9	3	14	19	19	19	182	182	182	—
X2	112	16	7,9,0	5	14	22	24	24	245	342	261	108:27

Table 1. Experimental results

all the macro blocks in the example, as well as the total code size generated for these functions, measured in *effective* lines of Java code (ELOC). Effective means that we do not count comments, Java class constructors, and other similar overhead lines of code (these can be different when using a different language). We also do not count `init()` functions and the corresponding lines of code. The last column labeled “max red.” corresponds to the maximum reduction achieved by ODC and is explained later in the section.

By studying Table 1, we can make the following observations:

First, it is interesting to see that the step-get method succeeds in eliminating the feedback loops and corresponding dependency cycles between macro blocks in almost all cases: the exception is the “Autotrans” example, modeling an automotive transmission system. In this example the step-get method fails, that is, results in a (false) dependency cycle in the SDG of one of the macro blocks. As already mentioned, the step-get method generates at most two interface functions per block. More precisely, it generates two interface functions for Moore-sequential blocks (that have inputs) and a single interface function for all other blocks (Lublinerman and Tripakakis 2008b). The explanation why step-get succeeds on most of the examples lies in the fact that most of the cycles in the examples are “broken” by Moore-sequential blocks. For these blocks, two interface functions is all that is needed to achieve maximal reusability, and the step-get method provides exactly this. In conclusion, the examples show both the interest but also the limitations of the

step-get method, which justifies the need for more elaborate methods, in particular Dyn and ODC.

The second observation is that ODC indeed achieves code size reduction in all cases except two (the “ABS” and “X1” examples). Moreover, in all but one case (the “Autotrans” example) this reduction comes at no expense in terms of modularity: that is, the number of interface functions remains the same. In the case of “Autotrans” only one extra interface function is generated. The reason why ODC can reduce code size without modularity penalty is because the overlapping that the dynamic method creates can sometimes be eliminated without introducing an additional interface function. This happens when sub-graph G_1 overlaps with sub-graph G_2 , and the set of inputs that G_1 depends upon is a subset of the set of inputs that G_2 depends upon. In this case the overlapping area can be removed from G_2 , and the dependency $G_1 \rightarrow G_2$ can be added. This is valid, since G_2 requires more inputs than G_1 , therefore whenever the inputs for G_2 are available, G_1 can be executed before G_2 .

The above phenomenon occurs in all the examples except “Autotrans”, where an additional sub-graph (and corresponding interface function) needs to be generated in order to preserve the input-output dependencies. This occurs for the “Transmission Ratio” macro block, the internal diagram of which is shown in Figure 10. It is interesting to observe that the structure of this diagram is identical to the structure of diagram shown in Figure 4(a). Indeed, blocks B and C of the latter figure correspond to the two “Product” blocks of “Transmission Ratio” and block A_n corresponds to the

“Look-Up Table” block. This indicates that the motivating example of Figure 4 is not just an academic example but actually occurs in practice.

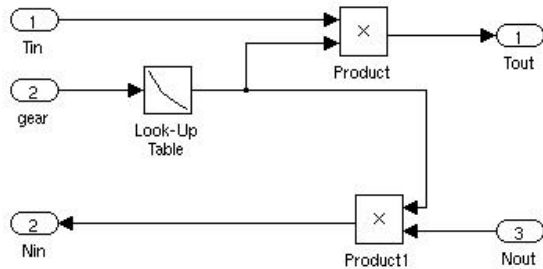


Figure 10. The internal diagram of the “Transmission Ratio” macro block

One may claim that the reduction that ODC achieves in code size is a small percentage of the overall size. This is true in most of the examples presented in Table 1, but not in all of them: in “X2” a reduction of almost 25% on the total code size is achieved. Moreover, in the context of modular code generation, it is meaningful to measure not only the reduction achieved in the entire set of blocks, but also in individual blocks. Indeed, these blocks are likely to be used not only in the current model, but in other models as well. The last column of Table 1, labeled “max red.” shows the greatest reduction achieved for some macro block in each example: the pair $L_1 : L_2$ is shown, where L_1 is the ELOC generated by Dyn for this block, and L_2 is the ELOC generated by ODC. It can be seen that the reduction in code size for individual blocks can be dramatic, e.g., 75% in “X2”, 50% in “Power window”, and so on. This justifies the practical interest in the ODC method.

Finally, it is worth pointing out that the examples presented in Table 1 involve blocks with a relatively small number of outputs each: from 1 to 5 at most. Because of this, and the way the dynamic method operates, the potential for overlap that can be removed is small on these examples. We expect that the savings that can be achieved by ODC are greater in blocks with a greater number of outputs, but experiments with larger examples are required to validate this expectation.

9. Conclusions

This work extends our previous work on modular code generation for synchronous block diagrams. In previous work we have studied the trade-off between modularity and reusability. In this paper we studied another trade-off, between modularity and code size. We can reduce code size by avoiding the code replication that one of our previous methods introduces. However, this generally comes with a price in terms of modularity. Optimizing modularity while avoiding replication and maintaining maximal reusability is a clustering problem in a directed acyclic graph. We showed that this problem is NP-complete, and that it can be solved by a simple enumeration of the number K of available clusters and calling a SAT solver to check satisfiability of a propositional formula for each K . We finally reported on a new tool that implements the methods developed in this and our previous work. Experimental results obtained by applying the tool on real Simulink models are encouraging and justify the interest in our approach. The experiments seem to suggest that the optimal number of clusters K is likely to be small in practice.

References

- P. Aubry, P. Le Guernic, and S. Machard. Synchronous distribution of Signal programs. In *29th Intl. Conf. Sys. Sciences*, pages 656–665. IEEE, 1996.
- A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proc. IEEE*, 91(1):64–83, January 2003.
- A. Benveniste, P. Le Guernic, and P. Aubry. Compositionality in dataflow synchronous languages: specification & code generation. Technical Report 3310, Irisa - Inria, 1997.
- G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
- D. Biernacki, J-L. Colaco, G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’08)*. ACM, 2008.
- P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Symp. POPL*, 1987.
- D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.*, 9:251–280, 1990.
- 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.
- M. Fischer and A. Meyer. Boolean matrix multiplication and transitive closure. *IEEE 12th Symp. on Switching and Automata Theory*, pages 129–131, 1971.
- R.W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.
- M.R. Garey and D.S. Johnson. *Computers and Intractability*. Freeman, 1978.
- 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.
- 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.
- R. Lubliner 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 2008a.
- R. Lubliner and S. Tripakis. Modularity vs. Reusability: Code Generation from Synchronous Block Diagrams. In *Design, Automation, and Test in Europe (DATE’08)*. ACM, March 2008b.
- O. Maffei 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.
- S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, 1994.
- 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.
- P. Raymond. Compilation séparée de programmes Lustre. Master’s thesis, IMAG, 1988. In French.
- 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.
- R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.