Modular Code Generation from Synchronous Block Diagrams: Interfaces, Abstraction, Compositionality

Stavros Tripakis[1](✉) and Roberto Lublinerman[2]

[1] Aalto University, Espoo, Finland
stavros.tripakis@gmail.com
[2] Google, Mountain View, USA
rluble@gmail.com

Abstract. We study abstract, compositional and executable representations of synchronous models in general and hierarchical synchronous block diagrams in particular. Our work is motivated by the problem of modular code generation, where sequential code (in, say, C or Java) must be generated for a given block independently of its context, that is, independently of the diagrams in which this block may be embedded.

We propose non-monolithic interfaces called *profiles* as a representation of blocks. A profile contains a set of interface functions that implement the semantics of the block, and a set of dependencies between these functions. Profiles are executable through the implementation of their interface functions. Profiles are compositional in the sense that a diagram of profiles can be represented as a single profile without loss of important information, such as input-output dependencies. This is contrary to traditional methods which use monolithic interfaces that contain a fixed number of interface functions, usually just one or two. Monolithic interfaces generally result in loss of input-output dependency information and are non-compositional. Profiles are abstract in the sense that they hide most of the internal details of a diagram (e.g., functionality).

We provide methods for profile synthesis and modular code generation: to automatically produce profiles and profile implementations of composite blocks, given profiles of their sub-blocks. Our work reveals fundamental trade-offs between the size and reusability of a profile, as well as between characteristics of the generated code and complexity of the synthesis algorithms. We discuss various algorithms that explore these trade-offs, among which algorithms that achieve maximal reusability with optimal profile size.

This paper unifies and extends the work presented in [30–32]. We gratefully acknowledge the contributions of our co-author Christian Szegedy. Roberto Lublinerman contributed to this work while he was at Cadence Design Systems; he is now at Google. We also thank the anonymous Reviewers of earlier versions of this work. This work was partially supported by the National Science Foundation (awards #1329759 and #1139138), and by the Academy of Finland. We gratefully acknowledge the support of Edward Lee, without whom this work would not have been possible.

M. Lohstroh et al. (Eds.): Lee Festschrift, LNCS 10760, pp. 449–477, 2018.
https://doi.org/10.1007/978-3-319-95246-8_26

1 Introduction

What is the parallel composition of two Mealy machines, or even two stateless functions? Consider, for instance, the block diagram shown to the left of Fig. 1. Blocks A and B represent two stateless functions (over some input and output domains). Block P is a *composite* block formed by encapsulating A and B: P represents the parallel composition of A and B. It is tempting to view P as a new stateless function whose input and output domains are the cartesian products of the input and output domains of A and B, respectively. This is problematic, however, as we then lose the information that the output of A does not depend on the input of B, and vice versa. Such information turns out to be critical when using P in certain contexts. For instance, if we connect P in a feedback configuration, as shown in the middle of Fig. 1, we obtain a diagram with a cyclic dependency: the input of P depends on its output. Although some methods exist to deal with such dependencies, they are expensive or even undecidable in general (see discussion below). Moreover, using such methods is sometimes an overkill. In our example, for instance, the situation is really simple: there is no real dependency cycle in the feedback configuration, as revealed by flattening P (right of Fig. 1).

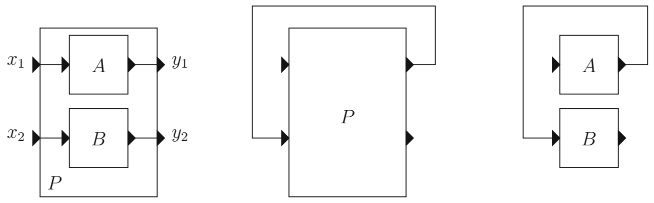


Fig. 1. A hierarchical block diagram (left), a possible way to connect macro block P (middle) and the same connection after flattening P (right).

The problem really lies in the fact that representing the parallel composition of functions A and B as a new function P loses the dependency information between inputs and outputs. In this paper we present a systematic method to represent, maintain and efficiently compute such information. Before further discussing our method and its benefits over alternatives, let us place our work in context.

This work is motivated by the need to develop reliable and efficient methods for the design and implementation of *embedded systems* [25]. Current practice can be qualified as being mostly about *low-level design*: build a prototype system, test it, discover problems, fix them and repeat the process. This is costly both in terms of money and time, and also offers few guarantees of producing reliable systems. So-called *model-based design* (MBD) has been proposed as an alternative. The MBD paradigm is based on the premise of using models for *high-level* design. Models can be analyzed in more exhaustive and less costly ways than prototype systems. MBD relies on powerful implementation techniques to derive executable systems from models. These techniques need to be as automatic as possible, in order to produce implementations efficiently. They also need to preserve as many of the properties of the high-level model as possible. This allows to produce implementations that are, as much as possible, *correct by construction*, which reduces the effort of testing at the implementation level.

In the field of embedded systems, like in many other fields, specialized (sometimes called "domain-specific") languages are used. These languages include features such as *concurrency*, *time* and *system dynamics*, which are integral parts of embedded system design. In this paper, we are particularly interested in *synchronous models*, whose execution proceeds by an infinite sequence of synchronous *rounds*. The synchronous model of computation (MoC) is a fundamental one, especially relevant in the context of embedded systems, since it is prevalent in many application domains, from control software to synchronous hardware.

Examples of synchronous models coming from the academia are the so-called *synchronous languages* [3], such as Lustre [14], Esterel [6,34] or Signal [20,27], or the *synchronous-reactive* domain of Ptolemy [19]. Simulink from The MathWorks[1] and SCADE from Esterel Technologies[2] are two commercial products, especially widespread in the automotive and avionics domains. SCADE has its foundations on Lustre and uses a purely synchronous MoC. Simulink contains both a continuous-time and a discrete-time part: the latter follows essentially the synchronous MoC.

The tools associated with languages such as the above include graphical model editors, simulators and code generators.[3] 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, uses of the latter type are increasingly being adopted by the industry. Thus, these tools can be seen as programming languages, debuggers and compilers for the embedded system domain.

In this paper, we use *synchronous block diagrams* (SBDs) [19,32] as a formal model that captures the synchronous MoC. A fundamental concept in our version of SBDs, directly inspired by Simulink, SCADE and Ptolemy, is *hierarchy*: a set of blocks can be connected to form a diagram, which may be encapsulated in a *composite*, or *macro*, block. The latter can be itself further connected and encapsulated. Hierarchies of arbitrary depth can be formed in this way. Hierarchy is essential in graphical formalisms since it allows to master complexity by building designs in a modular manner. Hierarchy facilitates the reuse of high-level components, both during model construction and code generation.

Our work has been motivated by the problem of *modular code generation* from synchronous models such as SBDs. We already explained the importance

[1] See http://www.mathworks.com/products/simulink/.

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

[3] Primarily software code generators, since software is becoming predominant in embedded systems, but also hardware code generators in some cases.

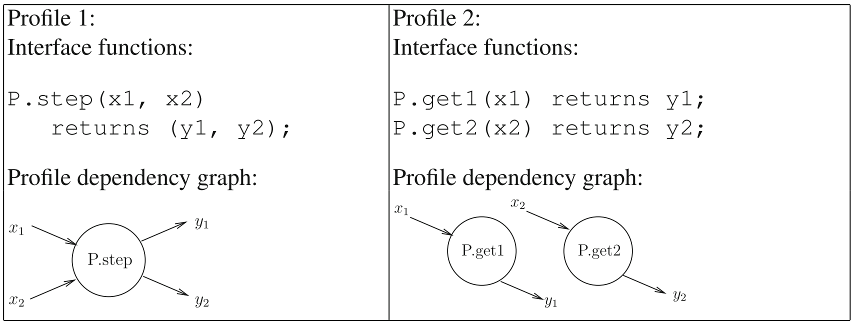
of code generation in the discussion above. Modular code generation consists in generating code from pieces of a model independently from other pieces. In the context of SBDs, modular code generation consists in generating code from a given block P independently from its context, that is, independently from the diagrams that P is or will be connected to. Just as separate compilation of different files of a large piece of software written in C++ or Java is essential, so is modular code generation from hierarchical models such as SBDs. It allows incremental compilation and scalability of the code generation process. It also allows building reusable model libraries. Finally, it allows to treat blocks as "black boxes" as much as possible. This is important in an industrial context, where intellectual property (IP) of models is a primary concern.

Most existing approaches to code generation from synchronous models are *monolithic*: they consist in generating, for a given block, a single `step` function that computes all block outputs given all its inputs. This is problematic because it loses input-output dependency information, as illustrated above. If the block has state, often two functions are generated, an `output` function to compute the outputs from the inputs and current state, and an `update` function to update the state, as in a Mealy machine. This does not solve the problem either, however, since inputs and outputs are still treated in a monolithic way in the `output` function.

One way to deal with this problem is to follow the approach proposed in [5,19,33] and used in Ptolemy [29]. This approach consists in generating two functions per block, an `output` and an `update` function as above (in Ptolemy these are called `fire` and `postfire`, respectively) but with the addition that these functions can operate over a special *unknown* value, corresponding to the bottom element of a complete partial order. At run-time, at every synchronous round, the `output` functions of all blocks are executed repeatedly until a fixpoint is computed. The fixpoint may contain unknown values, in which case the diagram is not well-defined and execution stops. Otherwise, execution proceeds to the next round where a new fixpoint is computed.

One problem with this approach is that it cannot guarantee *statically* (i.e., at compile-time) that no unknown values will be produced at run-time. Therefore, the approach is mostly suited for simulation, and cannot be used to produce code for safety-critical applications. One way to guarantee statically that the diagram is well-defined is to prove that the model is *constructive* in the sense of Berry [5]. Unfortunately, proving constructiveness is generally undecidable for models with infinite domains, and is expensive even for models with finite domains. Moreover, this approach requires semantic knowledge about each block, namely, what is the function that the block computes. Having such semantic knowledge is contrary to the goal of treating blocks as black boxes, that we pursue in this paper.

Our approach allows to make static guarantees. The key idea is to generate for a given block a *non-monolithic* interface, also called *profile*. The latter consists of a *not a-priori fixed* number of interface functions, plus a set of dependencies between these functions. Each function computes some outputs from some inputs. The dependencies capture the IO dependencies of the block. As an



Profile 1:
Interface functions:

```
P.step(x1, x2)
   returns (y1, y2);
```

Profile dependency graph:

Profile 2:
Interface functions:

```
P.get1(x1) returns y1;
P.get2(x2) returns y2;
```

Profile dependency graph:

Fig. 2. Two possible profiles for block P of Fig. 1: the left one is monolithic; the right one is not.

example, two possible profiles for block P of Fig. 1 are shown in Fig. 2. The leftmost profile is monolithic, and corresponds to the standard approach of treating P as simply a new function from both inputs to both outputs. The rightmost profile is non-monolithic, and corresponds to what one of our methods automatically generates.

The profile can be seen as an abstraction of the information contained in the block (i.e., in its internal hierarchy, which is not exposed in the profile). We present *profile synthesis* methods that allow to generate profiles automatically, and moreover, to explore different trade-offs during the generation of such profiles. In particular, trade-offs between the size of the generated profile and its accuracy. The smaller the size the better, for scalability and IP reasons. On the other hand, a profile that is too small may lose IO dependency information. This in turn results in a profile that is less *reusable*, that is, that cannot be used in some contexts. Apart from profile size vs. reusability trade-offs, we also study other trade-offs, such as between the quality of the resulting code and the complexity of computing the profile.

Contributions: This paper unifies and extends the work presented in [30–32]. This work provides a general and automatic solution to the problem of modular code generation from synchronous models, with static guarantees. Compared to the fixpoint-based approaches discussed above, ours can handle a smaller class of models, namely, those that exhibit no dependency cycles once the hierarchy is flattened (dependency cycles are allowed at higher levels, however, as in the example of Fig. 1). On the other hand, our approach can provide static (compile-time) guarantees, which cannot be generally provided by fixpoint-based approaches, as discussed above. With our method, diagrams (and the corresponding generated code) are guaranteed to have well-defined semantics (no unknown values) at compile-time. Moreover, interface functions are called at most once per round, in a statically determined order. Compared to fixpoint-based methods, where more than one iterations may be required to reach a fixpoint, static execution order has the benefits of smaller run-time overhead, better performance, and better predictability of the execution time of the generated code. All are crucial properties in an embedded system setting.

Organization: The rest of this paper is organized as follows. In Sect. 2 we discuss other related work. In Sect. 3 we explain the syntax and semantics of hierarchical SBDs. In Sect. 4 we present profiles. In Sect. 5 we describe our method for automatically synthesizing profiles and generating code that implements those profiles. Section 6 concludes this paper.

2 Other Related Work

Compositionality of synchronous models, and in particular the problem of cyclic dependencies, has been the topic of extensive study, and a variety of solutions have been proposed (e.g., see [3,15] for overviews). The most general is probably the one used in Esterel [5], however, it is often infeasible as discussed above. Simulink and Lustre compilers both rely of statically detecting cyclic dependencies and rejecting the model if one is found. In order to do this they flatten the model, however, which is not modular. SCADE does not flatten the model, but requires absence of cyclic dependencies at *every* level of the hierarchy, which is quite restrictive.

Equipping models with input-output dependency information has been proposed in [2] and also in [48]. These works use such information mainly for analysis (e.g., distinguish between true and false dependencies). Our goals are also synthesis and code generation. We also study trade-offs such as between profile size vs. reusability, which are not studied in these works.

Code generation for synchronous models and languages has been extensively studied, however, modular code generation has received less attention: in 2003, [3] stated that "a unified treatment [of this problem] remains a research topic". In fact, separate compilation (essentially the same problem) for synchronous languages has been identified as synonymous with monolithic compilation, and as such deemed to be generally infeasible [23,35]. Our non-monolithic framework provides the unified treatment that has been missing.

Although not identified explicitly as such, non-monolithic approaches have been described previously, for instance, in [4,21,24,38]. These works are, however, focusing on different problems, such as static scheduling and code distribution, and as such provide incomplete solutions to the modular code generation problem. In particular, they do not deal with hierarchies of arbitrary depth, they do not identify code generation trade-offs and they do not address the problems of optimizing metrics such as profile size or reusability.

[47] study partial evaluation in Esterel: generating code that computes outputs even in the presence of unknown inputs. Modular compilation for Quartz (a variant of Esterel) is studied in [10,39]. Their work focuses more on problems such as so-called *schizophrenia* which are specific to imperative synchronous languages like Esterel, and less on causality problems which is our main focus. Causality problems are also outside the focus of work on composable code generation from languages like Giotto where by definition all outputs are produced

with a unit delay [26]. The focus there is on compositionality of timing and scheduling, as is the case with work on compositional real-time scheduling [40].

Profiles are rich interfaces. Interfaces are a key mechanism for abstraction, modularity, compositionality, and many other important properties of software and systems. Interfaces have appeared in the literature in many different settings and communities, such as software engineering and programming languages (e.g., *Typestate* [41]), or formal methods (e.g., *interface automata* [1], *relational interfaces* [45], and *timed actor interfaces* [22]). Particularly close to our work here is the theory of relational interfaces which have synchronous semantics similar to SBDs [45]. This work has since been extended into a powerful compositional framework called *refinement calculus of reactive systems* (RCRS) [37]. RCRS includes methods and tools to translate hierarchical SBDs into a formal algebra of contracts which can be manipulated formally (e.g., using a theorem prover) and symbolically [18]. RCRS also includes a formal notion of *refinement* which allows to specify a system at different levels of abstraction, and also to speak formally about *substitutability* (when can a component replace another one) [45].

Interfaces are key for simulation environments like Ptolemy. Most modern simulators are built in a modular fashion, where the simulation engine is separated from the simulated models. This allows the same engine to be used for a large variety of models, and also allows the addition of new models, model components, model libraries, etc. To achieve this modularity in the implementation, a clear API (application program interface) is used. This API is typically implemented by the model components (e.g., "blocks" in Simulink, "actors" in Ptolemy) and called by the simulation engine (although call-backs are also sometimes used, e.g., the `fireAt` method in Ptolemy). A formalization of (part of) Ptolemy's actor interface is provided in [46], as part of an attempt to give formal semantics to the language.

Different simulators typically use different APIs, which hinders the sharing and exchange of models, if these models are written in different languages. The FMI standard [7,8] aims to remedy this by providing standard APIs for model exchange and co-simulation. The development of FMI has received great attention recently as it raises several interesting questions, such as what properties should a "good" co-simulation algorithm have [11,12,16], how to bridge the semantic gap between heterogeneous modeling formalisms and the standard API [9,43], how to integrate FMI in existing simulation tools [17], etc.

The ideas presented in this paper are not limited to models with synchronous semantics. They have indeed inspired us to explore modular code generation and compositionality in other contexts, such as dataflow [28]. Our study revealed that hierarchical SDF graphs used in tools such as Ptolemy are non-compositional in the sense that a composite SDF actor cannot always be replaced by an atomic one [42], a problem reminiscent of the limitations of monolithic interfaces in the case of SBDs. A compositional alternative inspired from the concept of non-monolithic interfaces is proposed in [28].

A broader discussion about the role of compositionality in the science of system design can be found in [44].

3 Synchronous Block Diagrams

3.1 Hierarchical Block Diagrams

We consider a notation based on a set of *blocks* that can be connected to form *diagrams* (see Fig. 3). Each block has a number of *input ports* (possibly zero) and a number of *output ports* (possibly zero). Diagrams are formed by connecting some output port of a block A to some input port of a block B (B can be the same as A). We assume that a port can only be connected to a single other port: *fan-out* can be explicitly modeled using blocks that replicate their input to their outputs. We also assume that every output port in a diagram is connected: again, this is without loss of generality, since outputs can be connected to "dummy" blocks that do nothing. Each port has a given *data type* (integer, boolean, ...) and connections can only be done among ports with compatible data types, as in a standard typed programming language. We will not worry about data types in this paper as these can be handled using standard type-theoretic methods.

Blocks are either *atomic* or *macro*. A macro (i.e., composite) block encapsulates a block diagram into a block. The blocks forming the diagram are called the *internal* blocks of the macro block, or (synonymously) its *sub-blocks*. In the example shown to the left of Fig. 3, block Q is a macro block and A, B, C are 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. Signals inherit the data types of their source ports.

In the context of a modular and hierarchical notation such as the block diagrams we consider in this paper, it is useful to distinguish between block *types* and block *instances*. Indeed, a block, whether atomic or composite, can be used in a given diagram multiple times. For example, a block of type *Adder*, that computes the arithmetic sum of its inputs, can be used multiple times in a given diagram. In this case, we say that the block of type *Adder* is *instantiated* multiple times. Each "copy" of the block is called an *instance*. In the rest of the paper, we omit to distinguish between type and instance when the distinction is clear from context.

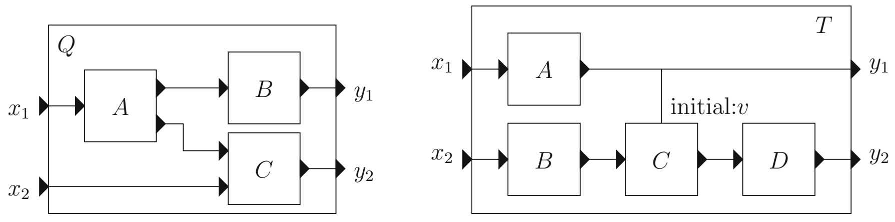


Fig. 3. Left: a hierarchical block diagram consisting of macro block Q with sub-blocks A, B, C. Right: a diagram with a triggered block C.

3.2 Triggers

In a diagram, any (atomic or macro) block may be *triggered* by a Boolean signal x: the intention is that the triggered block is to "fire" only when x is true. If x is false, then the outputs of the triggered block retain their value (i.e., the value that they had in the previous synchronous round). The signal x is called the *trigger* of the triggered block. A block can have at most one trigger. The diagram shown to the left of Fig. 3 has no triggers. An example of a diagram with triggers is shown to the right of Fig. 3: block A produces a (Boolean) signal that triggers block C.

When a block is triggered, the user specifies initial values for each output of that block. These determine the values of the outputs during the initial time interval (possibly empty) until the block is triggered for the first time. We call such a value a *trigger-initial value*. In the example shown to the right of Fig. 3, a trigger-initial value v is specified for the (single) output of triggered block C. Note that if a block has many outputs, a potentially different trigger-initial value can be specified for each output.[4]

[31] show that triggers do not add to the expressiveness of synchronous block diagrams and can be eliminated by a structural transformation, which essentially transforms triggers into inputs. This transformation is not modular, however, because it propagates in a top-down manner throughout the entire hierarchy, all the way to the atomic blocks. This contradicts our requirement that blocks be seen as "black boxes". To achieve modularity, we provide methods that handle triggers directly, without eliminating them.

Our motivation for studying triggers is to capture Simulink's *triggered subsystems*. Triggers are a simpler and more restricted concept than the concept of *clocks*, used in synchronous languages and more generally in synchronous dataflow [13]. Indeed, signals in a synchronous block diagram are always "present", that is, they have a well-defined value at every synchronous round. This includes signals that are outputs of triggered blocks. For this reason, a sophisticated type-checking mechanism such as a *clock calculus* [13] is not needed in our case.

3.3 Combinational, Sequential, and Moore Blocks

Blocks (more precisely, block types) can be either *combinational* (i.e., *stateless*) or *sequential* (i.e., *stateful*, that is, having internal state). Atomic blocks are pre-classified as either combinational or sequential. A macro block is combinational

[4] A Reviewer of an earlier version of this article correctly pointed out that there may be potential problems with the specification of trigger-initial values. In particular, complications may arise if downstream models are only valid for certain inputs: what happens if a trigger-initial value is not a legal input for the downstream model? While we agree that this is a problem, we feel that it is not confined to the use of triggers. The same problem might arise in a diagram without triggers. In general, the problem arises from *non-input-receptive* components. For a thorough study of such components, we refer the reader to [36,45].

iff all its sub-blocks are combinational; otherwise it is sequential. Some sequential blocks are *Moore* (from Moore machines). All outputs of a Moore block only depend on the current state of the block, but not on the inputs. See also Sect. 3.4.

3.4 Semantics

As we shall see in Sect. 4, each block in our framework is represented by a set of interface functions and a directed acyclic graph whose nodes are these functions. Let f be such an interface function with inputs $x_1, \ldots, x_n$ and outputs $y_1, \ldots, y_m$, where $m, n \in \mathbb{N}$ and $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$. Implicitly, f is also associated with a state variable s (possibly a vector). Denote by D_v the domain of variable v. Then, semantically f is a function

$$f : D_{x_1} \times \cdots \times D_{x_n} \times D_s \to D_{y_1} \times \cdots \times D_{y_m} \times D_s \tag{1}$$

Such a function f then defines the behavior of a SBD as a dynamical system in time. In particular, each signal x is interpreted semantically as a total function $x : \mathbb{N} \to D_x$, where $x(k)$ denotes the value of x at synchronous round k. Suppose, for the moment, that f belongs to a non-triggered block in the diagram (the case of triggered blocks is examined below). Then, if x is an input to f then $x(k)$ is determined by the environment (which can be another function in the diagram), otherwise it is determined by f as follows:

$$\big(y_1(k), \ldots, y_m(k), s(k+1)\big) = f\big(x_1(k), \ldots, x_n(k), s(k)\big) \tag{2}$$

That is, f takes as input the current values of all its input ports and the current value of the state, and produces as output the current values of all its output ports and the next value of the state.

For example, if f_+ is the (unique) interface function for an *Adder* block that has two inputs x_1, x_2, one output y, and no internal state, then semantically f_+ is defined by

$$f_+(v_{x_1}, v_{x_2}, v_s) = (v_{x_1} + v_{x_2}, v_s) \tag{3}$$

which defines the dynamical system

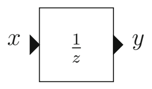
$$y(k) = x_1(k) + x_2(k) \tag{4}$$

As can be seen in this example, stateless blocks can be formalized as blocks with a single, "dummy" state v_s that never changes.

As another example, consider the *unit-delay* block, also denoted $\frac{1}{z}$. This is a stateful block with a single input port x and a single output port y. As we shall see in Sect. 4 the profile of $\frac{1}{z}$ contains two interface functions, one that computes the output from the current state and one that updates the state based on the input. Both are semantically the identity function, and define the dynamical system

$$\big(y(k), s(k+1)\big) = \big(s(k), x(k)\big) \tag{5}$$

The behavior of the unit-delay block is illustrated in Fig. 4. The value of the input signal x at round k is $x(k)$. The value of the state at round 0, i.e., $s(0)$, is denoted s_{init}.



x:	$x(0)$	$x(1)$	$x(2)$	$x(3)$	$x(4)$	$x(5)$	
s:	s_{init}	$x(0)$	$x(1)$	$x(2)$	$x(3)$	$x(4)$	$\cdots$
y:	s_{init}	$x(0)$	$x(1)$	$x(2)$	$x(3)$	$x(4)$	
	0	1	2	3	4	5	rounds

Fig. 4. A unit-delay block (left) and its semantics (right).

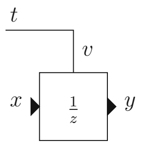
We now turn to the case of triggered blocks. Suppose f is an interface function of some block A which is triggered, in the diagram in question, by some signal t. Notice that the semantics of f remain the same, since f is defined independently from context. However, the semantics of the output signals of f change, because of the fact that A is triggered. In particular, let $y \in \{y_1, \ldots, y_m\}$ be an output of f. Then, Eq. (2) generalizes to

$$y(k) = \begin{cases} f_y(x_1(k), \ldots, x_n(k), s(k)), & \text{if } t(k) = \mathit{true} \\ y(k-1), & \text{if } t(k) = \mathit{false} \text{ and } k > 0 \\ v_y, & \text{if } t(k) = \mathit{false} \text{ and } k = 0 \end{cases} \tag{6}$$

$$s(k+1) = \begin{cases} f_s(x_1(k), \ldots, x_n(k), s(k)), & \text{if } t(k) = \mathit{true} \\ s(k), & \text{if } t(k) = \mathit{false} \end{cases} \tag{7}$$

where f_y, f_s are projections of f to variables y and s, respectively, and v_y is the trigger-initial value specified in the diagram for y.

An example that illustrates the semantics of triggered blocks is given in Fig. 5: t is the triggering signal, "T" and "F" denote true and false, respectively, and v is the trigger-initial value for y.



t:	F	T	F	F	T	T	
x:	$x(0)$	$x(1)$	$x(2)$	$x(3)$	$x(4)$	$x(5)$	
s:	s_{init}	s_{init}	$x(1)$	$x(1)$	$x(1)$	$x(4)$	$\cdots$
y:	v	s_{init}	s_{init}	s_{init}	$x(1)$	$x(4)$	
	0	1	2	3	4	5	rounds

Fig. 5. A triggered unit-delay block (left) and its semantics (right).

Now, consider a given composite block P so that the profiles of all its sub-blocks are known. That is, the interface functions of the sub-blocks of P are semantically defined. The internal diagram of P defines a set of dependencies between these interface functions, corresponding to the *scheduling dependency* *graph*, described in Sect. 5.1. If this graph contains a cycle, then the semantics of P is undefined. Otherwise, the semantics is defined in terms of a new function f_P of the same form as in (1). f_P is obtained by function composition of the interface functions of the sub-blocks of P. Acyclicity of the scheduling dependency graph guarantees that the composition is well-defined.

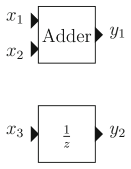
4 Profiles: An Abstract, Compositional and Executable Representation of Synchronous Block Diagrams

4.1 Profiles

A *profile* can be seen as an *interface* or a *summary* of a block type. All blocks, atomic or composite, have profiles. A block may have multiple profiles, each suited for different purposes. This will become clear when we discuss tradeoffs in Sect. 4.3.

The profile of a block contains:

- A list of *interface functions* and their *signatures*. Each such function takes as input a set of values corresponding to some of the input ports of the block, and returns as output a set of values corresponding to some of the output ports of the block. The signature specifies which ports the arguments of the function correspond to, their data types, and so on.[5]
- A *profile dependency graph* (PDG). The PDG is a directed, acyclic graph (DAG), the nodes of which are the interface functions listed in the profile. The PDG specifies the correct order in which these functions are to be called at every synchronous round. If $f \rightarrow g$ is an edge in the PDG, then function f must be called before function g.



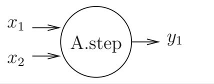
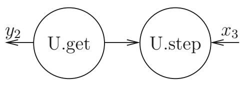
	Interface functions	Profile dependency graphs
Profile of Adder: (combinational)	A.step(x_1, x_2) returns y_1;	
Profile of $\frac{1}{z}$: (Moore)	U.get() returns y_2; U.step(x_3) returns void;	

Fig. 6. Profiles for Adder and Unit-Delay blocks.

For example, Fig. 6 shows the profiles of an Adder block and a Unit-Delay block. Data types are omitted from the signatures of the profiles. The inputs and

[5] For sequential blocks (i.e., blocks with internal state) profiles contain a special `init` function that initializes the state. In our framework, `init` functions of macro blocks are synthesized from `init` functions of their sub-blocks. This is a simple procedure whose details are omitted.

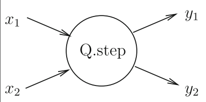
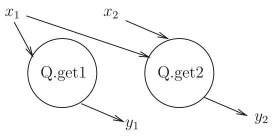
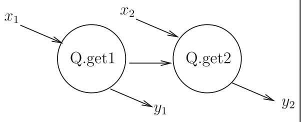
Profile 1:	Profile 2:	Profile 3:
Interface functions:	Interface functions:	Interface functions:
`Q.step(x1, x2) returns (y1, y2);`	`Q.get1(x1) returns y1;` `Q.get2(x1,x2) returns y2;`	`Q.get1(x1) returns y1;` `Q.get2(x2) returns y2;`
Profile dependency graph:	Profile dependency graph:	Profile dependency graph:
		

Fig. 7. Three possible profiles for block Q of Fig. 3.

outputs of the interface functions are also shown on the nodes of the PDG: note that this information can be derived from the signatures of the interface functions. Figure 2 shows two possible profiles for macro block P of Fig. 1. Figure 7 shows three possible profiles for macro block Q of Fig. 3. As these examples illustrate, a given block can have more than one profile. Indeed, different profiles realize different trade-offs and thus are more or less suited in different situations, as discussed in Sect. 4.3. Also note that different blocks can have identical profiles, as illustrated by Figs. 2 and 7.

Profiles that contain a single interface function are called *monolithic*. The profile of the Adder in Fig. 6 is monolithic, whereas the profile of $\frac{1}{z}$ is not. Profile 1 in Figs. 2 and 7 is monolithic, whereas the other profiles shown in these figures are non-monolithic.

Let P be a macro block and consider a profile of P. The PDG of the profile induces a set of dependencies between inputs and outputs of the block. In particular, output y depends on input x if the PDG has a directed path from x to y. On the other hand, the internal diagram of P, together with the profiles of all sub-blocks of P, also induce a set of dependencies between inputs and outputs. These dependencies are captured in the *scheduling dependency graph*, formalized in Sect. 5.1. Here we discuss them informally, through examples. For instance, from the internal diagram of block P of Fig. 1 we can deduce that y_1 does not depend on x_2. Now, y_1 may or may not depend on x_1, depending on the profile of A. If A has a monolithic profile, then its output depends on its input, thus, y_1 depends on x_1. If A has a profile like the one of the Moore block $\frac{1}{z}$ then y_1 does not depend on x_1.

We require that all input-output dependencies that are induced by the internal graph of a block P and the profiles of its sub-blocks are also induced by the PDG of the profile of P. We then say that the profile of P is *sound*. The profile synthesis methods presented in Sect. 5 guarantee that the generated profiles are sound.

Note that the profile of a block is independent of whether the block is triggered or not. Indeed, whether the block is triggered is not a property of the block, it is a property of its context: the same block (type) may be triggered in some diagrams and not triggered in other diagrams. The same profile for this block can be used in both cases. Triggering *will* affect how the profile is used, however, as explained in Sect. 5.

4.2 Profile Implementations

The profile contains a list of interface functions. These functions are implemented in a given programming language, e.g., C++ or Java. The implementations of these functions are part of the *profile implementation*. The latter also includes state and other internal variables, encapsulated in some form, depending on the mechanisms that the programming language provides (e.g., a C++ or Java class).

For example, the profile implementations of the Adder and Unit-Delay blocks (Fig. 6) are given below in object-oriented pseudo-code:

```
class Adder {
  Adder.step( x1, x2 )
      returns y1
  {
    return (x1 + x2);
  }
}
```

```
class UnitDelay {
  private state;

  UnitDelay.init() { state := ... }

  UnitDelay.get() returns y2 {
    return state;
  }

  UnitDelay.step( x3 ) {
    state := x3;
  }
}
```

The implementations of the two profiles of P shown in Fig. 2 are as follows:

Monolithic profile:

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

Non-monolithic profile:

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

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

In the above example we have assumed monolithic profiles for sub-blocks A and B of P, with functions `A.step` and `B.step`, respectively. Unless otherwise stated, we assume monolithic profiles for sub-blocks in all examples that follow.

The implementation of Profile 3 of block Q, shown in Fig. 7, is as follows:

```
Q.get1(x1) returns y1 {
  (z1,z2) := A.step(x1);
  y1 := B.step(z1);
  return y1;
}
```

```
Q.get2(x2) returns y2 {
  y2 := C.step(z2,x2);
  return y2;
}
```

More examples of profile implementations are given in the sequel.

4.3 Trade-Offs

As can be seen from the examples above, the same block can have multiple different profiles. These accomplish different trade-offs, some of which are discussed below.

Modularity vs. Reusability. This is probably the most important tradeoff. We define reusability in terms of the set of contexts (i.e., diagrams) that the profile can be used in: the larger this set, the more reusable the profile is (note that this is a partial order). It follows that monolithic profiles are no more reusable than non-monolithic ones. Often they are strictly less reusable (e.g., examples of Figs. 2 and 7). Reusability is directly related to the set of IO dependencies defined by the PDG of a profile. The larger the set of IO dependencies, the less reusable the profile. A monolithic profile is the least reusable, as it contains all possible IO dependencies. A profile is *maximally reusable* if it contains exactly those IO dependencies contained in the internal diagram of the corresponding macro block. (A profile cannot contain less dependencies, otherwise it would not be sound.) The non-monolithic profiles of Figs. 2 and 7 are both maximally reusable.

Modularity, in our framework, is a *quantitative* notion: it is measured in terms of the size of the profile, for instance, the number of interface functions, or the size of the PDG. The *smaller* the profile, the more modular it is. In that sense, the most modular profile is the monolithic profile. This definition is justified by a number of considerations. First, *scalability*: the complexity of profile synthesis algorithms is a direct function of the size of the profiles, thus, the smaller the profiles, the better the algorithms scale. A second consideration has to do with IP concerns: the smaller the profile, the less details it reveals about the internals of the block, therefore, the more the block appears as a "black box" to its user.

From the above definitions, it follows that modularity and reusability are in conflict. To optimize modularity we are led towards monolithic profiles, but we may have to pay a price in terms of reusability. Both examples in Figs. 2 and 7 illustrate this trade-off.

Modularity vs. Code Size and Other Metrics. An interesting set of trade-offs arise between modularity and various metrics of the profile implementation,

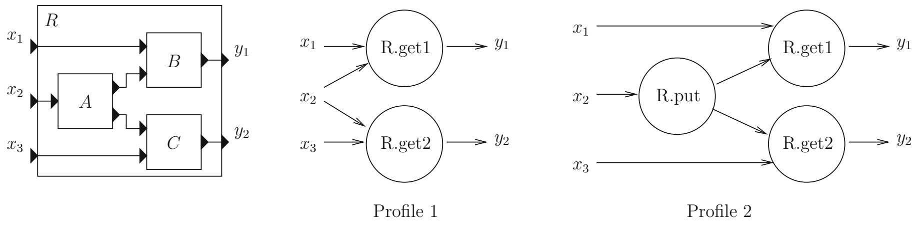


Fig. 8. A macro block R (left) and two possible profiles for R (middle and right).

such as size of the code that implements the interface functions, run-time performance (e.g., worst-case execution time), and so on. We illustrate such trade-offs here through an example. More details can be found in [30].

Figure 8 shows a macro block R and two maximally-reusable profiles for R. Profile 1 is smaller, since it contains only 2 functions, whereas Profile 2 contains 3. The implementation of Profile 1 is as follows:

```
R.get1(x1,x2) returns y1 {
 if ( c = 0 ) {
  (z1, z2) := A.step(x2);
}
 c := (c + 1) modulo 2;
 return B.step( x1, z1 );
}
```

```
R.get2(x2,x3) returns y2 {
 if ( c = 0 ) {
  (z1, z2) := A.step(x2);
}
 c := (c + 1) modulo 2;
 return C.step( z2, x3 );
}
```

It can be seen that the first three lines of code in `P.get1` and `P.get2` are identical. These lines serve to guard execution of `A.step`, which should only be called once per synchronous round. Since the order of calling `P.get1` and `P.get2` depends on the context of R, it is not known at compile-time which function will first call `A.step`, and the choice is made at run-time.

The implementation of Profile 2 of Fig. 8 is as follows:

```
R.put(x2) {
 (z1,z2) :=
    A.step(x2);
}
```

```
R.get1(x2) returns y1 {
return B.step(x1,z1);
}
```

```
R.get2(x3) returns y2 {
return C.step(z2,x3);
}
```

This implementation has better characteristics than the previous one: it contains no conditionals and no code replication. Thus, the code is both smaller in size and also executes faster. Such differences may seem small in this example, but they can be critical in the context of a real embedded application, where memory and execution time are often scarce resources.

Algorithmic Complexity Trade-Offs. Another set of trade-offs concerns the complexity, in theory or in practice, of the algorithms involved in profile synthesis, against other metrics such as modularity, reusability, or code characteristics. For example, producing a monolithic profile is trivial and inexpensive. Synthesizing non-monolithic profiles involves more sophisticated algorithms such as clustering. Many of these algorithms have polynomial worst-case complexity, but may result in profiles that are non-optimal in terms of modularity, or that cannot be implemented without conditional code, as with Profile 2 of Fig. 7, or Profile 1 of Fig. 8. These issues are discussed in more detail in Sect. 5.

4.4 Abstraction, Compositionality and Executability

In summary, profiles in general, and non-monolithic profiles in particular, form a modular, compositional and executable representation of hierarchical SBDs.

They are executable in the sense that every interface function comes with a piece of executable code: by calling these functions in an order that respects the dependencies prescribed in the PDG, we have an implementation of the semantics of the model. This is in contrast, for instance, to non-executable representations that simply maintain input-output dependencies, as in the works of [2,48]. Profiles are compositional in the sense that a diagram of profiles can be abstracted into a single profile without any loss of information, that is, preserving exactly the same set of input-output dependencies. Finally, profiles are abstract in the sense that they allow many of the internal details of composite blocks to be omitted (e.g., as in Fig. 7) which results in a more compact and thus less costly representation.

5 Profile Synthesis and Code Generation

In this section we describe how profiles and their implementations can be generated automatically. In summary, our method takes as inputs:

1. a macro block M with its internal block diagram;
2. a profile for each type of sub-block of M; and
3. a set of user constraints or goals;

and automatically generates as outputs:

1. a profile for M (this part of the process is called *profile synthesis*);
2. the implementation of the profile in a certain programming language such as C++ or Java.

User constraints and goals include any sort of information that the user may provide to influence the profile and code that is generated. This includes modularity vs. reusability preferences, desired code characteristics, and so on. In practice, this type of information is given as options and inputs to the algorithms involved in the different steps of the process, discussed below.

It is worth noting that although the profiles of the sub-blocks of M are required in the profile synthesis and code generation process, the implementation of the interface functions of these profiles is not required. The implementation of the sub-blocks of M is only required for model execution. This is another aspect of modularity in our approach, and a desirable feature especially for IP reasons, or treating blocks as "black boxes". In particular, only executable code (e.g., object files) need to be made available to the user, and not source code.

Profile synthesis can be applied to SBDs of arbitrary hierarchy depths, in a bottom-up manner. Starting with macro blocks that contain only atomic blocks, synthesizing a profile for the former, and then moving up the hierarchy. Profiles of atomic blocks are inputs to this process. They can be produced "manually", or automatically, for instance, by some method that automatically extracts summaries from the implementation of blocks. How to do this is beyond the scope of this paper. Note that once a profile has been synthesized for a macro block, the latter can be viewed as an atomic block, since no information about its internals (e.g., its internal diagram) is any longer necessary. Thus, apart from the information contained in the profile, the block is a "black box".

5.1 Profile Synthesis

Profile synthesis consists in synthesizing a profile for a macro block M given the internal diagram of M and profiles for all sub-blocks of M. Profile synthesis involves a number of sub-steps, described below:

Building the Scheduling Dependency Graph (SDG). The SDG is a directed graph obtained by connecting the PDGs of the profiles of all sub-blocks of M. The connections are made according to the internal diagram of M, that is, by inserting an edge $f \rightarrow g$ if an output of function f is connected in the diagram to an input of function g.

Consider the example of Fig. 9. At the top left of the figure is shown a macro block M and its internal diagram. At the top right of the figure are shown the profiles of all sub-blocks of M. Sub-blocks A and C have a single interface function each, 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. 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 round.

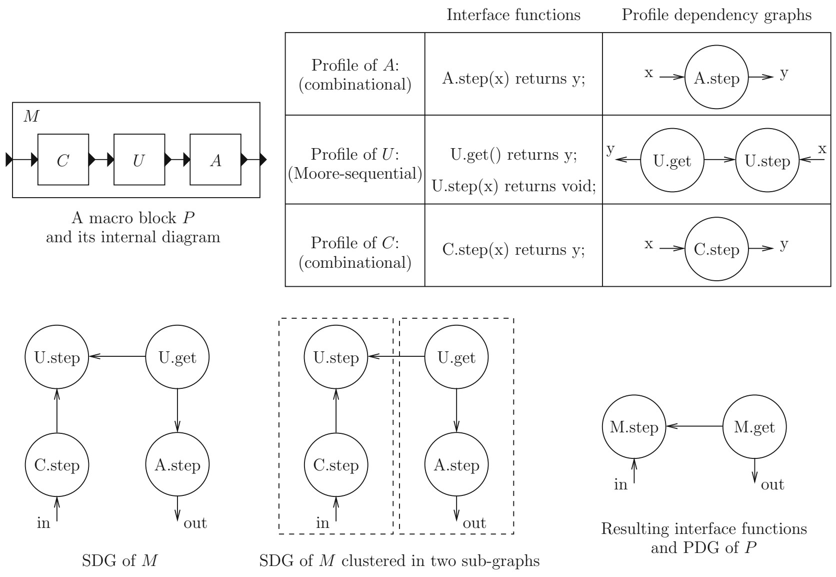


Fig. 9. Example of profile synthesis.

The SDG of block M is shown at the bottom left of Fig. 9. The SDG of M has been produced by connecting the PDGs of sub-blocks C, U and A. For instance, the output port of C is connected to the input port of U. This results in adding

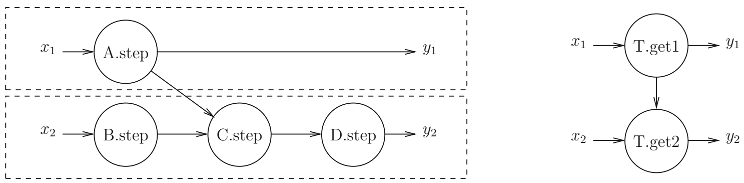


Fig. 10. Left: clustered SDG of macro block T of Fig. 3. Right: profile for T.

a directed edge from `C.step` (which produces the output of C) to `U.step` (which consumes the input of U) to the SDG of M. Similarly with the rest of the edges added to the SDG of M.

When the internal diagram of M contains triggers, these are handled by adding directed edges from the interface function that produces the trigger to all interface functions of the triggered sub-block of M. For example, the SDG of macro block T of Fig. 3 is shown in Fig. 10.

Dependency Analysis. Once the SDG of M is built, it is checked to see whether it contains a directed cycle. If it does, then this implies a cyclic dependency that cannot be resolved, either because the original diagram has a true such dependency, or because the profiles of the sub-blocks of M are too coarse (i.e., not reusable in the context of M). A cycle in the SDG results in rejecting the diagram and stopping the profile synthesis process. If the SDG is acyclic, we proceed to the *clustering* step.

Clustering. Clustering consists in grouping the nodes of the SDG G of M into a set of *clusters*. Every node of G must be included in at least one cluster, however, the clusters need not be *disjoint*, i.e., some nodes may belong to more than one clusters. Each cluster can also be seen as a *sub-graph* of G that contains all the nodes in the cluster along with all dependencies between any two such nodes. For purposes of clustering, input and output ports of M are also considered to be nodes of the SDG, called *input* and *output nodes*, respectively. For example, x_1, x_2 are input nodes in the SDG of T shown in Fig. 10 and y_1, y_2 are output nodes. Nodes that have no outputs (and therefore no outgoing edges either) are called *terminal* nodes. For example, node `U.step` in the SDG of M shown in Fig. 9 is a terminal node.

Once clustering is fixed, each cluster is mapped into an interface function for M. Therefore, the number of clusters is equal to the number of interface functions contained in the synthesized profile of M.

Dependencies between nodes of G that belong to different clusters induce dependencies between those clusters. In the case of disjoint clusters, these dependencies are defined as follows. Let C_1, C_2 be two clusters and let f_1, f_2 be two nodes of G such that $f_1 \in C_1$ and $f_2 \in C_2$. (Notice that, since clustering is assumed to be disjoint, $f_1 \notin C_2$ and $f_2 \notin C_1$.) A dependency $f_1 \rightarrow f_2$ in G induces a dependency $C_1 \rightarrow C_2$ between the two clusters. In the case of *overlapping* (i.e., non-disjoint) clusters, how the dependencies between clusters are defined generally depends on the clustering algorithm. An example is the O^{2}C algorithm, explained in Sect. 5.1 below.

Once the dependencies between clusters are fixed, they define a directed graph G_M whose nodes are clusters. Since every cluster corresponds to an interface function for M, G_M is also a graph whose nodes are interface functions of M. Therefore, G_M is the PDG of M. G_M needs to be acyclic, therefore, care must be taken so that clustering results in no cyclic dependencies between clusters. The clustering algorithms that we discuss below all have this property. Moreover, the profile of M must be sound, which means that all input-output dependencies included in G, the SDG of M, must also be included in G_M, the PDG of M. This always holds in the case of disjoint clusterings, as follows from the definition of G_M given above. For overlapping algorithms, care must be taken so that the definition of inter-cluster dependencies results in a PDG that is sound. In Sect. 5.1 we show that this is the case for the O^{2}C algorithm.

From the above discussion, it follows that clustering completely determines the profile of M. This is why clustering is the most important step in profile synthesis. It is also a step where different choices can be made, that lead to different trade-offs. For instance, a coarse-grain clustering results in a more modular profile than a fine-grain clustering. In particular, grouping all nodes of the SDG into a single cluster results in a monolithic profile. A clustering that introduces false input-output dependencies, that is, input-output dependencies not existing in the SDG, results in a non-maximally-reusable profile. A clustering with *disjoint* clusters (i.e., clusters that do not share nodes) results in code without conditionals or replication. Let us illustrate these points through examples.

In Fig. 9, the SDG of M is clustered in two sub-graphs, resulting in a two-function profile for M, shown to the bottom-right of the figure. Observe that the profile of M is identical to the profile of the unit-delay block $\frac{1}{z}$ (Fig. 6). This is not a coincidence, since this is a maximally-reusable and optimal in terms of modularity profile for all Moore blocks. The example of Fig. 9 also illustrates a simple case of reduction in size, where a macro block with three sub-blocks has the same profile as one of its sub-blocks.

Other examples of clustering are the following:

– The non-monolithic profile of P shown in Fig. 2 is produced by grouping all nodes in the PDG of A in one cluster (corresponding to `P.get1`) and all nodes in the PDG of B in a second cluster (corresponding to `P.get2`).
– The non-monolithic profile of Q shown in the middle of Fig. 7 is produced by grouping nodes of A and B in one cluster (corresponding to `Q.get1`) and nodes of A and C in a second cluster (corresponding to `Q.get2`). In this case the clusters overlap (i.e., are not disjoint). A similar clustering produces the profile shown in the middle of Fig. 8.
– The non-monolithic profile of Q shown to the right of Fig. 7 is produced by grouping nodes of A and B in one cluster (corresponding to `Q.get1`) and nodes of C in a second cluster (corresponding to `Q.get2`). In this case the clusters are disjoint.

– The profile of R shown to the right of Fig. 8 is produced by grouping nodes of A in one cluster (corresponding to `R.put`), nodes of B in a second cluster (corresponding to `R.get1`), and nodes of C in a third cluster (corresponding to `R.get2`).

We now briefly describe some clustering algorithms.

Step-Get Clustering (SGC). The SGC algorithm, proposed in [32], generates at most two interface functions for a given macro block M. In particular, if M is Moore, then SGC generates two interface functions for M: an `M.get` function that computes the outputs of M and an `M.step` function that updates the state of M. This is an optimal, in terms of modularity, and maximally-reusable profile for all Moore blocks. If M is not Moore, then SGC generates a single interface function for M, that is, a monolithic profile. In this case the profile is not maximally reusable, in general.

The SGC algorithm is simple. It starts by analyzing the SDG of M, checking whether there exists some output of M that depends on some input. If this is the case, then M is not Moore, and SGC produces a single cluster containing all nodes in the SDG. Otherwise, the SDG can be partitioned into two sub-graphs, a "right" sub-graph that contains all nodes that have a path to some output, and a "left" sub-graph that contains all the remaining nodes. The "right" and "left" sub-graphs correspond to `M.get` and `M.step`, respectively. SGC has polynomial worst-case complexity.

The SGC algorithm produces the clustering of Fig. 9 where block M is Moore. For all other examples given in the paper, SGC produces a monolithic profile.

Optimal Overlapping Clustering (O^2C). The O^2C algorithm is an improved variant of the *dynamic clustering* algorithm proposed in [32]. O^2C achieves maximal reusability and optimal modularity, that is, a minimal number of clusters (subject to the maximal reusability constraint). Moreover, O^2C is guaranteed to generate no more than $n+1$ clusters in the worst case, where n is the number of outputs of macro block M, and no more than n clusters if M is combinational. O^2C has polynomial worst-case complexity. The only drawback of O^2C is that it may result in overlapping clusters, therefore, in profile implementations that require conditionals.

O^2C executes the following procedure:

```
for each output node or terminal node f do {
  create a cluster C := { f };
  while there exist nodes g in C and h not in C s.t. h -> g do
    add h to C;
}
merge clusters containing exactly the same sets of input nodes;
for each terminal cluster C do
  if there exists cluster C' s.t. inputs(C) is a subset of
     inputs(C') then merge C with C';
merge all remaining terminal clusters (if any) into a single
        cluster;
```

where a *terminal cluster* is a cluster containing a terminal node, and *inputs*(C) denotes the set of input nodes contained in cluster C.

O^{2}C starts by creating a cluster for every output node and every terminal node in the SDG of M. Then the *backward closure* of each of these clusters is computed, by adding all *predecessor* nodes, i.e., all nodes that have a directed edge to some node already in the cluster, until no new nodes can be added. If at this point two clusters C_1 and C_2 contain the same sets of input nodes, i.e., $\mathit{inputs}(C_1) = \mathit{inputs}(C_2)$, then these clusters are *merged* into a single cluster $C_1 \cup C_2$, and this is repeated until no more clusters can be merged in this way. Then, for each terminal cluster C for which there exists cluster C' such that $\mathit{inputs}(C) \subseteq \mathit{inputs}(C')$, C and C' are merged into $C \cup C'$, and again the process is repeated until no terminal clusters can be merged in this way. Finally, all remaining terminal clusters (if any) are merged into a single cluster named `M.step`. The rest of the clusters are named `M.get1`, `M.get2`, and so on.

The inter-cluster dependencies defined by O^{2}C are as follows: all clusters of type `M.get`i are independent from each other; if there is a cluster `M.step`, then there is a dependency `M.get`i $\rightarrow$ `M.step`, for every cluster of type `M.get`i. In other words, interface function `M.step` must be called after all interface functions `M.get`i are called, and the latter can be called in any order.

Examples of profiles produced by O^{2}C are: Profile 2 of block P in Fig. 2; Profile 2 of block Q in Fig. 7; Profile 1 of block R in Fig. 8; and the profile of block M in Fig. 9.

O^{2}C has the following properties:

First, if M has n output ports, and therefore the SDG of M has n output nodes, O^{2}C will generate at most $n + 1$ clusters. This is because there can be at most one cluster per output node, plus at most one terminal cluster, `M.step`, when O^{2}C terminates. It can be also shown that for combinational blocks, i.e., blocks without internal state, there can be at most n clusters.

Second, O^{2}C produces a sound and acyclic PDG. Acyclicity follows from the fact that the only edges in the PDG of M are from some node `M.get`i to `M.step`, if it exists, and the latter has no outgoing edge. To see why the PDG is also sound, consider a path $x \rightarrow f_1 \rightarrow \cdots \rightarrow f_n \rightarrow y$ in the SDG of M, from some input node x to some output node y. Let C be the cluster that contains y. C is closed by predecessors, therefore, all nodes in the above path from x to y are contained in C. Thus, the dependency $x \rightarrow y$ is maintained in the PDG of M.

Third, O^{2}C achieves maximal reusability, that is, every IO dependency $x \rightarrow y$ in the PDG of M is a true IO dependency. Consider such a dependency. By definition of the PDG, there exists a cluster C such that both x and y are in C. Since y is in C, C cannot be a terminal cluster. C is generally the result of merging clusters $C_1, \ldots, C_k$ produced in the first `for each` loop of O^{2}C, for $k \geq 1$. By definition, the set $\mathit{inputs}(C_i)$ is the same for all $i = 1, \ldots, k$. Therefore, there exists C_i such that both x and y are in C_i, and C_i was obtained by computing the backward closure of y. Thus, there is a path from x to y in the SDG of M, and $x \rightarrow y$ is a true dependency.

Fourth, O^{2}C is optimal, that is, there exists no clustering with fewer clusters that achieves maximal reusability. Suppose such a clustering $\mathcal{C}^*$ exists. $\mathcal{C}^*$ must merge in one cluster at least two nodes f_1, f_2 that the clustering $\mathcal{C}$ produced by O^{2}C separates in two different clusters, $f_1 \in C_1$ and $f_2 \in C_2$. Suppose, first, that both C_1, C_2 are of type `M.get`i. Then $\mathit{inputs}(C_1) \neq \mathit{inputs}(C_2)$, otherwise C_1 and C_2 would have been merged by O^{2}C. Let, without loss of generality, $x \in \mathit{inputs}(C_2) \setminus \mathit{inputs}(C_1)$. Let y be an output node in C_1 (C_1 is not terminal, so it must contain at least one output node). Then merging f_1 and f_2 introduces false IO dependency $x \to y$, thus, $\mathcal{C}^*$ cannot be maximally reusable. Now suppose C_1 is of type `M.get`i and C_2 is `M.step`. Then $\mathit{inputs}(C_2) \not\subseteq \mathit{inputs}(C_1)$, otherwise terminal cluster C_2 would have been merged with C_1. Thus, we can find again $x \in \mathit{inputs}(C_2) \setminus \mathit{inputs}(C_1)$ and $y \in C_1$ and repeat the last argument.

Note that O^{2}C may unnecessarily produce an overlapping clustering. This means that there exists a disjoint clustering with the same number of clusters that is also maximally reusable. For example, for the SDG shown in Fig. 10, O^{2}C would produce an overlapping clustering where `A.step` is shared between two clusters, whereas a disjoint clustering of two clusters exists, as shown in the figure.

Optimal Disjoint Clustering (ODC). The ODC algorithm, proposed in [30], guarantees, like O^{2}C, maximal reusability. Unlike O^{2}C, ODC always produces disjoint clusters. Finally, ODC generates a minimal number of clusters, subject to the maximal reusability and disjointness constraints. Unfortunately, the problem of partitioning a DAG into a minimal number of disjoint clusters without introducing false input-output dependencies is NP-complete [30]. Thus, the worst-case complexity of ODC is exponential. Nevertheless, ODC uses powerful SAT solvers and performs well in practice, as the experimental results reported in [30] show.

We will only sketch the main ideas behind ODC, and refer the reader to [30] for the details, which are involved. ODC executes the following procedure:

```
partition output nodes according to input dependencies;
let k be the number of output partitions;
i := k;
repeat
  build a boolean formula stating that a solution
      with i clusters exists;
  call a SAT solver to check whether the formula is satisfiable;
  if formula is satisfiable then solution found
  else i := i+1;
until solution found;
```

The first step consists in partitioning the outputs of M into a set of disjoint partitions such that in every partition, all output nodes depend on exactly the same set of input nodes. If k partitions are found then there can be no less than k clusters required to achieve maximal reusability. Indeed, it can be seen that O^{2}C produces at least k clusters in this case, and since O^{2}C is optimal, at least k disjoint clusters are needed to achieve maximal reusability. Therefore, we start the iteration by setting i to k. For each i, we build a boolean formula that encodes the existence of a solution with i clusters. A "solution" means a disjoint clustering that introduces no false input-output dependencies. A SAT solver is used to check satisfiability of the formula (and also produce a solution). If the formula is satisfiable we have found a solution, otherwise, no solution with i clusters exists, and we increment i. The procedure is guaranteed to terminate when i reaches the number of nodes in the SDG of M: indeed, clustering every node separately is obviously a valid disjoint clustering.

Examples of profiles produced by ODC are: Profile 3 of block Q in Fig. 7; Profile 2 of block R in Fig. 8; and the profile of block T in Fig. 10.

Disjoint Clustering Heuristics. A number of heuristics can be used to produce disjoint clusterings that are maximally-reusable, albeit not always optimal in terms of modularity. A simple heuristic is to use O^{2}C and check whether the clustering it produces is disjoint: if it is we are done, otherwise, we turn it into a disjoint clustering by somehow separating shared nodes (a trivial method is to cluster every such node separately). Other, more sophisticated heuristics, are proposed in [42]. More experimental work is needed to evaluate how these algorithms compare in practice, both in terms of execution time, as well as in terms of the optimality of results produced. Such experiments are beyond the scope of this paper.

Non-Maximally-Reusable Clusterings. All algorithms discussed above, with the exception of SGC, are guaranteed to introduce no false input-output dependencies, thus producing maximally-reusable profiles. In SGC, on the other hand, the user has no way of "controlling" the reusability of the produced profile. In some cases, it may be desirable to relax the requirement on maximal reusability, for instance, in order to gain in modularity. This may be the case, for example, if it is known that a given block will never be connected in a context with feedback. Then, a monolithic profile suffices. More generally, it may be known that, even though a certain output y of the block does not depend on a certain input x, it is "safe" to introduce a false dependency $x \rightarrow y$. That is, such a false dependency is known not to result in serious restrictions in the set of contexts that the block can be used in. The above algorithms can be modified so as to allow the user to provide this type of information, thus being able to control the reusability of the produced profile. For instance, in ODC, the encoding of the formula can be modified so that it selectively allows some false IO dependencies, whereas it forbids the rest.

5.2 Code Generation

Profile synthesis determines the profile of macro block M given as input. In the code generation step, code that implements each interface function in the profile of M is generated in a language such as C++ or Java. Any internal state variables, or other persistent variables needed to communicate data between different calls of the interface functions are generated as well. Together with an `init` function, these functions and data can be encapsulated in a class or other object-oriented mechanism that the target language may provide. In Java, for instance, the code is encapsulated in a Java class, the interface functions become public methods of this class, and the variables become private variables of the class.

The principle of generating code implementing the interface functions is the following. Every interface function f_i corresponds to a sub-graph G_i of the SDG, produced in the clustering step. To generate code for f_i, the nodes of G_i are ordered in a total order that respects the dependencies of G_i: the SDG of M is acyclic, therefore G_i is also acyclic and such a total order always exists. Now, every node of G_i corresponds to an interface function of some sub-block of M. The code of f_i then consists in calling these functions in the order specified above.

Interface functions generally need to communicate data to one another. This arises when an output y of a sub-block A of M is connected to an input x of another sub-block B, and the interface functions producing y and consuming x belong to different clusters. If f and g are two such functions, then a persistent variable `z` is created, to store the value of y. `z` is persistent in the sense that it maintains its value across calls to f and g. It can be implemented, for instance, as a private variable in the class generated for M.[6] Whenever f is called, it writes to `z` and whenever g is called, it can read from `z`.

Following the above principles, we can generate code for all interface functions without conditionals given in our examples so far. For the additional example of block M of Fig. 9, the implementation of the interface functions is shown below:

```
M.get( ) returns out {                 M.step( in ) {
  return A.step( U.get() );              U.step( C.step(in) );
}                                      }
```

Slight complications arise in two cases: first, in the case of overlapping clusters; second, in the case of diagrams with triggers. Both cases require code with conditionals.

In the case of overlapping clusters, the objective is to generate code that ensures that, despite overlapping, every interface function of every sub-block of M is called only once at every synchronous round. Notice that calling an interface function more than once is generally *incorrect*, since the function may modify some state variables. Even when a function does not update state variables, calling it more than once in a round is wasteful, thus we want to avoid it.

To do this, we use a counting scheme that keeps track of how many times a function has been called so far in the synchronous round. In particular, for each interface function f of a sub-block of M, let N_f be the number of clusters that f is included in. If $N_f > 1$ (i.e., f is shared among multiple clusters) then we create a modulo-N_f counter for f, denoted c_f: the counter is initialized to 0 and "wraps" again to 0 when its value reaches N_f. Each such counter is part of the persistent internal variables of the class of M. Counter c_f indicates whether f has already been called in the current round: *f has been called iff $c_f > 0$*. Every

[6] If f and g belong to the same cluster, a local variable in the corresponding interface function of M can be used to store the value of y.

call to a function f that has $N_f > 1$ is guarded by the condition $c_f = 0$. The counter is incremented by 1, independently of whether the condition is true or false. An example of using this technique is the implementation of the profile shown in the middle of Fig. 8.

We now turn to the case of diagrams with triggers. First, we identify all sub-blocks of M that are triggered. To do this, we use the internal diagram of M.[7] Let A be a triggered sub-block of M. For every output port y of A, a persistent variable z_y is generated in the profile implementation of M. This variable is initialized to the trigger-initial value for y specified in the diagram. It is updated every time A is triggered, and maintains its previous value in other rounds. Let f be an interface function of A. Let t be the signal that triggers A: t is either an input of M, or is produced by some other sub-block of M.[8] When generating code for M, every call to f is embedded in a conditional, guarded by t: if t is true then f is called, otherwise it is not.

For example, consider macro block T of Fig. 3 and suppose its SDG is clustered in two clusters, as shown in Fig. 10. The synthesized profile for T is then as shown to the right of Fig. 10 and the implementation of the two interface functions is as follows:

```
T.get1( x1 ) returns y1 {
  z1 := A.step(x1);
  return z1;
}
```

```
T.get2( x2 ) returns y2 {
  local tmp := B.step(x2);
  if (z1) {
    z2 := C.step(tmp);
  }
  return D.step(z2);
}
```

Persistent variables `z1` and `z2` have been added for communication between the two interface functions and for the output of triggered sub-block C, respectively. Note that, even when `z1` is false, `z2` has a well-defined value because it is initialized to the trigger-initial value specified for C.

6 Conclusions and Perspectives

We have proposed non-monolithic profiles as an abstract, compositional and executable representation of hierarchical synchronous block diagrams. Our work offers the unified treatment of the problem of modular code generation from synchronous models which has been lacking so far. A prototype implementation of our methods exists and experimental results reported in [30] encourage us to believe that the approach is also feasible and relevant in practice.

A number of issues remain open. Clustering is of course a topic in itself, as mentioned above. Apart from devising new or evaluating new and existing

[7] Note that information about triggers is lost in the SDG of M: indeed, in the SDG, dependencies arising due to triggers and those arising due to port connections are indistinguishable.

[8] t cannot be produced by A, as this would result in a cycle in the SDG of M.

clustering algorithms, another aspect of particular interest is integrating user controls in the algorithms. For example, the user could specify which input-output dependencies can be relaxed, i.e., which false input-output dependencies can be admitted in order to obtain better clusterings.

Enriching profiles with additional information is another interesting direction. It has been partly explored in [31] in the case of *timed* diagrams, a subclass of triggered diagrams where triggers are known at compile time (e.g., they are periodic). In that paper, it is shown how profiles can be enriched with finite-state automata representing the set of rounds when a given block is triggered. This allows to avoid redundant function calls in the generated code, and also to identify false IO dependencies during profile synthesis.

References

1. de Alfaro, L., Henzinger, T.: Interface automata. In: Foundations of Software Engineering (FSE). ACM Press (2001)
2. Alur, R., Henzinger, T.: Reactive modules. Formal Methods Syst. Des. **15**, 7–48 (1999)
3. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proc. IEEE **91**(1), 64–83 (2003)
4. Benveniste, A., Le Guernic, P., Aubry, P.: Compositionality in dataflow synchronous languages: specification & code generation. Technical report 3310, Irisa - Inria (1997)
5. Berry, G.: The Constructive Semantics of Pure Esterel (1999). http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf
6. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992)
7. Blochwitz, T., Otter, M., et al.: The functional mockup interface for tool independent exchange of simulation models. In: Proceedings of the 8th International Modelica Conference. Linkoping University Electronic Press (2011). http://www.ep.liu.se/ecp/063/013/ecp11063013.pdf
8. Blochwitz, T., Otter, M., et al.: Functional mock-up interface 2.0: the standard for tool independent exchange of simulation models. In: Proceedings of the 9th International Modelica Conference. Linkoping University Electronic Press (2012). http://www.ep.liu.se/ecp/076/017/ecp12076017.pdf
9. Bogomolov, S., Greitschus, M., Jensen, P.G., Larsen, K.G., Mikucionis, M., Strump, T., Tripakis, S.: Co-simulation of hybrid systems with SpaceEx and Uppaal. In: Proceedings of the 11th International Modelica Conference. Linkoping University Electronic Press (2015). http://www.ep.liu.se/ecp_article/index.en.aspx?issue=118;article=017
10. Brandt, J., Schneider, K.: Separate compilation for synchronous programs. In: SCOPES 2009: 12th International Workshop on Software and Compilers for Embedded Systems, pp. 1–10 (2009)
11. Broman, D., Brooks, C., Greenberg, L., Lee, E.A., Tripakis, S., Wetter, M., Masin, M.: Determinate composition of FMUs for co-simulation. In: Proceedings of the 13th ACM and IEEE International Conference on Embedded Software (EMSOFT 2013), pp. 2:1–2:12. IEEE (2013)
12. Broman, D., Greenberg, L., Lee, E.A., Masin, M., Tripakis, S., Wetter, M.: Requirements for hybrid cosimulation standards. In: Hybrid Systems: Computation and Control (HSCC 2015) (2015)

13. Caspi, P.: Clocks in dataflow languages. Theor. Comput. Sci. **94**, 125–140 (1992)
14. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: LUSTRE: a declarative language for programming synchronous systems. In: 14th ACM Symposium POPL. ACM (1987)
15. Caspi, P., Raymond, P., Tripakis, S.: Synchronous programming. In: Lee, I., Leung, J., Son, S. (eds.) Handbook of Real-Time and Embedded Systems, pp. 14-1–14-21. Chapman & Hall, London (2007)
16. Cremona, F., Lohstroh, M., Broman, D., Natale, M.D., Lee, E.A., Tripakis, S.: Step revision in hybrid co-simulation with FMI. In: 14th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE) (2016)
17. Cremona, F., Lohstroh, M., Tripakis, S., Brooks, C., Lee, E.: FIDE - an FMI integrated development environment. In: 31st ACM/SIGAPP Symposium on Applied Computing, Embedded Systems Track (SAC), pp. 1759–1766. ACM (2016)
18. Dragomir, I., Preoteasa, V., Tripakis, S.: Compositional semantics and analysis of hierarchical block diagrams. In: Bošnački, D., Wijs, A. (eds.) SPIN 2016. LNCS, vol. 9641, pp. 38–56. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32582-8_3
19. Edwards, S., Lee, E.: The semantics and execution of a synchronous block-diagram language. Sci. Comput. Program. **48**, 21–42 (2003)
20. Gamatié, A.: Designing Embedded Systems with the SIGNAL Programming Language. Springer, New York (2009). https://doi.org/10.1007/978-1-4419-0941-1
21. Gautier, T., Le Guernic, P.: Code generation in the SACRES project. In: Redmill, F., Anderson, T. (eds.) SSS 1999, pp. 127–149. Springer, London (1999). https://doi.org/10.1007/978-1-4471-0823-8_9
22. Geilen, M., Tripakis, S., Wiggers, M.: The earlier the better: a theory of timed actor interfaces. In: 14th International Conference Hybrid Systems: Computation and Control (HSCC 2011). ACM (2011)
23. Girault, A.: A survey of automatic distribution method for synchronous programs. In: International Workshop on Synchronous Languages, Applications and Programs, SLAP 2005. ENTCS, Elsevier, Edinburgh, April 2005. ftp://ftp.inrialpes.fr/pub/bip/pub/girault/Publications/Slap05/main.pdf
24. Hainque, O., Pautet, L., Le Biannic, Y., Nassor, É.: Cronos: a separate compilation tool set for modular esterel applications. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1709, pp. 1836–1853. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48118-4_47
25. Henzinger, T., Sifakis, J.: The discipline of embedded systems design. IEEE Comput. **40**(10), 32–40 (2007)
26. Henzinger, T.A., Kirsch, C.M., Matic, S.: Composable code generation for distributed Giotto. In: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2005, pp. 21–30. ACM, New York (2005). https://doi.org/10.1145/1065910.1065914
27. Le Guernic, P., Gautier, T., Borgne, M.L., Lemaire, C.: Programming real-time applications with signal. Proc. IEEE **79**(9), 1321–1336 (1991)
28. Lee, E., Messerschmitt, D.: Synchronous data flow. Proc. IEEE **75**(9), 1235–1245 (1987)
29. Lee, E., Zheng, H.: Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In: EMSOFT 2007: Proceedings of 7th ACM and IEEE International Conference on Embedded software, pp. 114–123. ACM (2007)

30. Lublinerman, R., Szegedy, C., Tripakis, S.: Modular code generation from synchronous block diagrams - modularity vs. code size. In: 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009), pp. 78–89. ACM, January 2009
31. Lublinerman, R., Tripakis, S.: Modular code generation from triggered and timed block diagrams. In: 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008), pp. 147–158. IEEE CS Press, April 2008
32. Lublinerman, R., Tripakis, S.: Modularity vs. reusability: code generation from synchronous block diagrams. In: Design, Automation, and Test in Europe (DATE 2008), pp. 1504–1509. ACM, March 2008
33. Malik, S.: Analysis of cyclic combinational circuits. IEEE Trans. Comput.-Aided Des. **13**(7), 950–956 (1994)
34. Potop-Butucaru, D., Edwards, S., Berry, G.: Compiling Esterel. Springer, New York (2007). https://doi.org/10.1007/978-0-387-70628-3
35. Pouzet, M., Raymond, P.: Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation. In: ACM International Conference on Embedded Software (EMSOFT 2009), pp. 215–224, October 2009
36. Preoteasa, V., Dragomir, I., Tripakis, S.: The refinement calculus of reactive systems. CoRR abs/1710.03979 (2017)
37. Preoteasa, V., Tripakis, S.: Refinement calculus of reactive systems. In: Proceedings of the 14th ACM and IEEE International Conference on Embedded Software (EMSOFT 2014), pp. 2:1–2:10. ACM, October 2014
38. Raymond, P.: Compilation séparée de programmes Lustre. Master's thesis, IMAG (1988). (in French)
39. Schneider, K., Brandt, J., Vecchié, E.: Modular compilation of synchronous programs. In: Kleinjohann, B., Kleinjohann, L., Machado, R.J., Pereira, C.E., Thiagarajan, P.S. (eds.) DIPES 2006. IIFIP, vol. 225, pp. 75–84. Springer, Boston (2006). https://doi.org/10.1007/978-0-387-39362-9_9
40. Shin, I., Lee, I.: Compositional real-time scheduling framework with periodic model. ACM Trans. Embed. Comput. Syst. **7**, 30:1–30:39 (2008). https://doi.org/10.1145/1347375.1347383
41. Strom, R.E., Yemini, S.: Typestate: a programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. **12**(1), 157–171 (1986)
42. Tripakis, S., Bui, D., Geilen, M., Rodiers, B., Lee, E.A.: Compositionality in synchronous data flow: modular code generation from hierarchical SDF graphs. ACM Trans. Embed. Comput. Syst. (TECS) **12**(3), 83:1–83:26 (2013)
43. Tripakis, S.: Bridging the semantic gap between heterogeneous modeling formalisms and FMI. In: International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation - SAMOS XV (2015)
44. Tripakis, S.: Compositionality in the science of system design. Proc. IEEE **104**(5), 960–972 (2016)
45. Tripakis, S., Lickly, B., Henzinger, T.A., Lee, E.A.: A theory of synchronous relational interfaces. ACM Trans. Program. Lang. Syst. (TOPLAS) **33**(4), 14 (2011)
46. Tripakis, S., Stergiou, C., Shaver, C., Lee, E.A.: A modular formal semantics for Ptolemy. Math. Struct. Comput. Sci. **23**, 834–881 (2013)
47. Zeng, J., Edwards, S.A.: Separate compilation for synchronous modules. In: Yang, L.T., et al. (eds.) ICESS 2005. LNCS, vol. 3820, pp. 129–140. Springer, Heidelberg (2005). https://doi.org/10.1007/11599555_15
48. Zhou, Y., Lee, E.: Causality interfaces for actor networks. ACM Trans. Embed. Comput. Syst. **7**(3), 1–35 (2008)