Compositionality in Synchronous Data Flow: Modular Code Generation from Hierarchical SDF Graphs

STAVROS TRIPAKIS and DAI BUI, University of California, Berkeley
MARC GEILEN, Eindhoven University of Technology
BERT RODIERS, LMS International
EDWARD A. LEE, University of California, Berkeley

Hierarchical SDF models are not compositional: a composite SDF actor cannot be represented as an atomic SDF actor without loss of information that can lead to rate inconsistency or deadlock. Motivated by the need for incremental and modular code generation from hierarchical SDF models, we introduce in this paper DSSF profiles. DSSF (Deterministic SDF with Shared FIFOs) forms a compositional abstraction of composite actors that can be used for modular compilation. We provide algorithms for automatic synthesis of non-monolithic DSSF profiles of composite actors given DSSF profiles of their sub-actors. We show how different trade-offs can be explored when synthesizing such profiles, in terms of compactness (keeping the size of the generated DSSF profile small) versus reusability (maintaining necessary information to preserve rate consistency and deadlock-absence) as well as algorithmic complexity. We show that our method guarantees maximal reusability and report on a prototype implementation.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms: Algorithms, Design, Languages

Additional Key Words and Phrases: Embedded software, hierarchy, synchronous data flow, code generation, clustering

ACM Reference Format:
Tripakis, S., Bui, D., Geilen, M., Rodiers, B., and Lee, E. A. 2013. Compositionality in synchronous data flow: Modular code generation from hierarchical SDF graphs. ACM Trans. Embedd. Comput. Syst. 12, 3, Article 83 (March 2013), 26 pages.
DOI: http://dx.doi.org/10.1145/2442116.2442133

1. INTRODUCTION

Programming languages have been constantly evolving over the years, from assembly, to structural programming, to object-oriented programming, etc. Common to this evolution is the fact that new programming models provide mechanisms and notions that

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #CCR-0225610 (ITR), #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs)), the U.S. Army Research Office (ARO #W911NF-07-2-0019), the U.S. Air Force Office of Scientific Research (MURI #FA9550-06-0312 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC) and the following companies: Bosch, National Instruments, Thales, and Toyota.
Authors' addresses: S. Tripakis, D. Bui, and E. A. Lee, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley; emails: {stavros, daib, eal}@eecs.berkeley.edu; M. Geilen, Department of Electrical Engineering, Eindhoven University of Technology; email: m.c.w.geilen@tue.nl; B. Rodiers, LMS International; email: bert.rodiers@gmail.com.

DOI: http://dx.doi.org/10.1145/2442116.2442133

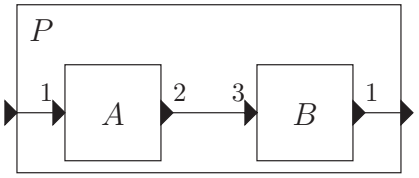

Fig. 1. Example of a hierarchical SDF graph.

are more *abstract*, that is, remote from the actual implementation, but better suited to the programmer's intuition. Raising the level of abstraction results in undeniable benefits in productivity. But it is more than just building systems faster or cheaper. It also allows to create systems that could not have been conceived otherwise, simply because of too high complexity.

Modeling languages with built-in concepts of concurrency, time, I/O interaction, and so on, are particularly suitable in the domain of embedded systems. Indeed, languages such as Simulink, UML, or SystemC, and corresponding tools, are particularly popular in this domain, for various applications. The tools provide mostly modeling and simulation, but often also code generation and static analysis or verification capabilities, which are increasingly important in an industrial setting. We believe that this tendency will continue, to the point where modeling languages of today will become the programming languages of tomorrow, at least in the embedded software domain.

A widespread model of computation in this domain is Synchronous (or Static) Data Flow (SDF) [Lee and Messerschmitt 1987]. SDF is particularly well-suited for signal processing and multimedia applications and has been extensively studied over the years (e.g., see [Bhattacharyya et al. 1996; Sriram and Bhattacharyya 2009]). Recently, languages based on SDF, such as StreamIt [Thies et al. 2002], have also been applied to multicore programming.

In this paper we consider hierarchical SDF models, where an SDF graph can be encapsulated into a composite SDF actor. The latter can then be connected with other SDF actors, further encapsulated, and so on, to form a hierarchy of SDF actors of arbitrary depth. This is essential for *compositional modeling*, which allows designing systems in a modular, scalable way, enhancing readability and allowing mastery of complexity in order to build larger designs. Hierarchical SDF models are part of a number of modeling environments, including the Ptolemy II framework [Eker et al. 2003]. A hierarchical SDF model is shown in Figure 1.

The problem we solve in this paper is *modular code generation* for hierarchical SDF models. Modular means that code is generated for a given composite SDF actor P independently from context, that is, independently from which graphs P is going to be used in. Moreover, once code is generated for P, then P can be seen as an *atomic* (non-composite) actor, that is, a "black box" without access to its internal structure. Modular code generation is analogous to separate compilation, which is available in most standard programming languages: the fact that one does not need to compile an entire program in one shot, but can compile files, classes, or other units, separately, and then combine them (e.g., by *linking*) to a single executable. This is obviously a key capability for a number of reasons, ranging from incremental compilation (compiling only the parts of a large program that have changed), to dealing with IP (intellectual property) concerns (having access to object code only and not to source code). We want to do the same for SDF models. Moreover, in the context of a system like Ptolemy II, in addition to the benefits mentioned above, modular code generation is also useful for speeding-up simulation: replacing entire sub-trees of a large hierarchical model

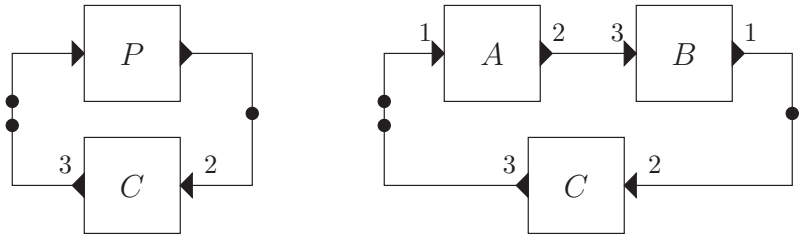

Fig. 2. Left: using the composite actor P of Figure 1 in an SDF graph with feedback and initial tokens. Right: the same graph after flattening P.

by a single actor for which code has been automatically generated and pre-compiled removes the overhead of executing all actors in the sub-tree individually.

Modular code generation is not trivial for hierarchical SDF models because they are not compositional. Let us try to give the intuition of this fact here through an example. A more detailed description is provided in the sections that follow. Consider the left-most graph of Figure 2, where the composite actor P of Figure 1 is used. This left-most graph should be equivalent to the right-most graph of Figure 2, where P has been replaced by its internal contents (i.e., the right-most graph is the "flattened" version of the left-most one). Observe that the right-most graph has no deadlock: indeed, actors A, B, C can fire infinitely often according to the periodic schedule $(A, A, B, C, A, B)^{\omega}$. Now, suppose we treat P as an atomic actor that consumes 3 tokens and produces 2 tokens every time it fires: this makes sense, since it corresponds to a complete iteration of its internal SDF graph, namely, (A, A, B, A, B). We then find that the left-most graph has a deadlock: P cannot fire because it needs 3 tokens but only 2 are initially available in the queue from C to P; C cannot fire either because it needs 2 tokens but only 1 is initially available in the queue from P to C.

The above example illustrates that composite SDF actors cannot be represented by atomic SDF actors without loss of information that can lead to deadlocks. Even in the case of acyclic SDF graphs, problems may still arise due to rate inconsistencies. Compositionality problems also arise in simpler hierarchical models such as *synchronous block diagrams* (SBDs) which (in the absence of triggers) can be seen as the subclass of *homogeneous* SDF where token rates are all equal [Lublinerman and Tripakis 2008b, 2008a; Lublinerman et al. 2009]. Our work extends the ideas of modular code generation for SBDs introduced in the above works. In particular, we borrow their notion of *profile* which characterizes a given actor. Modular code generation then essentially becomes a *profile synthesis* problem: how to synthesize a profile for composite actors, based on the profiles of its internal actors.

In SBDs, profiles are essentially DAGs (directed acyclic graphs) that capture the dependencies between inputs and outputs of a block, at the same synchronous round. In general, not all outputs depend on all inputs, which allows feedback loops with unambiguous semantics to be built. For instance, in a *unit delay* block the output does not depend on the input at the same clock cycle, therefore this block "breaks" dependency cycles when used in feedback loops.

The question is, what is the right model for profiles of SDF graphs. We answer this question in this paper. For SDF graphs, profiles turn out to be more interesting than simple DAGs. SDF profiles are essentially SDF graphs themselves, but with the ability to associate multiple producers and/or consumers with a single FIFO queue. Sharing queues among different actors generally results in non-deterministic models. In our case, however, we can guarantee that actors that share queues are always fired in a deterministic order. We call this model *deterministic SDF with shared FIFOs* (DSSF).

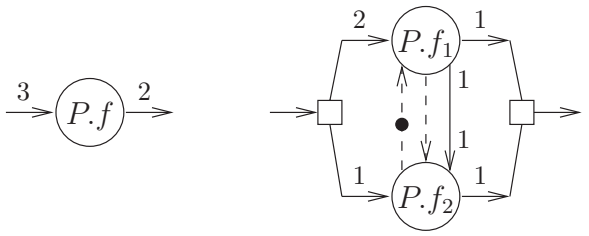

Fig. 3. Two DSSF graphs that are also profiles for the composite actor P of Figure 1.

DSSF allows for decomposing the firing of a composite actor into an arbitrary number of *firing functions* that may consume tokens from the same input port or produce tokens to the same output port. Having multiple firing functions allows decoupling firings of different internal actors of the composite actor, so that deadlocks are avoided when the composite actor is embedded in a given context. Our method guarantees *maximal reusability* [Lublinerman and Tripakis 2008b], i.e., the absence of deadlock in any context where the corresponding "flat" (non-hierarchical) SDF graph is deadlock-free, as well as consistency in any context where the flat SDF graph is consistent.

For example, two possible profiles for the composite actor P of Figure 1 are shown in Figure 3. The left-most one is a standard SDF graph with a single SDF actor representing a single firing function. This firing function corresponds to the internal sequence of firings (A, A, B, A, B). Such *monolithic* profiles are problematic as explained above. The right-most profile in Figure 1 is a DSSF graph with two actors, $P.f_1$ and $P.f_2$, representing two firing functions: the first corresponds to the sequence of firings (A, A, B), while the second corresponds to the sequence of firings (A, B). The two firing functions share the external input and output FIFO queues, depicted as small squares in the figure. Dependency edges depicted as dashed lines ensure a deterministic order of firing these functions (a detailed explanation is provided in the main body of the paper). This *non-monolithic* profile is maximally-reusable, and also optimal in the sense that no less than two firing functions can achieve maximal reusability.

We show how to perform profile synthesis for SDF graphs automatically. This means to synthesize for a given composite actor a profile, in the form of a DSSF graph, given the profiles of its internal actors (also DSSF graphs). This process involves multiple steps, among which are the standard *rate analysis* and *deadlock detection* procedures used to check whether a given SDF graph can be executed infinitely often without deadlock and with bounded queues [Lee and Messerschmitt 1987]. In addition to these steps, SDF profile synthesis involves *unfolding* a DSSF graph (i.e., replicating actors in the graph according to their relative rates produced by rate analysis) to produce a DAG that captures the dependencies between the different consumptions and productions of tokens at the same port.

Reducing the DSSF graph to a DAG is interesting because it allows to apply for our purposes the idea of *DAG clustering* proposed originally for SBDs [Lublinerman and Tripakis 2008b; Lublinerman et al. 2009]. As in the SBD case, we use DAG clustering in order to group together firing functions of internal actors and synthesize a small (hopefully minimal) number of firing functions for the composite actor. These determine precisely the profile of the latter. Keeping the number of firing functions small is essential, because it results in further compositions of the actor being more efficient, thus allowing the process to scale to arbitrary levels of hierarchy.

As shown by Lublinerman and Tripakis [2008b] and Lublinerman et al. [2009] there exist different ways to perform DAG clustering, that achieve different trade-offs, in particular in terms of number of clusters produced vs. reusability of the generated

profile. Among the clustering methods proposed for SBDs, of particular interest to us are those that produce *disjoint* clusterings, where clusters do not share nodes. Unfortunately, *optimal disjoint clustering*, that guarantees maximal reusability with a minimal number of clusters, is NP-complete [Lublinerman et al. 2009]. This motivates us to devise a new clustering algorithm, called *greedy backward disjoint clustering* (GBDC). GBDC guarantees maximal reusability but due to its greedy nature cannot guarantee optimality in terms of number of clusters. On the other hand, GBDC has polynomial complexity.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces DSSF graphs and SDF as a subclass of DSSF. Section 4 reviews analysis methods for SDF graphs. Section 5 reviews modular code generation for SBDs which we build upon. Section 6 introduces SDF profiles. Section 7 describes the profile synthesis procedure. Section 8 details DAG clustering, in particular, the GBDC algorithm. Section 9 presents a prototype implementation. Section 10 presents the conclusions and discusses future work.

2. RELATED WORK

Dataflow models of computation have been extensively studied in the literature. Dataflow models with deterministic actors, such as Kahn Process Networks [Kahn 1974] and their various subclasses, including SDF, are compositional at the semantic level. Indeed, actors can be given semantics as continuous functions on streams, and such functions are closed by composition. (Interestingly, it is much harder to derive a compositional theory of *non-deterministic* dataflow, e.g., see [Brock and Ackerman 1981; Jonsson 1994; Stark 1995].) Our work is at a different, non-semantical level, since we mainly focus on finite representations of the behavior of networks at their interfaces, in particular of the dependencies between inputs and output. We also take a "black-box" view of atomic actors, assuming their internal semantics (e.g., which function they compute) are unknown and unimportant for our purpose of code generation. Finally, we only deal with the particular subclass of SDF models.

Despite extensive work on code generation from SDF models and especially scheduling (e.g., see [Bhattacharyya et al. 1996; Sriram and Bhattacharyya 2009]), there is little existing work that addresses compositional representations and modular code generation for such models. Geilen [2009] proposes abstraction methods that reduce the size of SDF graphs, thus facilitating throughput and latency analysis. His goal is to have a conservative abstraction in terms of these performance metrics, whereas our goal here is to preserve input-output dependencies to avoid deadlocks during further composition.

Non-compositionality of SDF due to potential deadlocks has been observed in earlier works such as Pino et al. [1995], where the objective is to schedule SDF graphs on multiple processors. This is done by partitioning the SDF graph into multiple sub-graphs, each of which is scheduled on a single processor. This partitioning (also called *clustering*, but different from DAG clustering that we use in this paper, see below) may result in deadlocks, and the so-called "SDF composition theorem" [Pino et al. 1995] provides a sufficient condition so that no deadlock is introduced.

More recently, Falk et al. [2008] also identify the problem of non-compositionality and propose *Cluster Finite State Machines* (CFSMs) as a representation of composite SDF. They show how to compute a CFSM for a composite SDF actor that contains standard, atomic, SDF sub-actors, however, they do not show how a CFSM can be computed when the sub-actors are themselves represented as CFSMs. This indicates that this approach may not generalize to more than one level of hierarchy. Our approach works for arbitrary depths of hierarchy.

Another difference between the above work and ours is on the representation models, namely, CFSM vs. DSSF. CFSM is a state-machine model, where transitions are annotated with guards checking whether a sufficient number of tokens is available in certain input queues. DSSF, on the other hand, is a data flow model, only slightly more general than SDF. This allows re-using many of the techniques developed for standard SDF graphs, for instance, rate analysis and deadlock detection, with minimal adaptation.

The same remark applies to other automata-based formalisms, such as I/O automata [Lynch and Tuttle 1987], interface automata [de Alfaro and Henzinger 2001a], and so on. Such formalisms could perhaps be used to represent consumption and production actions of SDF graphs, resulting in compositional representations. These would be at a much lower level than DSSF, however, and for this reason would not admit SDF techniques such as rate analysis, which are more "symbolic."

To the extent that we propose DSSF profiles as interfaces for composite SDF graphs, our work is related to so-called *component-based design* and *interface theories* [de Alfaro and Henzinger 2001b]. Similarly to that line of research, we propose methods to synthesize interfaces for compositions of components, given interfaces for these components. We do not, however, include notions of refinement in our work. We are also not concerned with how to specify the "glue code" between components, as is done in *connector algebras* [Arbab 2005; Bliudze and Sifakis 2007]. Indeed, in our case, there is only one type of connection, namely, conceptually unbounded FIFOs, defined by the SDF semantics. Moreover, connections of components are themselves specified in the SDF graphs of composite actors, and are given as an input to the profile synthesis algorithm. Finally, we are not concerned with issues of timeliness or distribution, as in the work of Kopetz [1999].

Finally, we should emphasize that our DAG clustering algorithms solve a different problem than the clustering methods used in Pino et al. [1995], Falk et al. [2008], and other works in the SDF scheduling literature. Our clustering algorithms operate on plain DAGs, as do the clustering algorithms originally developed for SBDs [Lublinerman and Tripakis 2008b; Lublinerman et al. 2009]. On the other hand, Falk et al. [2008], Pino et al. [1995] perform clustering directly at the SDF level, by grouping SDF actors and replacing them by a single SDF actor (e.g., see Figure 4 of Falk et al. [2008]). This, in our terms, corresponds to monolithic clustering, which is not compositional.

3. HIERARCHICAL DSSF AND SDF GRAPHS

Deterministic SDF with shared FIFOs, or DSSF, is an extension of SDF in the sense that, whereas shared FIFOs (first-in, first-out queues) are explicitly prohibited in SDF graphs, they are allowed in DSSF graphs, provided determinism is ensured.

Syntactically, a DSSF graph consists of a set of nodes, called *actors*[1], a set of FIFO *queues*, a set of *external ports*, and a set of directed *edges*. Each actor has a set of *input ports* (possibly zero) and a set of *output ports* (possibly zero). An edge connects an output port of an actor to the input of a queue, or an output port of a queue to an input port of an actor. Actor ports can be connected to at most one queue.[2] An edge may also connect an external input port of the graph to the input of a queue, or the output of a queue to an external output port of the graph.

[1]It is useful to distinguish between actor *types* and actor *instances*. Indeed, an actor can be used in a given graph multiple times. For example, an actor of type *Adder*, that computes the arithmetic sum of its inputs, can be used multiple times in a given graph. In this case, we say that the Adder is *instantiated* multiple times. Each "copy" is an actor *instance*. In the rest of the paper, we often omit to distinguish between type and instance when we refer to an actor, when the meaning is clear from context.

[2]Implicit *fan-in* or *fan-out* is not allowed, however, it can be implemented explicitly, using actors. For example, an actor that consumes an input token and replicates to each of its output ports models fan-out.

Actors are either *atomic* or *composite*. A composite actor P encapsulates a DSSF graph, called the *internal* graph of P. The input and output ports of P are identified with the input and output external ports of its internal graph. Composite actors can themselves be encapsulated in new composite actors, thus forming a hierarchical model of arbitrary depth. A graph is *flat* if it contains only atomic actors, otherwise it is *hierarchical*. A *flattening* process can be applied to turn a hierarchical graph into a flat graph, by removing composite actors and replacing them with their internal graph, while making sure to re-institute any connections that would be otherwise lost.

Each port of an atomic actor has an associated *token rate*, a positive integer number, which specifies how many tokens are consumed from or produced to the port every time the actor fires. Composite actors do not have token rate annotations on their ports. They inherit this information from their internal actors, as we will explain in this paper.

A queue can be connected to more than one ports, at its input or output. When this occurs we say that the queue is *shared*, otherwise it is *non-shared*. An SDF graph is a DSSF graph where all queues are non-shared. Actors connected to the input of a queue are the *producers* of the queue, and actors connected to its output are its *consumers*. A queue stores *tokens* added by producers and removed by consumers when these actors *fire*. An atomic actor can fire when each of its input ports is connected to a queue that has enough tokens, i.e., more tokens than specified by the token rate of the port. Firing is an atomic action, and consists in removing the appropriate number of tokens from every input queue and adding the appropriate number of tokens to every output queue of the actor. Queues may store a number of *initial* tokens. Queues are of unbounded size in principle. In practice, however, we are interested in graphs that can execute forever using bounded queues.

To see why having shared queues generally results in non-deterministic models, consider two producers A_1, A_2 sharing the same output queue, and a consumer B reading from that queue and producing an external output. Depending on the order of execution of A_1 and A_2, their outputs will be stored in the shared queue in a different order. Therefore, the output of B will also generally differ (note that tokens may carry values).

To guarantee determinism, it suffices to ensure that A_1 and A_2 are always executed in a fixed order. This is the condition we impose on DSSF graphs, namely, that if a queue is shared among a set of producers $A_1, \ldots, A_a$ and a set of consumers $B_1, \ldots, B_b$, then the graph ensures, by means of its connections, a deterministic way of firing $A_1, \ldots, A_a$, as well as a deterministic way of firing $B_1, \ldots, B_b$. In other words, for any two valid executions of the graph, the order of firings of actors $A_1, \ldots, A_a$ will be the same, and similarly for $B_1, \ldots, B_b$. Notice that this condition does not imply that the order of firing of all actors will be the same across executions. Indeed, for actors that do not write to or read from the same queue, multiple firing orders are possible.

Let us provide some examples of SDF and DSSF graphs. A hierarchical SDF graph is shown in Figure 1. P is a composite actor while A and B are atomic actors. Every time it fires, A consumes one token from its single input port and produces two tokens to its single output port. B consumes three tokens and produces one token every time it fires. There are several non-shared queues in this graph: a queue with producer A and consumer B; a queue connecting the input port of P to the input port of its internal actor A; and a queue connecting the output port of internal actor B to the output port of P. Non-shared queues are identified with corresponding directed edges.

Two other SDF graphs are shown in Figure 2, which also shows an example of flattening. The left-most graph is hierarchical, since it contains composite actor P. By flattening this graph we obtain the right-most graph. These graphs contain queues with initial tokens, depicted as black dots. The queue connecting the output port of C

to the input port of P has two initial tokens. Likewise, there is one initial token in the queue from P to C.

Figure 3 shows two more examples of DSSF graphs. Both graphs are flat. Actors in these graphs are drawn as circles instead of squares because, as we shall see in Section 6, these graphs are also SDF profiles. External ports are depicted by arrows. The left-most graph is an SDF graph with a single actor $P.f$. The right-most one is a DSSF graph with two actors, $P.f_1$ and $P.f_2$. This graph has two shared queues, depicted as small squares. The two queues are connected to the two external ports of the graph. The graph also contains three non-shared queues, depicted implicitly by the dashed-line and solid-line edges connecting $P.f_1$ and $P.f_2$. Dashed-line edges are called *dependency edges* and are distinguished from solid-line edges that are "true" *dataflow edges*. The distinction is made only for reasons of clarity, in order to understand better the way edges are created during profile generation (Section 7.6). Otherwise the distinction plays no role, and dependency edges can be encoded as standard dataflow edges with token production and consumption rates both equal to 1. Notice, first, that the dataflow edge ensures that $P.f_2$ cannot fire before $P.f_1$ fires for the first time; and second, that the dependency edge from $P.f_2$ to $P.f_1$ ensures that $P.f_1$ can fire for a second time only *after* the first firing of $P.f_2$. Together these edges impose a total order on the firing of these two actors, therefore fulfilling the DSSF requirement of determinism.

DSSF graphs can be *open* or *closed*. A graph is closed if all its input ports are connected; otherwise it is open. The graph of Figure 1 is open because the input port of P is not connected. The graphs of Figure 3 are also open. The graphs of Figure 2 are closed.

4. ANALYSIS OF SDF GRAPHS

The SDF analysis methods proposed by Lee and Messerschmitt [1987] allow to check whether a given SDF graph has a *periodic admissible sequential schedule* (PASS). Existence of a PASS guarantees two things: first, that the actors in the graph can fire infinitely often without *deadlock*; and second, that only *bounded queues* are required to store intermediate tokens produced during the firing of these actors. We review these analysis methods here, because we are going to adapt them and use them for modular code generation (Section 7).

4.1. Rate Analysis

Rate analysis seeks to determine if the token rates in a given SDF graph are *consistent*: if this is not the case, then the graph cannot be executed infinitely often with bounded queues. We illustrate the analysis in the simple example of Figure 1. The reader is referred to Lee and Messerschmitt [1987] for the details.

We wish to analyze the internal graph of P, consisting of actors A and B. This is an open graph, and we can ignore the unconnected ports for the rate analysis. Suppose A is fired r_A times for every r_B times that B is fired. Then, in order for the queue between A and B to remain bounded in repeated execution, it has to be the case that

$$r_A \cdot 2 = r_B \cdot 3,$$

that is, the total number of tokens produced by A equals the total number of tokens consumed by B. The above *balance equation* has a non-trivial (i.e., non-zero) solution: $r_A = 3$ and $r_B = 2$. This means that this SDF graph is indeed consistent. In general, for larger and more complex graphs, the same analysis can be performed, which results in solving a system of multiple balance equations. If the system has a non-trivial solution then the graph is consistent, otherwise it is not. At the end of rate analysis, if consistent, a *repetition vector* $(r_1, \ldots, r_n)$ is produced that specifies the number r_i of times that every actor A_i in the graph fires with respect to other actors. This vector is

used in the subsequent step of deadlock analysis. Note that for disconnected graphs, the individual parts each have their own repetition vector and any linear combination of multiples of these repetition vectors is a PASS of the whole graph.

4.2. Deadlock Analysis

Having a consistent graph is a necessary, but not sufficient condition for infinite execution: the graph might still contain *deadlocks* that arise because of absence of enough initial tokens. Deadlock analysis ensures that this is not the case. An SDF graph is deadlock free if and only if every actor A can fire r_A times, where r_A is the repetition value for A in the repetition vector (i.e., it has a PASS [Lee and Messerschmitt 1987]). The method works as follows. For every queue e_i in the SDF graph, an integer counter b_i is maintained, representing the number of tokens in e_i. Counter b_i is initialized to the number of initial tokens present in e_i (zero if no such tokens are present). For every actor A in the SDF graph, an integer counter c_A is maintained, representing the number of remaining times that A should fire to complete the PASS. Counter c_A is initialized to r_A. A tuple consisting of all above counters is called a *configuration* v. A transition from a configuration v to a new configuration v' happens by firing an actor A, provided A is *enabled* at v, i.e., all its input queues have enough tokens, and provided that $c_A > 0$. Then, the queue counters are updated, and counter c_A is decremented by 1. If a configuration is reached where all actor counters are 0, there is no deadlock, otherwise, there is one. Notice that a single path needs to be explored, so this is not a costly method (i.e., not a full-blown reachability analysis). In fact, at most $\Sigma_{i=1}^{n} r_i$ steps are required to complete deadlock analysis, where $(r_1, \ldots, r_n)$ is the solution to the balance equations.

We illustrate deadlock analysis with an example. Consider the SDF graph shown at the left of Figure 2 and suppose P is an atomic actor, with input/output token rates 3 and 2, respectively. Rate analysis then gives $r_P = r_C = 1$. Let the queues from P to C and from C to P be denoted e_1 and e_2, respectively. Deadlock analysis then starts with configuration $v_0 = (c_P = 1, c_C = 1, b_1 = 1, b_2 = 2)$. P is not enabled at v_0 because it needs 3 input tokens but $b_2 = 2$. C is not enabled at v_0 either because it needs 2 input tokens but $b_1 = 1$. Thus v_0 is a deadlock. Now, suppose that instead of 2 initial tokens, queue e_2 had 3 initial tokens. Then, we would have as initial configuration $v_1 = (c_P = 1, c_C = 1, b_1 = 1, b_2 = 3)$. In this case, deadlock analysis can proceed: $v_1 \xrightarrow{P} (c_P = 0, c_C = 1, b_1 = 3, b_2 = 0) \xrightarrow{C} (c_P = 0, c_C = 0, b_1 = 1, b_2 = 3)$. Since a configuration is reached where $c_P = c_C = 0$, there is no deadlock.

4.3. Transformation of SDF to Homogeneous SDF

A *homogeneous* SDF (HSDF) graph is an SDF graph where all token rates are equal (and without loss in generality, can be assumed to be equal to 1). Any consistent SDF graph can be transformed to an equivalent HSDF graph using a type of an *unfolding* process consisting in replicating each actor in the SDF as many times as specified in the repetition vector [Lee and Messerschmitt 1987; Pino et al. 1995; Sriram and Bhattacharyya 2009]. The unfolding process subsequently allows to identify explicitly the input/output dependencies of different productions and consumptions at the same output or input port. Examples of unfolding are presented in Section 7.4, where we adapt the process to our purposes and to the case of DSSF graphs.

5. MODULAR CODE GENERATION FRAMEWORK

As mentioned in the introduction, our modular code generation framework for SDF builds upon the work of Lublinerman and Tripakis [2008b] and Lublinerman et al. [2009]. A fundamental element of the framework is the notion of *profiles*. Every SDF

actor has an associated profile. The profile can be seen as an *interface*, or *summary*, that captures the essential information about the actor. Atomic actors have predefined profiles. Profiles of composite actors are synthesized automatically, as shown in Section 7.

A profile contains, among other things, a set of *firing functions*, that, together, implement the firing of an actor. In the simple case, an actor may have a single firing function. For example, actors A, B of Figure 1 may each have a single firing function

```
A.fire(input x[1]) output (y[2]);
B.fire(input x[3]) output (y[1]);
```

The above signatures specify that `A.fire` takes as input 1 token at input port x and produces as output 2 tokens at output port y, and similarly for `B.fire`. In general, however, an actor may have more than one firing function in its profile. This is necessary in order to avoid *monolithic* code, and instead produce code that achieves *maximal reusability*, as is explained in Section 7.3.

The *implementation* of a profile contains, among other things, the implementation of each of the firing functions listed in the profile as a sequential program in a language such as C++ or Java. We will show how to automatically generate such implementations of SDF profiles in Section 7.7.

Modular code generation is then the following process: given a composite actor P, its internal graph, and profiles for every internal actor of P, synthesize automatically a profile for P and an implementation of this profile.

Note that a given actor may have multiple profiles, each achieving different trade-offs, for instance, in terms of *compactness* of the profile and *reusability* (ability to use the profile in as many contexts as possible). We illustrate such trade-offs in the sequel.

6. SDF PROFILES

We will use a special class of DSSF graphs to represent profiles of SDF actors, called *SDF profiles*. An SDF profile is a DSSF graph such that all its shared queues are connected to its external ports. Moreover, all connections between actors of the graph are such that the number of tokens produced and consumed at each firing by the source and destination actors are equal: this implies that connected actors fire with equal rates. The shared queues of SDF profiles are called *external*, because they are connected to external ports. This is to distinguish them from *internal* shared queues that may arise in other types of DSSF graphs that we use in this paper, in particular, in so-called *internal profiles graphs* (see Section 7.1).

Two SDF profiles are shown in Figure 3. They are two possible profiles for the composite actor P of Figure 1. We will see how these two profiles can be synthesized automatically in Section 7. We will also see that these two profiles have different properties. In particular, they represent different Pareto points in the compactness vs. reusability trade-off (Section 7.3).

The actors of an SDF profile represent firing functions. The left-most profile of Figure 3 contains a single actor $P.f$ corresponding to a single firing function `P.fire`. Profiles that contain a single firing function are called *monolithic*. The right-most profile of Figure 3 contains two actors, $P.f_1$ and $P.f_2$, corresponding to two firing functions, `P.fire1` and `P.fire2`: this is a *non-monolithic* profile. Notice that the dependency edge from $P.f_1$ to $P.f_2$ is redundant, since it is identical to the dataflow edge between the two. But the dataflow edge, in addition to a dependency, also encodes a transfer of data between the two firing functions.

Unless explicitly mentioned otherwise, in the examples that follow we assume that atomic blocks have monolithic profiles.

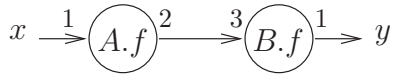

Fig. 4. Internal profiles graph of composite actor P of Figure 1.

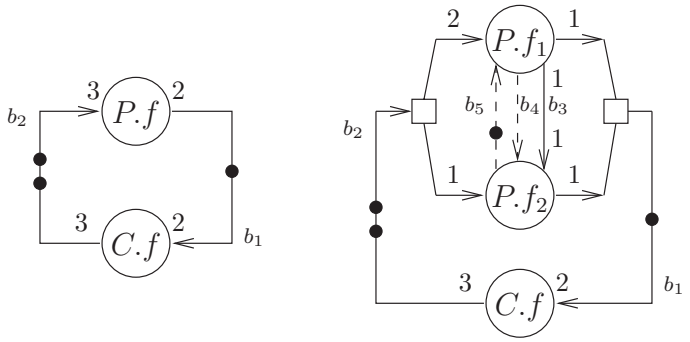

Fig. 5. Two internal profiles graphs, resulting from connecting the two profiles of actor P shown in Figure 3 and a monolithic profile of actor C, according to the graph at the left of Figure 2.

7. PROFILE SYNTHESIS AND CODE GENERATION

As mentioned above, modular code generation takes as input a composite actor P, its internal graph, and profiles for every internal actor of P, and produces as output a profile for P and an implementation of this profile. Profile synthesis refers to the computation of a profile for P, while code generation refers to the automatic generation of an implementation of this profile. These two functions require a number of steps, detailed below.

7.1. Connecting the SDF Profiles

The first step consists in connecting the SDF profiles of internal actors of P. This is done simply as dictated by the connections found in the internal graph of P. The result is a flat DSSF graph, called the *internal profiles graph* (IPG) of P. We illustrate this through an example. Consider the composite actor P shown in Figure 1. Suppose both its internal actors A and B have monolithic profiles, with $A.f$ and $B.f$ representing `A.fire` and `B.fire`, respectively. Then, by connecting these monolithic profiles we obtain the IPG shown in Figure 4. In this case, the IPG is an SDF graph.

Two more examples of IPGs are shown in Figure 5. There, we connect the profiles of internal actors P and C of the (closed) graph shown at the left of Figure 2. Actor C is assumed to have a monolithic profile. Actor P has two possible profiles, shown in Figure 3. The two resulting IPGs are shown in Figure 5. The left-most one is an SDF graph. The right-most one is a DSSF graph, with two internal shared queues and three non-shared queues.

7.2. Rate Analysis with SDF Profiles

This step is similar to the rate analysis process described in Section 4.1, except that it is performed on the IPG produced by the connection step, instead of an SDF graph. This presents no major challenges, however, and the method is essentially the same as the one proposed by Lee and Messerschmitt [1987].

Let us illustrate the process here, for the IPG shown to the right of Figure 5. We associate repetition variables r_p^1, r_p^2, and r_q, respectively, to $P.f_1$, $P.f_2$, and $C.f$. Then,

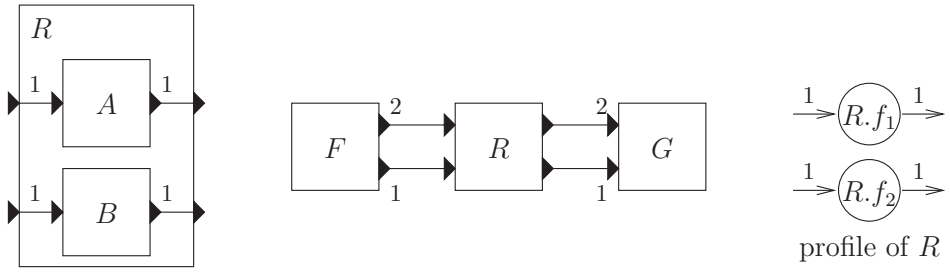

Fig. 6. Composite SDF actor R (left); using R (middle); non-monolithic profile of R (right).

we have the following balance equations.

$$\begin{aligned} r_p^1 \cdot 1 + r_p^2 \cdot 1 &= r_q \cdot 2, \\ r_q \cdot 3 &= r_p^1 \cdot 2 + r_p^2 \cdot 1, \\ r_p^1 \cdot 1 &= r_p^2 \cdot 1. \end{aligned}$$

As this has a non-trivial solution (e.g., $r_p^1 = r_p^2 = r_q = 1$), this graph is consistent, i.e., rate analysis succeeds in this example.

If the rate analysis step fails the graph is rejected. Otherwise, we proceed with the deadlock analysis step.

It is worth noting that rate analysis can sometimes succeed with non-monolithic profiles, whereas it would fail with a monolithic profile. An example is given in Figure 6. A composite actor R is shown to the left of the figure and its non-monolithic profile to the right. If we use R in the diagram shown to the middle of the figure, then rate analysis with the non-monolithic profile succeeds. It would fail, however, with the monolithic profile, since $R.f_2$ has to fire twice as often as $R.f_1$. This observation also explains why rate analysis must generally be performed on the IPG, and not on the internal SDF graph using monolithic profiles for internal actors.

7.3. Deadlock Analysis with SDF Profiles

Success of the rate analysis step is a necessary, but not sufficient, condition in order for a graph to have a PASS. Deadlock analysis is used to ensure that this is the case. Deadlock analysis is performed on the IPG produced by the connection step. It is done in the same way as the deadlock detection process described in Section 4.2. We illustrate this on the two examples of Figure 5.

Consider first the IPG to the left of Figure 5. There are two queues in this graph: a queue from $P.f$ to $C.f$, and a queue from $C.f$ to $P.f$. Denote the former by b_1 and the latter by b_2. Initially, b_1 has 1 token, whereas b_2 has 2 tokens. $P.f$ needs 3 tokens to fire but only 2 are available in b_2, thus $P.f$ cannot fire. $C.f$ needs 2 tokens but only 1 is available in b_1, thus $C.f$ cannot fire either. Therefore there is a deadlock already at the initial state, and this graph is rejected.

Now consider the IPG to the right of Figure 5. There are five queues in this graph: a queue from $P.f_1$ and $P.f_2$ to $C.f$, a queue from $C.f$ to $P.f_1$ and $P.f_2$, two queues from $P.f_1$ to $P.f_2$, and a queue from $P.f_2$ to $P.f_1$. Denote these queues by b_1, b_2, b_3, b_4, b_5, respectively. Initially, b_1 has 1 token, b_2 has 2 tokens, b_3 and b_4 are empty, and b_5 has 1 token. $P.f_1$ needs 2 tokens to fire and 2 tokens are indeed available in b_2, thus $P.f_1$

can fire and the initial state is not a deadlock. Deadlock analysis gives

$$\begin{aligned}
&(c_{p_1} = 1, c_{p_2} = 1, c_q = 1, b_1 = 1, b_2 = 2, b_3 = 0, b_4 = 0, b_5 = 1) \xrightarrow{P.f_1} \\
&(c_{p_1} = 0, c_{p_2} = 1, c_q = 1, b_1 = 2, b_2 = 0, b_3 = 1, b_4 = 1, b_5 = 0) \xrightarrow{C.f} \\
&(c_{p_1} = 0, c_{p_2} = 1, c_q = 0, b_1 = 0, b_2 = 3, b_3 = 1, b_4 = 1, b_5 = 0) \xrightarrow{P.f_2} \\
&(c_{p_1} = 0, c_{p_2} = 0, c_q = 0, b_1 = 1, b_2 = 2, b_3 = 0, b_4 = 0, b_5 = 1).
\end{aligned}$$

Therefore, deadlock analysis succeeds (no deadlock is detected).

This example illustrates the trade-off between compactness and reusability. For the same composite actor P, two profiles can be generated, as shown in Figure 3. These profiles achieve different trade-offs. The monolithic profile shown to the left of the figure is more compact (i.e., smaller) than the non-monolithic one shown to the right. The latter is more reusable than the monolithic one, however: indeed, it can be reused in the graph with feedback shown at the left of Figure 2, whereas the monolithic one cannot be used, because it creates a deadlock.

Note that if we flatten the graph as shown in Figure 2, that is, remove composite actor P and replace it with its internal graph of atomic actors A and B, then the resulting graph has a PASS, i.e., exhibits no deadlock. This shows that deadlock is a result of using the monolithic profile, and not a problem with the graph itself. Of course, flattening is not the solution, because it is not modular: it requires the internal graph of P to be known and used in every context where P is used. Thus, code for P cannot be generated independently from context.

If the deadlock analysis step fails then the graph is rejected. Otherwise, we proceed with the unfolding step.

7.4. Unfolding with SDF Profiles

This step takes as input the IPG produced by the connection step, as well as the repetition vector produced by the rate analysis step. It produces as output a DAG (directed acyclic graph) that captures the input-output dependencies of the IPG. As mentioned in Section 4.3 the unfolding step is an adaptation of existing transformations from SDF to HSDF.

The DAG is computed in two steps. First, the IPG is *unfolded*, by replicating each node in it as many times as specified in the repetition vector. These replicas represent the different firings of the corresponding actor. For this reason, the replicas are ordered: dependencies are added between them to represent the fact that the first firing comes before the second firing, the second before the third, and so on. Ports are also replicated. In particular, port replicas are created for ports connected to the same queue. This is the case for replicas x_i and y_i shown in the figure. Note that we do not consider these to be shared queues, precisely because we want to capture dependencies between each separate production and consumption of tokens at the same queue. Finally, for every internal queue of the IPG, a queue is created in the unfolded graph with the appropriate connections. The process is illustrated in Figure 7, for the IPG of Figure 4. Rate analysis in this case produces the repetition vector $(r_A = 3, r_B = 2)$. Therefore $A.f$ is replicated 3 times and $B.f$ is replicated 2 times. In this example there is a single internal queue between $A.f$ and $B.f$, and the unfolded graph contains a single shared queue.

In the second and final step of unfolding, the DAG is produced, by computing dependencies between the replicas. This is done by separately computing dependencies between replicas that are connected to a given queue, and repeating the process for every queue. We first explain the process for a non-shared queue such as the one between A and B in the IPG of Figure 4. Suppose that the queue has d initial tokens,

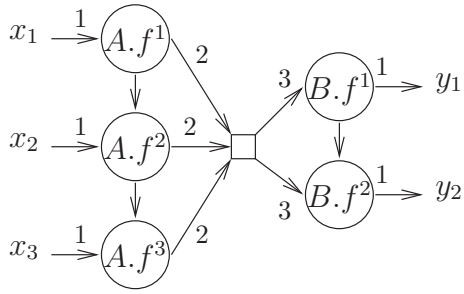

Fig. 7. First step of unfolding the IPG of Figure 4: replicating nodes and creating a shared queue.

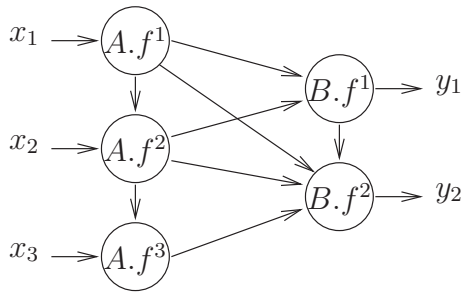

Fig. 8. Unfolding the IPG of Figure 4 produces the IODAG shown here.

its producer A adds k tokens to the queue each time it fires, and its consumer B removes n tokens each time it fires. Then the j-th occurrence of B depends on the i-th occurrence of A if and only if

$$d + (i-1) \cdot k < j \cdot n. \tag{1}$$

In that case, an edge from $A.f^i$ to $B.f^j$ is added to the DAG. For the example of Figure 4, this gives the DAG shown in Figure 8.

In the general case, a queue in the IPG of P may be shared by multiple producers and multiple consumers. Consider such a shared queue between a set of producers $A_1, \ldots, A_a$ and a set of consumers $B_1, \ldots, B_b$. Let k_h be the number of tokens produced by A_h, for $h = 1, \ldots, a$. Let n_h be the number of tokens consumed by B_h, for $h = 1, \ldots, b$. Let d be the number of initial tokens in the queue. By construction (see Section 7.6) there is a total order $A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_a$ on the producers and a total order $B_1 \rightarrow B_2 \rightarrow \cdots \rightarrow B_b$ on the consumers. As this is encoded with standard SDF edges of the form $A_i \overset{1}{\rightarrow} \overset{1}{} A_{i+1}$, this also implies that during rate analysis the rates of all producers will be found equal, and so will the rates of all consumers. Then, the j-th occurrence of B_u, $1 \leq u \leq b$, depends on the i-th occurrence of A_v, $1 \leq v \leq a$, if and only if:

$$d + (i-1) \cdot \sum_{h=1}^{a} k_h + \sum_{h=1}^{v-1} k_h < (j-1) \cdot \sum_{h=1}^{b} n_h + \sum_{h=1}^{u} n_h. \tag{2}$$

Notice that, as should be expected, Equation (2) reduces to Equation (1) in the case $a = b = 1$.

Another example of unfolding, starting with an IPG that contains a non-monolithic profile, is shown in Figure 9. P is the composite actor of Figure 1. Its non-monolithic right-most profile of Figure 3 is used to form the IPG shown in Figure 9.

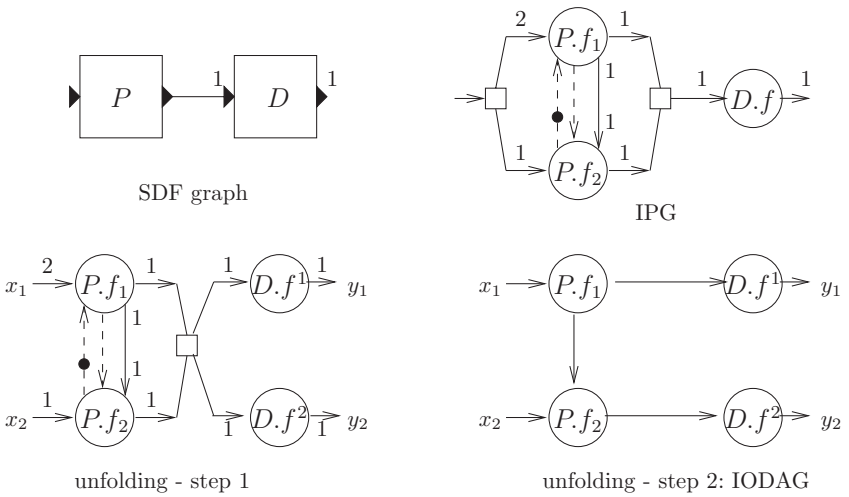

Fig. 9. Another example of unfolding.

In the DAG produced by unfolding, input and output port replicas such as x_i and y_i in Figures 8 and 9 are represented explicitly as special nodes with no predecessors and no successors, respectively. For this reason, we call this DAG an IODAG. Nodes of the IODAG that are neither input nor output are called *internal* nodes.

7.5. DAG Clustering

DAG clustering consists in partitioning the internal nodes of the IODAG produced by the unfolding step into a number of *clusters*. The clustering must be *valid* in the sense that it must not create cyclic dependencies among distinct clusters.[3] Note that a monolithic clustering is trivially valid, since it contains a single cluster. Each of the clusters in the produced clustered graph will result in a firing function in the profile of P, as explained in Section 7.6 that follows. Exactly how DAG clustering is done is discussed in Section 8. There are many possibilities, which explore different trade-offs, in terms of compactness, reusability, and other metrics. Here, we illustrate the *outcome* of DAG clustering on our running example. Two possible clusterings of the DAG of Figure 8 are shown in Figure 10, enclosed in dashed curves. The left-most clustering contains a single cluster, denoted C_0. The right-most clustering contains two clusters, denoted C_1 and C_2.

7.6. Profile Generation

Profile generation is the last step in profile synthesis, where the actual profile of composite actor P is produced. The clustered graph, together with the internal graph of P, completely determine the profile of P. Each cluster C_i is mapped to a firing function `fire`i, and also to an atomic node $P.f_i$ in the profile graph of P. For every input (resp. output) port of P, an external, potentially shared queue L is created in the profile of P. For each cluster C_i, we compute the total number of tokens k_i read from (resp. written to) L by C_i: this can be easily done by summing over all actors in C_i. If $k_i > 0$ then an edge is added from L to $P.f_i$ (resp. from $P.f_i$ to L) annotated with a rate of k_i tokens.

[3]A dependency between two distinct clusters exists iff there is a dependency between two nodes from each cluster.

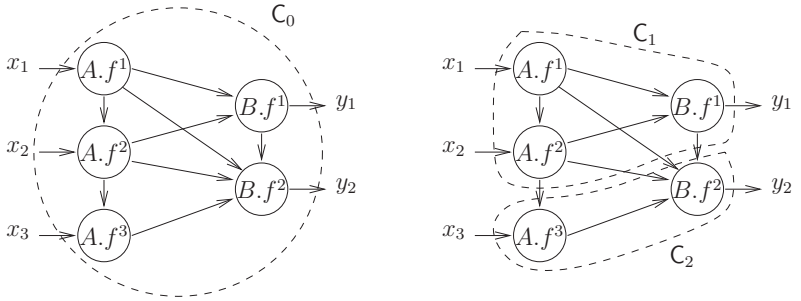

Fig. 10. Two possible clusterings of the DAG of Figure 8.

Dependency edges between firing functions are computed as follows. For every pair of distinct clusters C_i and C_j, a dependency edge from $P.f_i$ to $P.f_j$ is added iff there exist nodes v_i in C_i and v_j in C_j such that v_j depends on v_i in the IODAG. Validity of clustering ensures that this set of dependencies results in no cycles. In addition to these dependency edges, we add a set of *backward* dependency edges to ensure a deterministic order of writing to (resp. reading from) shared queues also across iterations. Let L be a shared queue of the profile. L is either an external queue, or an internal queue of P, like the one shown in Figure 7. Let W_L (resp. R_L) be the set of all clusters writing to (resp. reading from) L. By the fact that different replicas of the same actor that are created during unfolding are totally ordered in the IODAG, all clusters in W_L are totally ordered. Let $\mathsf{C}_i, \mathsf{C}_j \in W_L$ be the first and last clusters in W_L with respect to this total order, and let $P.f_i$ and $P.f_j$ be the corresponding firing functions. We encode the fact that the $P.f_i$ cannot re-fire before $P.f_j$ has fired, by adding a dependency edge from $P.f_j$ to $P.f_i$, supplied with an initial token. This is a backward dependency edge. Note that if $i = j$ then this edge is redundant. Similarly, we add a backward dependency edge, if necessary, among the clusters in R_L, which are also guaranteed to be totally ordered.

To establish the dataflow edges of the profile, we iterate over all internal (shared or non-shared) queues of the IPG of P. Let L be an internal queue and suppose it has d initial tokens. Let m be the total number of tokens produced at L by all clusters writing to L: by construction, m is equal to the total number of tokens consumed by all clusters reading from L. For our running example (Figures 4, 8, and 10), we have $d = 0$ and $m = 6$.

Conceptually, we define m output ports denoted $z_0, z_1, \ldots, z_{m-1}$, and m input ports, denoted $w_0, w_1, \ldots, w_{m-1}$. For $i = 0$ to $m - 1$, we connect output port z_i to input port w_j, where $j = (d + i) \div m$, and $\div$ is the modulo operator. Intuitively, this captures the fact that the i-th token produced will be consumed as the $((d + i) \div m)$-th token of some iteration, because of the initial tokens. Notice that $j = i$ when $d = 0$. We then place $\lfloor \frac{d+m-1-i}{m} \rfloor$ initial tokens at each input port w_i, for $i = 0, \ldots, m - 1$, where $\lfloor v \rfloor$ is the integer part of v. Thus, if $d = 4$ and $m = 3$, then w_0 will receive 2 initial tokens, while w_1 and w_2 will each receive 1 initial token.

Finally, we assign the ports to producer and consumer clusters of L, according to their total order. For instance, for the non-monolithic clustering of Figure 10, output ports z_0 to z_3 and input ports w_0 to w_2 are assigned to C_1, whereas output ports z_4, z_5 and input ports w_3 to w_5 are assigned to C_2. Together with the port connections, these assignments define dataflow edges between the clusters. Self-loops (edges with same source and destination cluster) without initial tokens are removed. Note that more than one edge

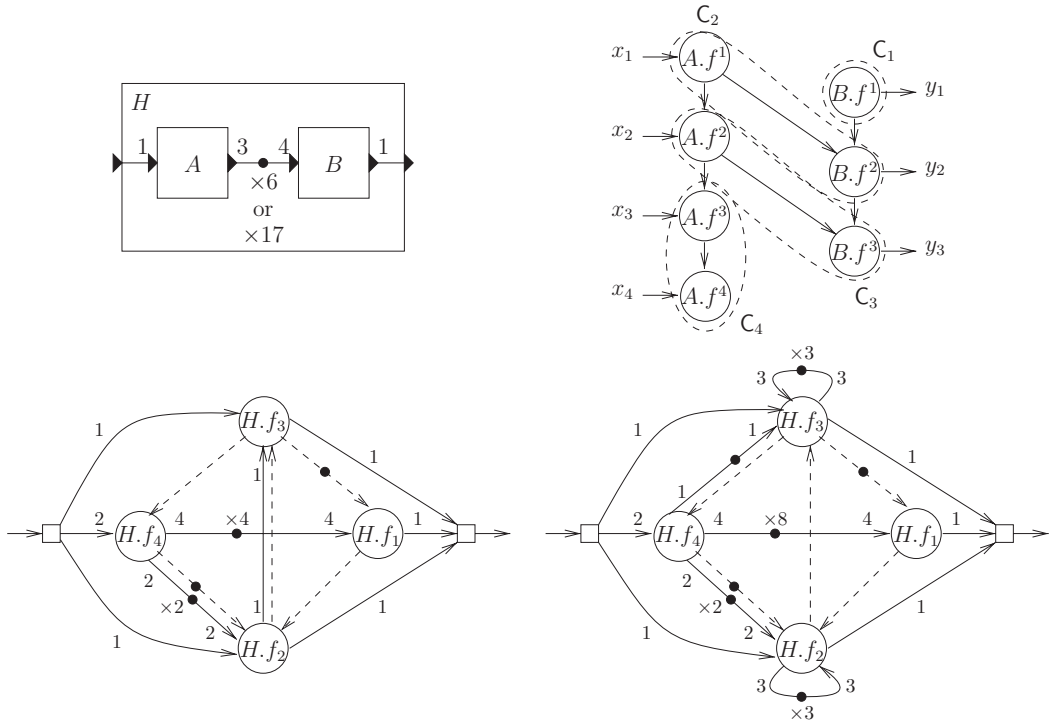

Fig. 11. Composite SDF actor H (top-left); possible clustering produced by unfolding (top-right); SDF profiles generated for H, assuming 6 initial tokens in the queue from A to B (bottom-left); assuming 17 initial tokens (bottom-right). Firing functions $H.f_1, H.f_2, H.f_3, H.f_4$ correspond to clusters $\mathsf{C}_1, \mathsf{C}_2, \mathsf{C}_3, \mathsf{C}_4$, respectively.

may exist between two distinct clusters, but these can always be merged into a single edge.

As an example, the two clusterings shown in Figure 10 give rise, respectively, to the two profiles shown in Figure 3. Another, self-contained example is shown in Figure 11. The two profiles shown at the bottom of the figure are generated from the clustering shown at the top-right, assuming 6 and 17 initial tokens in the queue from A to B, respectively. Notice that if the queue contains 17 tokens then this clustering is not optimal, in the sense that a more coarse-grain clustering exists. However, the clustering is valid, and used here to illustrate the profile generation process.

7.7. Code Generation

Once the profile has been synthesized, its firing functions need to be implemented. This is done in the code generation step. Every firing function corresponds to a cluster produced by the clustering step. The implementation of the firing function consists in calling in a sequential order all firing functions of internal actors that are included in the cluster. This sequential order can be arbitrary, provided it respects the dependencies of nodes in the cluster. We illustrate the process on our running example (Figures 1, 3, and 10).

Consider first the clustering shown to the left of Figure 10. This will result in a single firing function for P, namely, `P.fire`. Its implementation is shown below in pseudo-code.

```
P.fire(input x[3]) output y[2]
{
  local tmp[4];
  tmp <- A.fire(x);
  tmp <- A.fire(x);
  y   <- B.fire(tmp);
  tmp <- A.fire(x);
  y   <- B.fire(tmp);
}
```

In the above pseudo-code, `tmp` is a local FIFO queue of length 4. Such a local queue is assumed to be empty when initialized. A statement such as `tmp <- A.fire(x)` corresponds to a call to firing function `A.fire`, providing as input the queue `x` and as output the queue `tmp`. `A.fire` will consume 1 token from `x` and will produce 2 tokens into `tmp`. When all statements of `P.fire` are executed, 3 tokens are consumed from the input queue `x` and 2 tokens are added to the output queue `y`, as indicated in the signature of `P.fire`.

Now let us turn to the clustering shown to the right of Figure 10. This clustering contains two clusters, therefore, it results in two firing functions for P, namely, `P.fire1` and `P.fire2`. Their implementation is shown below.

```
persistent local tmp[N];  /* N is a parameter */
assumption: N >= 4;

P.fire1(input x[2])   P.fire2(input x[1], tmp[1])
output y[1], tmp[1]   output y[1]
{                     {
  tmp <- A.fire(x);     tmp <- A.fire(x);
  tmp <- A.fire(x);     y   <- B.fire(tmp);
  y   <- B.fire(tmp); }
}
```

In this case `tmp` is declared to be a *persistent* local variable, which means its contents "survive" across calls to `P.fire1` and `P.fire2`. In particular, of the 4 tokens produced and added to `tmp` by the two calls of `A.fire` within the execution of `P.fire1`, only the first 3 are consumed by the call to `B.fire`. The remaining 1 token is consumed during the execution of `P.fire2`. This is why `P.fire1` declares to produce at its output `tmp[1]` (which means it produces a total of 1 token at queue `tmp` when it executes), and similarly, `P.fire2` declares to consume at its input 1 token from `tmp`.

Dependency edges are not implemented in the code, since they carry no useful data, and only serve to encode dependencies between firing function calls. These dependencies must be satisfied by construction in any correct usage of the profile. Therefore they do not need to be enforced in the implementation of the profile.

7.8. Discussion: Inadequacy of Cyclo-Static Data Flow

It may seem that *cyclo-static data flow* (CSDF) [Bilsen et al. 1995] can be used as an alternative representation of profiles. Indeed, this works on our running example: we could capture the composite actor P of Figure 1 using the CSDF actor shown in Figure 12. This CSDF actor specifies that P will iterate between two firing modes. In the first mode, it consumes 2 tokens from its input and produces 1 token at its

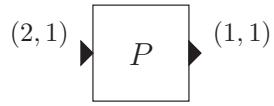

Fig. 12. CSDF actor for composite actor P of Figure 1.

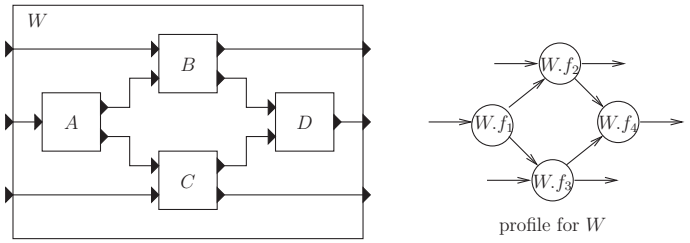

Fig. 13. An example that cannot be captured by CSDF.

output; in the second mode, it consumes 1 token and produces 1 token; the process is then repeated. This indeed works for this example: embedding P as shown in Figure 2 results in no deadlock, if the CSDF model for P is used.

In general, however, CSDF models are not expressive enough to be used as profiles. We illustrate this by two examples[4], shown in Figures 6 and 13. Actors R and W are two composite actors shown in these figures. All graphs are homogeneous (tokens rates are omitted in Figure 13, they are implicitly all equal to 1; dependency edges are also omitted from the profile, since they are redundant). Our method generates the profiles shown to the right of the two figures. The profile for R contains two completely independent firing functions, $R.f_1$ and $R.f_2$. CSDF cannot express this independence, since it requires a fixed order of firing modes to be specified statically. Although two separate CSDF models could be used to capture this example, this is not sufficient for composite actor W, which features both internal dependencies and independencies.

8. DAG CLUSTERING

DAG clustering is at the heart of our modular code generation framework, since it determines the profile that is to be generated for a given composite actor. As mentioned above, different trade-offs can be explored during DAG clustering, in particular, in terms of compactness and reusability. In general, the more *fine-grain* the clustering is, the more reusable the profile and generated code will be; the more *coarse-grain* the clustering is, the more compact the code is, but also less reusable. Note that there are cases where the most fine-grain clustering is wasteful since it is not necessary in order to achieve maximal reusability. Similarly, there are cases where the most coarse-grain clustering does not result in any loss of reusability. An instance of both cases is the trivial example where all outputs depend on all inputs, in which case the monolithic clustering is maximally reusable and also the most coarse possible. An extensive discussion of these and other trade-offs can be found in previous work on modular code generation for SBDs [Lublinerman and Tripakis 2008b, 2008a; Lublinerman et al. 2009]. The same principles apply also to SDF models.

[4]Falk et al. [2008] also observe that CSDF is not a sufficient abstraction of composite SDF models, however, the example they use embeds a composite SDF graph into a dynamic data flow model. Therefore the overall model is not strictly SDF. The examples we provide are much simpler, in fact, the models are homogeneous SDF models.

DAG clustering takes as input the IODAG produced by the unfolding step. A trivial way to perform DAG clustering is to produce a single cluster, which groups together all internal nodes in this DAG. This is called *monolithic* DAG clustering and results in monolithic profiles that have a single firing function. This clustering achieves maximal compactness, but it results in non-reusable code in general, as the discussion of Section 7.3 demonstrates.

In this section we describe clustering methods that achieve *maximal reusability*. This means that the generated profile (and code) can be used in *any* context where the corresponding flattened graph could be used. Therefore, the profile results in no information loss as far as reusing the graph in a given context is concerned. At the same time, the profile may be much smaller than the internal graph.

To achieve maximal reusability, we follow the ideas proposed by Lublinerman and Tripakis [2008b] and Lublinerman et al. [2009] for SBDs. In particular, we present a clustering method that is guaranteed not to introduce *false input-output dependencies*. These dependencies are "false" in the sense that they are not induced by the original SDF graph, but only by the clustering method.

To illustrate this, consider the monolithic clustering shown to the left of Figure 10. This clustering introduces a false input-output dependency between the third token consumed at input x (represented by node x_3 in the DAG) and the first token produced at output y (represented by node y_1). Indeed, in order to produce the first token at output y, only 2 tokens at input x are needed: these tokens are consumed respectively by the first two invocations of `A.fire`. The third invocation of `A.fire` is only necessary in order to produce the *second* token at y, but not the first one. The monolithic clustering shown to the left of Figure 10 loses this information. As a result, it produces a profile which is not reusable in the context of Figure 2, as demonstrated in Section 7.3. On the other hand, the non-monolithic clustering shown to the right of Figure 10 preserves the input-output dependency information, that is, does not introduce false dependencies. Because of this, it results in a maximally reusable profile.

The above discussion also helps to explain the reason for the unfolding step. Unfolding makes explicit the dependencies between different productions and consumptions of tokens *at the same ports*. In the example of actor P (Figure 4), even though there is a single external input port x and a single external output port y in the IPG, there are three copies of x and two copies of y in the unfolded DAG, corresponding to the three consumptions from x and two productions to y that occur within a PASS.

Unfolding is also important because it allows us to re-use the clustering techniques proposed for SBDs, which work on plain DAGs [Lublinerman and Tripakis 2008b; Lublinerman et al. 2009]. In particular, we can use the so-called *optimal disjoint clustering* (ODC) method which is guaranteed not to introduce false IO dependencies, produces a set of pairwise *disjoint* clusters (clusters that do not share any nodes), and is *optimal* in the sense that it produces a minimal number of clusters with the above properties. Unfortunately, the ODC problem is NP-complete [Lublinerman et al. 2009]. This motivated us to develop a "greedy" DAG clustering algorithm, which is one of the contributions of this paper. Our algorithm is not optimal, i.e., it may produce more clusters than needed to achieve maximal reusability. On the other hand, the algorithm has polynomial complexity. The greedy DAG clustering algorithm that we present below is "backward" in the sense that it proceeds from outputs to inputs. A similar "forward" algorithm can be used, that proceeds from inputs to outputs.

8.1. Greedy Backward Disjoint Clustering

The greedy backward disjoint clustering (GBDC) algorithm is shown in Figure 14. GBDC takes as input an IODAG (the result of the unfolding step) $G = (V, E)$ where V is a finite set of nodes and E is a set of directed edges. V is partitioned in three disjoint

Input: An IODAG $G = (V, E)$. $V = V_{\text{in}} \cup V_{\text{out}} \cup V_{\text{int}}$.
Output: A partition $\mathcal{C}$ of the set of internal nodes V_{int}.

```
1  foreach v ∈ V do
2  |  compute ins(v) and outs(v);
3  end
4  C := ∅;
5  Out := {f ∈ V_int | ∃y ∈ V_out : (f, y) ∈ E};
6  while ⋃C ≠ V_int do
7  |  partition Out into C_1, ..., C_k such that two nodes f, f' are grouped in the same set C_i iff
   |  ins(f) = ins(f');
8  |  C := C ∪ {C_1, ..., C_k};
9  |  for i = 1 to k do
10 |  |  while ∃f ∈ C_i, f' ∈ V_int \ ⋃C : (f', f) ∈ E ∧ ∀x ∈ ins(C_i), y ∈ outs(f') : (x, y) ∈ E* do
11 |  |  |  C_i := C_i ∪ {f'};
12 |  |  end
13 |  end
14 |  Out := {f ∈ V_int \ ⋃C | ¬∃f' ∈ V_int \ ⋃C : (f, f') ∈ E};
15 end
16 while quotient graph G_C contains cycles do
17 |  pick a cycle C_1→C_2→ ··· →C_k→C_1;
18 |  C := (C \ {C_1, ..., C_k}) ∪ {⋃_{i=1}^{k} C_i};
19 end
```

Fig. 14. The GBDC algorithm.

sets: $V = V_{\text{in}} \cup V_{\text{out}} \cup V_{\text{int}}$, the sets of input, output, and internal nodes, respectively. GBDC returns a partition of V_{int} into a set of disjoint sets, called clusters. The partition (i.e., the set of clusters) is denoted $\mathcal{C}$. The problem is non-trivial when all V_{in}, V_{out}, and V_{int} are non-empty (otherwise a single cluster suffices). In the sequel, we assume that this is the case.

$\mathcal{C}$ defines a new graph, called the *quotient graph*, $G_{\mathcal{C}} = (V_{\mathcal{C}}, E_{\mathcal{C}})$. $G_{\mathcal{C}}$ contains clusters instead of internal nodes, and has an edge between two clusters (or a cluster and an input or output node) if the clusters contain nodes that have an edge in the original graph G. Formally, $V_{\mathcal{C}} = V_{\text{in}} \cup V_{\text{out}} \cup \mathcal{C}$, and $E_{\mathcal{C}} = \{(x, \mathsf{C}) \mid x \in V_{\text{in}}, \mathsf{C} \in \mathcal{C}, \exists f \in \mathsf{C} : (x, f) \in E\} \cup \{(\mathsf{C}, y) \mid \mathsf{C} \in \mathcal{C}, y \in V_{\text{out}}, \exists f \in \mathcal{C} : (f, y) \in E\} \cup \{(\mathsf{C}, \mathsf{C}') \mid \mathsf{C}, \mathsf{C}' \in \mathcal{C}, \mathsf{C} \neq \mathsf{C}', \exists f \in \mathsf{C}, f' \in \mathsf{C}', (f, f') \in E\}$. Notice that $E_{\mathcal{C}}$ does not contain self-loops (i.e., edges of the form (f, f)).

The steps of GBDC are explained below. E^* denotes the transitive closure of relation E: $(v, v') \in E$ iff there exists a path from v to v', i.e., v' depends on v.

Identify input-output dependencies. Given a node $v \in V$, let $\mathsf{ins}(v)$ be the set of input nodes that v depends upon: $\mathsf{ins}(v) := \{x \in V_{\text{in}} \mid (x, v) \in E^*\}$. Similarly, let $\mathsf{outs}(v)$ be the set of output nodes that depend on v: $\mathsf{outs}(v) := \{y \in V_{\text{out}} \mid (v, y) \in E^*\}$. For a set of nodes F, $\mathsf{ins}(F)$ denotes $\bigcup_{f \in F} \mathsf{ins}(f)$, and similarly for $\mathsf{outs}(F)$.

Lines 1–3 of GBDC compute $\mathsf{ins}(v)$ and $\mathsf{outs}(v)$ for every node v of the DAG. We can compute these by starting from the output and following the dependencies backward. For example, consider the DAG of Figure 8. There are three input nodes, x_1, x_2, x_3 and two output nodes, y_1 and y_2. We have: $\mathsf{ins}(y_1) = \mathsf{ins}(B.f^1) = \mathsf{ins}(A.f^2) = \{x_1, x_2\}$, and $\mathsf{ins}(y_2) = \mathsf{ins}(B.f^2) = \{x_1, x_2, x_3\}$. Similarly: $\mathsf{outs}(x_1) = \mathsf{outs}(A.f^1) = \{y_1, y_2\}$, and $\mathsf{outs}(x_3) = \mathsf{outs}(A.f^3) = \{y_2\}$.

Line 4 of GBDC initializes $\mathcal{C}$ to the empty set. Line 5 initializes Out as the set of internal nodes that have an output y as an immediate successor. These nodes will be used as "seeds" for creating new clusters (Lines 7–8).

Then the algorithm enters the while-loop at Line 6. $\bigcup \mathcal{C}$ is the union of all sets in $\mathcal{C}$, i.e., the set of all nodes clustered so far. When $\bigcup \mathcal{C} = V_{\text{int}}$ all internal nodes have been added to some cluster, and the loop exits. The body of the loop consists in the following steps.

Partition seed nodes with respect to input dependencies. Line 7 partitions Out into a set of clusters, such that two nodes are put into the same cluster iff they depend on the same inputs. Line 8 adds these newly created clusters to $\mathcal{C}$. In the example of Figure 8, this gives an initial $\mathcal{C} = \{\{B.f^1\}, \{B.f^2\}\}$.

Create a cluster for each group of seed nodes. The for-loop starting at Line 9 iterates over all clusters newly created in the previous step and attempts to add as many nodes as possible to each of these clusters, going backward, and making sure no false input-output dependencies are created in the process. In particular, for each cluster C_i, we proceed backward, attempting to add unclustered predecessors f' of nodes f already in C_i (while-loop at Line 10). Such a node f' is a candidate to be added to C_i, but this happens only if an additional condition is satisfied: namely $\forall x \in \mathsf{ins}(\mathsf{C}_i), y \in \mathsf{outs}(f') : (x, y) \in E^*$. This condition is violated if there exists an input node x that some node in C_i depends upon, and an output node y that depends on f' but not on x. In that case, adding f' to C_i would create a false dependency from x to y. Otherwise, it is safe to add f', and this is done in Line 11.

In the example of Figure 8, executing the while-loop at Line 10 results in adding nodes $A.f^1$ and $A.f^2$ in the cluster $\{B.f^1\}$, and node $A.f^3$ in the cluster $\{B.f^2\}$, thereby obtaining the final clustering, shown to the right of Figure 10.

In general, more than one iteration may be required to cluster all the nodes. This is done by repeating the process, starting with a new Out set. In particular, Line 14 recomputes Out as the set of all unclustered nodes that have no unclustered successors.

Removing cycles. The above process is not guaranteed to produce an acyclic quotient graph. Lines 16–19 remove cycles by repeatedly *merging* all clusters in a cycle into a single cluster. This process is guaranteed not to introduce false input-output dependencies, as shown in Lemma 5 of Lublinerman et al. [2009].

THEOREM 8.1. *Provided the set of nodes V is finite, GBDC always terminates.*

PROOF. G is acyclic, therefore the set Out computed in Lines 5 and 14 is guaranteed to be non-empty. Therefore, at least one new cluster is added at every iteration of the while-loop at Line 6, which means the number of unclustered nodes decreases at every iteration. The for-loop and foreach-loop inside this while-loop obviously terminate, therefore, the body of the while-loop terminates. The second while-loop (Lines 16–19) terminates because the number of cycles is reduced by at least one at every iteration of the loops, and there can only be a finite number of cycles. □

THEOREM 8.2. *GBDC is polynomial in the number of nodes in G.*

PROOF. Let $n = |V|$ be the number of nodes in G. Computing sets ins and outs can be done in $O(n^2)$ time (perform forward and backward reachability from every node). Computing Out can also be done in $O(n^2)$ time (Line 5 or 14). The while-loop at Line 6 is executed at most n times. Partitioning Out (Line 7) can be done in $O(n^3)$ time and this results in $k \leq n$ clusters. The while-loop at Lines 10–12 is iterated no more than n^2 times and the safe-to-add-f' condition can be checked in $O(n^3)$ time. The quotient graph produced by the while-loop at Line 6 contains at most n nodes. Checking the condition at Line 16 can be done in $O(n)$ time, and this process also returns a cycle, if one exists. Executing Line 18 can also be done in $O(n)$ time. The loop at Line 16 can be executed at most n times, since at least one cluster is removed every time. □

Note that the above complexity analysis is largely pessimistic. A tighter analysis as well as algorithmic optimizations are beyond the scope of this paper and are left for future work. Also note that GBDC is polynomial in the IODAG G, which, being the result of the unfolding step, can be considerably larger than the original SDF graph. This is because the size of G depends on the repetition vector and therefore ultimately on the hyper-period of the system. Finding ways to deal with this complexity (which, it should be noted, is common to all methods for SDF graphs that rely on unfolding or similar steps) is also part of future work.

GBDC is correct, in the sense that, first, it produces disjoint clusters and clusters all internal nodes, second, the resulting clustered graph is acyclic, and third, the resulting graph contains no input-output dependencies that were not already present in the input graph.

THEOREM 8.3. *GBDC produces disjoint clusters and clusters all internal nodes.*

PROOF. Disjointness is ensured by the fact that only unclustered nodes (i.e., nodes in $V_{\text{int}} \setminus \bigcup \mathcal{C}$) are added to the set Out (Lines 5 and 14) or to a newly created cluster C_i (Line 11). That all internal nodes are clustered is ensured by the fact that the while-loop at Line 6 does not terminate until all internal nodes are clustered. □

THEOREM 8.4. *GBDC results in an acyclic quotient graph.*

PROOF. This is ensured by the fact that all potential cycles are removed in Lines 16–19. □

THEOREM 8.5. *GBDC produces a quotient graph $G_{\mathcal{C}}$ that has the same input-output dependencies as the original graph G.*

PROOF. We need to prove that $\forall x \in V_{\text{in}}, y \in V_{\text{out}} : (x, y) \in E^* \iff (x, y) \in E^*_{\mathcal{C}}$. We will show that this holds for the quotient graph produced when the while-loop of Lines 6–15 terminates. The fact that Lines 16–19 preserve IO dependencies is shown in Lemma 5 of Lublinerman et al. [2009].

The $\Rightarrow$ direction is trivial by construction of the quotient graph. There are two places where false IO dependencies can potentially be introduced in the while-loop of Lines 6–15: at Lines 7–8, where a new set of clusters is created and added to $\mathcal{C}$; or at Line 11, where a new node is added to an existing cluster. We examine each of these cases separately.

Consider first Lines 7–8: A certain number $k \geq 1$ of new clusters are created here, each containing one or more nodes. This can be seen as a sequence of operations: first, create cluster C_1 with a single node $f \in \text{Out}$, then add to C_1 a node $f' \in \text{Out}$ such that $\mathsf{ins}(f) = \mathsf{ins}(f')$ (if such an f' exists), and so on, until C_1 is complete; then create cluster C_2 with a single node, and so on, until all clusters $\mathsf{C}_1, \ldots, \mathsf{C}_k$ are complete. It suffices to show that no such creation or addition results in false IO dependencies.

Regarding creation, note that a cluster that contains a single node cannot add false IO dependencies, by definition of the quotient graph. Regarding addition, we claim that if a cluster C is such that $\forall f, f' \in \mathsf{C} : \mathsf{ins}(f) = \mathsf{ins}(f')$, then adding a node f'' such that $\mathsf{ins}(f'') = \mathsf{ins}(f)$, where $f \in \mathsf{C}$, results in no false IO dependencies. To see why the claim is true, let $y \in \mathsf{outs}(f'')$. Then $\mathsf{ins}(f'') \subseteq \mathsf{ins}(y)$. Since $\mathsf{ins}(f'') = \mathsf{ins}(f)$, for any $x \in \mathsf{ins}(f)$, we have $(x, y) \in E^*$. Similarly, for any $y \in \mathsf{outs}(f)$ and any $x \in \mathsf{ins}(f'')$, we have $(x, y) \in E^*$.

Consider next Line 11: The fact that f' is chosen to be a predecessor of some node $f \in \mathsf{C}_i$ implies that $\mathsf{ins}(f') \subseteq \mathsf{ins}(f) \subseteq \mathsf{ins}(\mathsf{C}_i)$. There are two cases where a new dependency can be introduced. Case 2(a): either between some input $x \in \mathsf{ins}(\mathsf{C}_i)$ and some output $y \in \mathsf{outs}(f')$; Case 2(b): or between some input $x' \in \mathsf{ins}(f')$ and some output $y' \in \mathsf{outs}(\mathsf{C}_i)$. In Case 2(a), the safe-to-add-f' condition at Line 10 ensures that if such

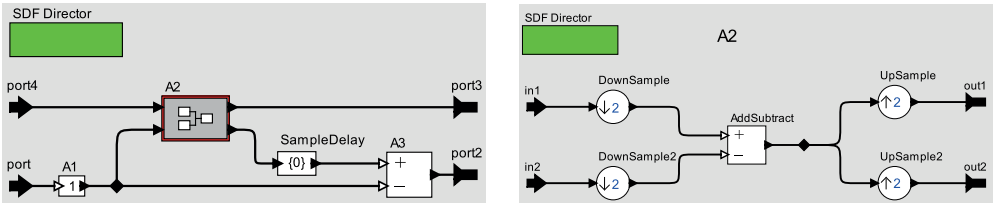

Fig. 15. A hierarchical SDF model in Ptolemy II. The internal diagram of composite actor A2 is shown to the right.

x and y exist, then y already depends on x, otherwise, f' is not added to C_i. In Case 2(b), $\mathsf{ins}(f') \subseteq \mathsf{ins}(\mathsf{C}_i)$ implies $x' \in \mathsf{ins}(\mathsf{C}_i)$. This and $y' \in \mathsf{outs}(\mathsf{C}_i)$ imply that $(x', y') \in E^*$: indeed, if this is not the case, then cluster C_i already contains a false IO dependency before the addition of f'. □

8.2. Clustering for Closed Models

It is worth discussing the special case where clustering is applied to a closed model, that is, a model where all input ports are connected. This in particular happens with top-level models used for simulation, which contain source actors that provide the input data. By definition, the IODAG produced by the unfolding step for such a model contains no input ports. In this case, a monolithic clustering that groups all nodes into a single cluster suffices and the GBDC algorithm produces the monolithic clustering for such a graph. Such a clustering will automatically give rise to a single firing function. Simulating the model then consists in calling this function repeatedly.

9. IMPLEMENTATION

We have built a preliminary implementation of the SDF modular code generation described above in the open-source Ptolemy II framework [Eker et al. 2003] (http://ptolemy.org/). The implementation uses a specialized class to describe composite SDF actors for which profiles can be generated. These profiles are captured in Java, and can be loaded when the composite actor is used within another composite. For debugging and documentation purposes, the tool also generates in the GraphViz format DOT (http://www.graphviz.org/) the graphs produced by the unfolding and clustering steps.

Using our tool, we can, for instance, generate automatically a profile for the Ptolemy II model depicted in Figure 15. This model captures the SDF graph given in Figure 3 of Falk et al. [2008]. Actor A2 is a composite actor designed so as to consume 2 tokens on each of its input ports and produce 2 tokens on each of its output ports each time it fires. For this, it uses the DownSample and UpSample internal actors: DownSample consumes 2 tokens at its input and produces 1 token at its output; UpSample consumes 1 token at its input and produces 2 tokens at its output. Actors A1 and A3 are homogeneous. The SampleDelay actor models an initial token in the queue from A2 to A3. All other queues are initially empty.

Assuming a monolithic profile for A2, GBDC generates for the top-level Ptolemy model the clustering shown to the left of Figure 16. This graph is automatically generated by DOT from the textual output automatically generated by our tool. The two replicas of A1 are denoted A1_1_0 and A1_2_0, respectively, and similarly for A2 and A3. Two clusters are generated, giving rise to the profile shown to the right of the figure. It is worth noting that there are 4 identical backward dependency edges generated for this profile (only one is shown). Moreover, all dependency edges are redundant in

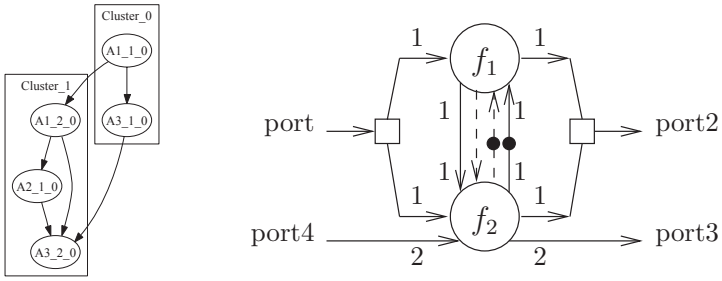

Fig. 16. Clustering (left) and SDF profile (right) of the model of Figure 15.

this case, thus can be removed. Finally, notice that the profile contains only two nodes, despite the fact that the Ptolemy model contains 9 actors overall.

10. CONCLUSIONS AND PERSPECTIVES

Hierarchical SDF models are not compositional: a composite SDF actor cannot be represented as an atomic SDF actor without loss of information that can lead to deadlocks. Extensions such as CSDF are not compositional either. In this paper we introduced DSSF profiles as a compositional representation of composite actors and showed how this representation can be used for modular code generation. In particular, we provided algorithms for automatic synthesis of DSSF profiles of composite actors given DSSF profiles of their sub-actors. This allows the handling of hierarchical models of arbitrary depth. We showed that different trade-offs can be explored when synthesizing profiles, in terms of compactness (keeping the size of the generated DSSF profile minimal) versus reusability (preserving information necessary to avoid deadlocks), as well as algorithmic complexity. We provided a heuristic DAG clustering method that has polynomial complexity and ensures maximal reusability.

In the future, we plan to examine how other DAG clustering algorithms could be used in the SDF context. This includes the clustering algorithm proposed by Lublinerman and Tripakis [2008b], which may produce overlapping clusters, with nodes shared among multiple clusters. This algorithm is interesting because it guarantees an upper bound on the number of generated clusters, namely, $n + 1$, where n is the number of outputs in the DAG. Overlapping clusters result in complications during profile generation that need to be resolved.

Apart from devising or adapting clustering algorithms in the SDF context, part of future work is also to implement this algorithms in a tool such as Ptolemy, and compare their performance.

Another important problem is the efficiency of the generated code. Different efficiency goals may be desirable, such as buffer size, code length, and so on. Problems of code optimization in the SDF context have been extensively studied in the literature (for instance, [Bhattacharyya et al. 1996; Sriram and Bhattacharyya 2009]). One direction of research is to adapt existing methods to the modular SDF framework proposed here.

We would also like to study possible applications of DSSF to contexts other than modular code generation, for instance, compositional performance analysis, such as throughput or latency computation. Finally, we plan to study possible extensions towards dynamic data flow models as well as towards distributed, multiprocessor implementations.

ACKNOWLEDGMENTS

The authors would like to thank Jörn Janneck and Maarten Wiggers for their valuable input.

REFERENCES

ARBAB, F. 2005. Abstract behavior types: A foundation model for components and their composition. *Sci. Comput. Program. 55,* 1–3, 3–52.

BHATTACHARYYA, S., LEE, E., AND MURTHY, P. 1996. *Software Synthesis from Dataflow Graphs*. Kluwer.

BILSEN, G., ENGELS, M., LAUWEREINS, R., AND PEPERSTRAETE, J. A. 1995. Cyclo-static data flow. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP-95)*. 3255–3258.

BLIUDZE, S. AND SIFAKIS, J. 2007. The algebra of connectors: Structuring interaction in bip. In *Proceedings of the International Conference on Embedded Software (EMSOFT'07)*. 11–20.

BROCK, J. AND ACKERMAN, W. 1981. Scenarios: A model of non-determinate computation. In *Proceedings of the International Colloquium on Formalization of Programming Concepts*. 252–259.

DE ALFARO, L. AND HENZINGER, T. 2001a. Interface automata. In *Proceedings of the SIGSOPT Symposium on Foundations of Software Engineering (FSE)*.

DE ALFARO, L. AND HENZINGER, T. 2001b. Interface theories for component-based design. In *Proceedings of the International Conference on Embedded Software (EMSOFT'01)*. LNCS, vol. 2211, Springer.

EKER, J., JANNECK, J., LEE, E., LIU, J., LIU, X., LUDVIG, J., NEUENDORFFER, S., SACHS, S., AND XIONG, Y. 2003. Taming heterogeneity – the Ptolemy approach. *Proc. IEEE 91,* 1, 127–144.

FALK, J., KEINERT, J., HAUBELT, C., TEICH, J., AND BHATTACHARYYA, S. 2008. A generalized static data flow clustering algorithm for MPSOC scheduling of multimedia applications. In *Proceedings of the International Conference on Embedded Software (EMSOFT'08)*. 189–198.

GEILEN, M. 2009. Reduction of synchronous dataflow graphs. In *Proceedings of the Design Automation Conference (DAC'09)*.

JONSSON, B. 1994. A fully abstract trace model for dataflow and asynchronous networks. *Distrib. Comput. 7,* 4, 197–212.

KAHN, G. 1974. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*.

KOPETZ, H. 1999. Elementary versus composite interfaces in distributed real-time systems. In *Proceedings of the 4th International Symposium on Autonomous Decentralized Systems (ISADS'99)*. 1–8.

LEE, E. AND MESSERSCHMITT, D. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput. 36,* 1, 24–35.

LUBLINERMAN, R., SZEGEDY, C., AND TRIPAKIS, S. 2009. Modular code generation from synchronous block diagrams: Modularity vs. code size. In *Proceedings of the SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*. 78–89.

LUBLINERMAN, R. AND TRIPAKIS, S. 2008a. Modular code generation from triggered and timed block diagrams. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*.

LUBLINERMAN, R. AND TRIPAKIS, S. 2008b. Modularity vs. reusability: Code generation from synchronous block diagrams. In *Proceedings of the Design, Automation and Test in Europe (DATE'08)*. 1504–1509.

LYNCH, N. AND TUTTLE, M. 1987. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC'87)*. 137–151.

PINO, J., BHATTACHARYYA, S., AND LEE, E. 1995. A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs. Tech. rep. UCB/ERL M95/36, EECS Department, University of California, Berkeley.

SRIRAM, S. AND BHATTACHARYYA, S. 2009. *Embedded Multiprocessors: Scheduling and Synchronization* 2nd Ed. CRC Press.

STARK, E. W. 1995. An algebra of dataflow networks. *Fundam. Inform. 22,* 1/2, 167–185.

THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. 2002. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*. LNCS, vol. 2304, Springer.

Received May 2010; revised September 2010; accepted December 2010