A Survey of Parametric Dataflow Models of Computation

ADNAN BOUAKAZ, PASCAL FRADET, and ALAIN GIRAULT, INRIA; Univ. Grenoble Alpes

Dataflow models of computation (MoCs) are widely used to design embedded signal processing and streaming systems. Dozens of dataflow MoCs have been proposed in the few last decades. More recently, several *parametric* dataflow MoCs have been presented as an interesting trade-off between analyzability and expressiveness. They offer a controlled form of dynamism under the form of parameters (*e.g.,* parametric rates), along with run-time parameter configuration. This survey provides a comprehensive description of the existing parametric dataflow MoCs (constructs, constraints, properties, static analyses) and compares them using a common example. The main objectives are to help designers of streaming applications to choose the most suitable model for their needs and to pave the way for the design of new parametric MoCs.

CCS Concepts:•**Theory of computation** → **Models of computation;**

Additional Key Words and Phrases: dataflow graphs, reconfiguration, parameterization, static analysis

ACM Reference Format:
Adnan Bouakaz, Pascal Fradet, and Alain Girault, 2016. A Survey of Parametric Dataflow Models of Computation. *ACM Trans. Des. Autom. Electron. Syst.* , , Article (January 2016), 25 pages.
DOI:

1. INTRODUCTION

A model of computation (MoC) consists of a set of laws that govern the interaction of components in a design [Lee 2001]. For embedded real-time systems, models of computation must deal with control, concurrency and time. Dataflow MoCs are widely used to design such systems and, in particular, streaming applications. They are usually defined by the following elements:

(1) Components, called *actors*, are computational nodes which, upon *firing*, consume a finite number of inputs, execute some (usually unspecified) code, and produce a finite number of outputs.
(2) Actors communicate data messages, called *tokens*. Most dataflow MoCs abstract from the actual values of tokens. Some MoCs differentiate between data tokens and control tokens.
(3) Actors communicate through one to one *unbounded* FIFO channels. An actor cannot test for the presence or absence of data, and hence blocks if it attempts to consume tokens from an empty channel.

Dataflow MoCs are usually classified into two major categories: static dataflow MoCs [Ha and Oh 2012] and dynamic dataflow MoCs [Bhattacharyya et al. 2012]. Static MoCs, such as synchronous dataflow (SDF) [Lee and Messerschmitt 1987] and cyclo-static dataflow (CSDF) [Bilsen et al. 1996], enjoy determinism, the decidability of many properties, and powerful compile-time optimizations. They usually have fixed rates, *e.g.,* an actor consumes and produces the same amount of tokens at each firing. Dynamic MoCs, such as Kahn process networks (KPN) [Kahn 1974], Boolean-

Authors' address: Inria Grenoble Rhône-Alpes, 655 Avenue de l'Europe, 38330 Montbonnot-Saint-Martin. Email: *first.last@inria.fr*

DOI:

controlled dataflow (BDF) [Buck 1993], dataflow process networks (DPN) [Lee and Parks 2002], and the CAL actor language (CAL) [Eker and Janneck 2003] have data-dependent communication protocols. They are more expressive, but at the cost of giving up some compile-time analyses and resorting to run-time execution management. The interested reader will find in [Ha and Oh 2012] and [Bhattacharyya et al. 2012] two surveys presenting the main static and dynamic dataflow MoCs respectively.

The present survey focuses on *parametric* dataflow MoCs: a sub-class of dynamic MoCs that has emerged in the last few years. In parametric MoCs, the dynamism is controlled and takes the form of parameters (*e.g.,* parametric rates) and run-time parameter configuration. This trade-off allows more compile-time analyzability than dynamic MoCs (*e.g.,* KPN, DPN), while being more expressive than static MoCs (*e.g.,* SDF). This paper provides comprehensive characterizations (constructs, constraints, properties, static analyses) of the existing parametric MoCs: Parametric Synchronous Data Flow (PSDF) [Bhattacharya and Bhattacharyya 2001], Variable Rate Data Flow (VRDF) [Wiggers et al. 2008], Schedulable Parametric Data Flow (SPDF) [Fradet et al. 2012], Boolean Parametric Data Flow (BPDF) [Bebelis et al. 2013; Bempelis 2015], Parameterized and Interfaced Synchronous Data Flow (PiSDF) [Desnos et al. 2013], and FSM-based parameterized scenario-aware dataflow (PFSM-SADF) [Skelin et al. 2015; 2014]. These descriptions permit the comparison of the different MoCs and help designers of streaming applications to choose the MoC best suited to their needs.

The article is organized as follows. Section 2 first introduces the basic dataflow terminology by presenting succinctly SDF and clarifies the criteria used to characterize and compare dataflow MoCs. Section 3 presents the main characteristics, properties and analyses of parametric MoCs. Sections 4 to 9 describe parametric MoCs (PSDF, VRDF, SPDF, BPDF, PiSDF, PFSM-SADF) in chronological order. Finally, Section 10 summarizes their main features and compares them using a common example.

2. BACKGROUND

We first present succinctly SDF [Lee and Messerschmitt 1987] because it constitutes the foundation of all existing parametric dataflow MoCs. Then we define four characterization criteria that are commonly used to compare MoCs.

2.1. The SDF MoC

An SDF graph is a directed graph, where nodes – called *actors* – are functional units. The actors are connected by *edges* which can be seen as FIFO channels. The atomic execution of a given actor – called *actor firing* – consumes data tokens from all its incoming edges (its *inputs*) and produces data tokens to all its outgoing edges (its *outputs*). The number of tokens consumed or produced at a given edge at each firing is called the *rate*. An actor can fire only when *all* its input edges have enough tokens (*i.e.,* at least the number specified by the rate of the corresponding edge). In SDF, all rates are constant integers known at compile time.

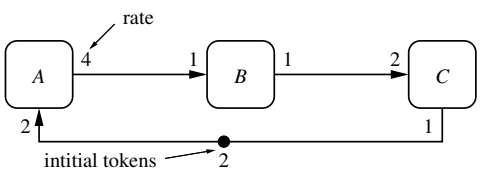


Fig. 1. A simple SDF graph.

Fig. 1 shows a simple SDF graph with three interconnected actors A, B and C. Actor A has one input and one output port, whose rates are 2 and 4, respectively.

The *state* of a dataflow graph is the number of tokens present at each edge (*i.e.,* buffered in each FIFO). Each edge carries zero or more tokens at any moment of time. The *initial state* of the graph is specified by the number of *initial tokens*. For instance, edge (C, A) in Fig. 1 has two initial tokens. After the first firing of actor A, the edge (A, B) gets four tokens while the two tokens of (C, A) are consumed.

A major advantage of SDF is that, if it exists, a bounded schedule can be found statically. Such a schedule ensures that each actor is eventually fired (ensuring liveness) and that the graph returns to its initial state after a certain sequence of firings (ensuring boundedness of the FIFOs). Such sequence is called an *iteration* and can be repeated an infinite number of times (in the case of a reactive system).

The numbers of firings of the different actors per iteration are computed by solving the so-called *system of balance equations*. This system is made of one equation per edge. Consider an edge (X, Y) with output and input rates r_x and r_y; its balance equation is:

$$z_X \cdot r_x = z_Y \cdot r_y \tag{1}$$

which states that, in an iteration, the number of firings of the producer X, denoted z_X, multiplied by its rate r_x, should be equal to the same expression for the consumer Y. For example, the balance equation for edge (A, B) in Fig. 1 is: $z_A \cdot 4 = z_B \cdot 1$.

The existence of strictly positive solutions of the system of balance equations is referred to as *consistency*, sometimes as *rate consistency*. Such a set of solutions forms a *repetition vector* noted $[z_X, z_Y, \ldots]$. Multiplying the solutions by the same positive constant makes another set of solutions. One usually considers only the minimal strictly positive integer solutions that are obtained by eliminating common factors. The graph of Fig. 1 is rate-consistent, and the minimal solutions are: $z_A = 1$, $z_B = 4$ and $z_C = 2$. Such a set of solutions forms a *repetition vector* noted $[z_A, z_B, z_C] = [1, 4, 2]$.

Consistency is a key property of a MoC because it guarantees that programs written in this MoC can be executed in bounded memory, which is crucial for embedded systems. Another important property is *liveness*, which guarantees that programs will never deadlock. Any acyclic (and consistent) SDF graph is live[1]. For consistent cyclic SDF graphs, liveness depends on the number and distribution of initial tokens.

2.2. Characterization criteria

Several criteria are usually considered to characterize dataflow MoCs or to motivate the many variations that have been proposed.

— *Expressiveness*. The notion of expressiveness has no clear or formal meaning in the dataflow community. It often refers to syntactic features or ease of expression and is related to succinctness (used in [Stuijk 2007]) and compactness. One model is said to be more expressive than another if its syntactic constructs allow shorter representations.

— *Hierarchy and compositionality.* All dataflow MoCs can be seen as a composition language to assemble programs abstracted in actors. Some dataflow MoCs offer hierarchy as a mechanism for internal modularization and composition. For instance, Ptolemy II [Eker et al. 2003] allows the hierarchical composition of graphs specified in different (dataflow or other) MoCs. However, hierarchy (or composite actors) may lose information compared to a flat graph and fail to preserve deadlock-absence (see [Tripakis et al. 2013] for examples of non-compositional hierarchical SDF graphs).

[1]Liveness only matters for consistent SDF graphs.

— *Analyzability.* The analyzability of a MoC is determined by the decidability of important properties and the complexity of the corresponding algorithms. The most important analyses of dataflow models are consistency, liveness, throughput/latency computation, and compile-time scheduling. Analyzability and expressivity are to some extent antagonistic.
— *Reconfiguration.* Reconfiguration is described by the *explicit* mechanisms of a MoC for handling changes at run-time. Different reconfiguration mechanisms have been proposed: scenarios (*e.g.,* FSM-SADF [Geilen and Stuijk 2010]), actor modes (*e.g.,* EIDF [Plishker et al. 2008]), message passing (*e.g.,* StreamIt [Thies et al. 2002]) and parameterization.

In this survey, we focus on *parameterization*. This reconfiguration technique provides concise constructs for enumerating large numbers of operating modes. Another advantage of parameterization is to replace successive analyses of all possible configurations by a single symbolic (parametric) analysis. Such a symbolic analysis may not be as precise, but it usually ensures that its results are a safe approximation of the exact analysis of all the individual configurations.

Many features of dataflow MoCs could be parameterized: rates, initial tokens, channels or actors (*i.e.,* their presence/absence), internal states, execution times (for throughput analysis or scheduling), and so on. The existing parametric dataflow MoCs use mostly parametric rates and parametric conditions for the (de)activation of channels[2]. In this survey, we review the following specific characteristics of these MoCs:

— The kind of parameters (*e.g.,* parametric rates, edge conditions to change the topology). In most MoCs, parameters may also change the functionality of actors but this side-effect is most often left unspecified.
— The allowed parametric functions (simple, products, or polynomials) and their type/range (*e.g.,* $\mathbb{N}, \mathbb{N}^*, Bool, \ldots$).
— How parameters are produced (*e.g.,* by an actor, a centralized controller, ...) and communicated (*e.g.,* using dataflow channels, shared variables, ...).
— When parameters may be set (*e.g.,* between global iterations, within inner iterations, or hierarchically).
— The symbolic analyses considered and studied (consistency, liveness, throughput, buffer size, latency, scheduling).

3. CHARACTERISTICS, PROPERTIES, AND ANALYSES OF PARAMETRIC MOCS

Parametric MoCs have all in common a number of features: they all permit to define *flat graphs* (but some also support hierarchical graphs) with *parametric rates* (but some also propose other kind of parameters) *changing between iterations* (but some also allow inner iteration changes) with *fixed numbers of initial tokens* (but one MoC allows also parametric numbers of initial tokens). To present the common characteristics of parametric MoCs, we consider a base model, that we call Parametric Rate DataFlow (PRDF), which embodies these mutual features.

A PRDF graph $G = (V, E)$ is a connected directed graph that consists of a finite set of actors V and a finite set of communication channels E. Each channel $e \in E$ may contain $\delta(e) \in \mathbb{N}$ initial tokens. A channel $A \xrightarrow{pr \quad cr} B$ between the producer A and the consumer B is annotated with a production rate pr and a consumption rate cr. A rate R is defined by the following grammar:

$$R ::= k \mid p \mid kp \text{ where } k \in \mathbb{N}^* \text{ and } p \in \mathcal{P}$$

[2]PFSM-SADF also uses parametric execution times and PSM-CFDF introduces parametric set of modes.

where $\mathcal{P}$ denote the finite set of integer parameters of the graph. The domain of values of each parameter p is denoted by $D(p)$. It is a finite set of at least two strictly positive integers. The maximum value of a parameter p is denoted by $p_{\max}$. Some parametric MoCs consider richer rates (*e.g.,* null values, parameters products, *etc.*).

Parameter configuration (*i.e.,* setting the values of parameters) occurs dynamically at global iteration boundaries. The set of all possible parameter configurations is denoted by $\mathcal{P}_{conf}$. Some parametric MoCs allow parameter configurations to occur within iterations. However, this feature requires additional constraints to ensure that the graph returns to its initial state at the end of the iteration.

As most parametric models, PRDF does not provide specific constructs for modeling the changes of configurations. Parameter configuration can be chosen by a central controller or specific actors (called *modifiers*) and could be specified, for instance, using a deterministic or a non-deterministic finite-state machine. The communication of parameters to actors may use shared data or specific dataflow channels.

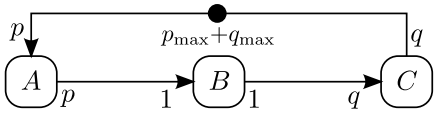
Most static analyses for parametric MoCs are agnostic to the configuration transition systems. They assume that a configuration $c \in \mathcal{P}_{conf}$ is non-deterministically chosen at the beginning of every global iteration. Therefore, these analyses might be conservative and could be further refined by taking into account a specific configuration transition system.

3.1. Consistency

Consistency is a necessary condition for the existence of an iteration. A bounded execution (*i.e.,* execution with bounded channels) can be obtained by repeating infinitely this iteration. Non-consistent graphs cannot execute in bounded memory and are rejected at compile-time. The *repetition vector* $\vec{z}$ of the graph indicates the number of firings of actors per iteration. It is obtained by solving the so-called system of *balance equations*. For an edge $A \xrightarrow{p \quad q} B$ the balance equation is $z_A p = z_B q$, which states that all produced tokens during one iteration are consumed in the same iteration. Note that *acyclic* dataflow graphs are always consistent (the cyclic property referring to the undirected version of the graph).

Consistency of parametric graphs consists in symbolically solving the system of balance equations (instead of solving it numerically as in SDF). Fig. 2(a) depicts a simple cyclic PRDF graph where the set of parameters is $\mathcal{P} = \{p, q\}$ with $D(p) = [2..10]$ and $D(q) = \{3, 5, 7\}$. The black dot represents $(p_{max} + q_{max})$ initial tokens. The associated system of symbolic balance equations is shown in Fig. 2(b).

(a)



$$\begin{aligned} z_A \times p &= z_B \times 1 \\ z_B \times 1 &= z_C \times q \\ z_C \times q &= z_A \times p \end{aligned} \qquad \text{(b)}$$

Fig. 2. (a) A cyclic PRDF graph. (b) The corresponding balance equations.

A PRDF graph is *consistent* if the system of balance equations accepts a non-null solution for every parameter configuration in $\mathcal{P}_{conf}$. A PRDF graph is *strongly consistent* if the system of balance equations accepts a non-null solution for every configuration in $\mathbb{N}^{*|\mathcal{P}|}$ (*i.e.,* the domain of every parameter is considered to be $\mathbb{N}^*$). The symbolic solution of the system of balance equations of Fig. 2(b) is [3] $\vec{z} = \left[\frac{q}{\gcd(p,q)}, \frac{pq}{\gcd(p,q)}, \frac{p}{\gcd(p,q)}\right]$.

[3]Some existing analyses compute a simpler solution by not simplifying by $\gcd(p, q)$ *i.e.,* $\vec{z} = [q, pq, p]$

A non-consistent PRDF graph has to be rejected not only because it might not execute in bounded memory but also because the configuration rule (*i.e.,* changing parameters at iteration boundaries) requires that a repetition vector (*i.e.,* iteration) exists.

Strong consistency is generally a conservative consistency analysis. However, it is an exact analysis when there is no control on parameter configuration in the model, *i.e.,* when $\mathcal{P}_{conf}$ is the Cartesian product of all parameter domains $\prod_{p\in\mathcal{P}} D(p)$: in this case it can be shown that a PRDF graph is consistent if and only if it is strongly consistent. However, this would not hold if rates were expressed as polynomials: in this case, strong consistency would reject some programs that are actually consistent *w.r.t.* the domains of parameters.

3.2. Liveness

Consistency is not a sufficient condition to ensure the existence of an iteration the graph must be live *i.e.,* deadlock free. To ensure the liveness of a SDF graph, one iteration is found by abstract execution. A PRDF graph is live if and only if each configuration (*i.e.,* the corresponding SDF graph) is live. For instance, it is easy to check that all configurations of the graph in Fig. 2 are live. This can be used to check liveness of FSM-SADF [Geilen and Stuijk 2010] graphs where there are few scenarios. For parametric models with large configuration spaces, this method would be unpractical. Parametric liveness analyses have to be considered.

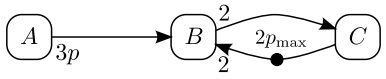
First, we define a *quasi-static schedule* as an ultimately periodic infinite firing sequence, $S_1 S_2^{\omega}$ where the (possibly empty) prologue S_1 and the periodic sequence S_2 can be described by the following grammar:

$S ::= A \mid S^f \mid S_1\ S_2$ where A denotes an actor and f a parametric integer expression.

Acyclic consistent PRDF graphs are always live since the topological order of actors is sufficient to define a single appearance schedule [Battacharyya et al. 1996]. Similarly, if all cycles can be broken, *i.e.,* if at least one edge in each cycle is *saturated* [Fradet et al. 2012], the graph is live. An edge is saturated if it contains enough initial tokens for its consumer A to fire z_A times. For graph in Fig.2, if the edge $C \to A$ contains at least $\max(z_A \times p) = 70$ tokens, then the single appearance quasi-static schedule $(\uparrow_p \uparrow_q A^q B^{pq} C^p)^{\omega}$ is valid, where "$\uparrow_x$" denotes the setting of parameter x. Note that the graph obtained by deleting all saturated edges is not equivalent to the original graph because these edges actually constrain the pipelining in the execution.

The technique proposed in [Bebelis et al. 2013; Bempelis 2015], called PSLC for Parametric SDF-like Liveness Checking, is a conservative parametric liveness analysis that can be applied to arbitrary PRDF graphs. It constructs a quasi-static schedule by an abstract execution that can be seen as a parametric version of the one used for SDF. If an actor A has a parametric number of firings per iteration, *i.e.,* $z_A = k \times P$ where k is a positive integer and P is a product of parameters, then each time A is fired in the schedule, it is fired at least P times.

The repetition vector of the PRDF graph in Fig. 3(a) is $\vec{z} = [1, 3p, 6p]$; its number of initial tokens in is $2p_{\max}$; and its initial state is $[0, 0, 2p_{\max}]$. The constructed live abstract execution is shown in Fig. 3(b). As a result, the schedule is $(\uparrow_p \uparrow_q A(B^p C^{2p})^3)^{\omega}$. This technique can also use preliminary clustering. If there were only two initial tokens in Fig. 3(a), the cycle consisting of actors B and C would be clustered and the live schedule $(\uparrow_p \uparrow_q A(BC^2)^{3p})^{\omega}$ would be found. The technique however is not complete and fails to attest the liveness of the graph in Fig. 3(a).

(a) 

(b)
$$[0,0,2p_{\max}] \xrightarrow{A} [3p,0,2p_{\max}] \xrightarrow{B^p}$$
$$[2p,2p,2p_{\max}-2p] \xrightarrow{C^{2p}} [2p,0,2p_{\max}] \xrightarrow{B^p}$$
$$[p,2p,2p_{\max}-2p] \xrightarrow{C^{2p}} [p,0,2p_{\max}] \xrightarrow{B^p}$$
$$[0,2p,2p_{\max}-2p] \xrightarrow{C^{2p}} [0,0,2p_{\max}]$$

Fig. 3. (a) A PRDF graph attested live by PSLC. (b) Its abstract execution.

3.3. Performance analysis

Timing (throughput, latency) and memory (buffer sizes) analyses are very important steps in embedded real-time systems design. In particular, throughput is a key property of real-time streaming systems. For example, a video decoder must guarantee a throughput expressed as a minimum number of frames per second. These properties depend strongly on the scheduling policy. Most studies consider self-timed execution [Sriram and Bhattacharyya 2000] in which an actor fires as soon as it is enabled (*i.e.,* has enough tokens on its input channels). This is also known as the As Soon As Possible (ASAP) scheduling, which assumes a sufficient, even infinite, number of processing units[4]. One important property of this policy is that the execution (at least for HSDF, SDF, CSDF and UCSDF) consists of a *prologue* phase followed by a periodic phase called the *steady* state of the execution. The throughput and period, defined as limits, are computed from the steady state. Moreover, ASAP scheduling allows a maximal throughput.

Such analyses require to extend the MoC with timing information, usually the worst-case execution time (WCET) of each actor. For the PRDF model, this can be defined as a function $\tau : V \times \mathcal{P}_{conf} \longrightarrow \mathbb{N}^*$ such that $\tau(A, c)$ is the worst-case execution time of A in configuration c. At the time of writing, only few *symbolic* analyses have been developed for parametric dataflow MoCs. The analyses proposed in [Bouakaz et al. 2016] can be applied to PRDF with the following restrictions: (1) only acyclic graphs are considered, and (2) auto-concurrency is disabled (*e.g.,* by adding self-loops to actors). These symbolic analyses give timing and memory properties as parametric formulas that can be evaluated for any parameter configuration. They do not consider the cost of configuration transitions and assume a single but symbolic configuration. The proposed analyses are:

— The symbolic throughput $\mathcal{T}_G$ of the ASAP execution of an acyclic PRDF graph G (assuming large enough buffers). It has been proved that

$$\mathcal{T}_G = \frac{1}{\max\limits_{A \in V}\{z_A \tau_A\}} \text{ with } \tau_A = \max_c\{\tau(A,c)\} \tag{2}$$

However, configurations could change so frequently that a steady state is never reached. A different symbolic analysis computes $\dot{\mathcal{T}}_G(n)(c)$ an under-approximation of the throughput of the first n iterations where the (symbolic) configuration c is fixed. By considering a worst-case execution where the configuration is changed after each iteration and a strong synchronization is required at iteration boundaries, a worst-case under-approximation of the throughput can be computed. Formally,

$$\mathcal{T}_G \geq \min_{c \in \mathcal{P}_{conf}} \{\mathcal{T}_G(1)(c)\}$$

[4]These assumptions do not forbid to encode resource constraints in the graph

— Minimal buffer sizes that allow the maximal throughput $\mathcal{T}_G$ (assuming a fixed symbolic configuration). Symbolic safe upper bounds have been proposed for different topologies of acyclic graphs. For instance, it has been proved that if the size of each channel $A_i \xrightarrow{p_i \quad q_i} A_{i+1}$ in a chain $\{A_1 \xrightarrow{p_1 \quad q_1} A_2 \xrightarrow{p_2 \quad q_2} A_3 \rightarrow \cdots \rightarrow A_n\}$ is at least $2(p_i + q_i - \gcd(p_i, q_i))$, then the ASAP execution achieves the maximal throughput. Furthermore, exact formulas have been provided for a single edge with symbolic rates $A \xrightarrow{p \quad q} B$ and multiple heuristics to improve the results for chains and trees.
— The maximum input-output latency of chains, which is the maximum duration of an iteration in the ASAP execution of the graph (assuming a fixed symbolic configuration).

4. PARAMETRIC SYNCHRONOUS DATAFLOW (PSDF)

The first parametric dataflow MoC to be proposed was Parametric Synchronous Dataflow (PSDF), as a parameterization of SDF [Bhattacharya and Bhattacharyya 2001]. It can be roughly described as PRDF equipped with hierarchy. Actually, the proposed approach consists in a *meta-model* that can be applied to any MoC with a clear definition of iterations: it has been applied both to the parameterization of SDF, resulting in the PSDF MoC, of CFDF, resulting in the CF-PSDF MoC [Wang et al. 2013], and of CSDF, resulting in the PCSDF MoC [Kee et al. 2012]. The same approach has been followed with the PiMM meta-model [Desnos et al. 2013] (see Section 8). PSDF explicitly incorporates hierarchy and parameter communication; yet, many points (rates, analyses, *etc.*) remain unspecified.

4.1. Parameters

A PSDF actor A is parameterized with a set of parameters $\mathcal{P}_A$ and $\mathcal{P}_{conf,A}$ a set of valid parameter combinations for actor A. PSDF does not indicate how these configurations are specified (enumeration, constraints, ...). Parameters can be used to control the production and consumption rates of the actor or to control its functionality. The grammar for rate functions is not defined, but null rates are disallowed. Unlike other parametric models, an edge in PSDF can have a parametric number of initial tokens. As pointed out in Section 3, the semantic of parametric initial tokens is ambiguous at best and [Bhattacharya and Bhattacharyya 2001] does not specify any. For bounded memory analysis, the maximum values of rates and numbers of initial tokens must be specified and satisfied by the corresponding parametric functions. Again, no specific techniques or analyses are imposed or suggested.

4.2. Hierarchy

A PSDF actor is either an atomic actor (*i.e.,* an SDF actor with possibly parametric rates) or a hierarchical one. The dataflow input and output ports of a hierarchical PSDF actor (also called dataflow *interface ports*) are connected to ports of actors in the lower level. Each hierarchical actor A is composed of an obligatory graph (the *body* graph) and two optional PSDF graphs (the *init* and *subinit* graphs) which are responsible for the parameter reconfiguration of A. The *body* graph models the core functionality of A; therefore, its interface ports should be connected to the interface ports of A.

4.3. Parameter reconfiguration

All parameters of the *body* graph of a hierarchical actor A are set by its *init* and *subinit* graphs. Parameters of the *subinit* graph can be set by the *init* graph, provided by dataflow input ports of A, or left unspecified. All parameters of the *init* graph and unspecified parameters of the *subinit* graph are considered as the set of parameters of

A. These parameters must be set by the *init* and *subinit* graphs of the hierarchically higher-level actor.

Fig. 4 depicts a simple PSDF graph G. The *body* graph of G contains a hierarchical actor A. The parameters of an actor that are not set by its internal *init* and *subinit* graphs are indicated within curly braces (*e.g.,* $\{p\}$ in A). In our example, the only parameter of A set externally by the *init* graph of G is p. Dashed arrows represents control channels used to communicate parameter values. Interface ports are annotated with lower case letters (a, b, c, and d).

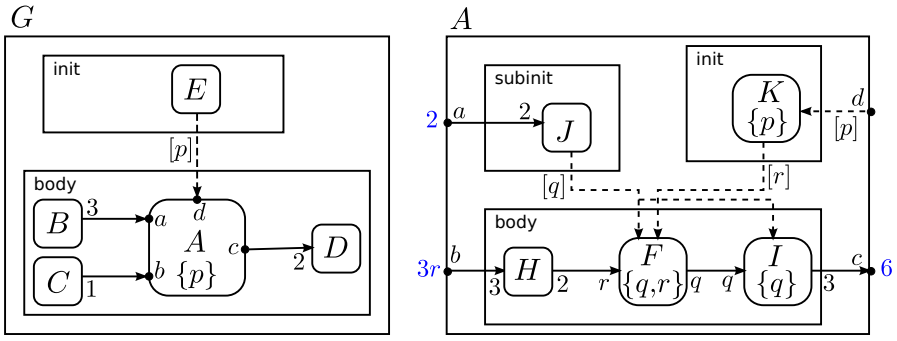


Fig. 4. A simple hierarchical PSDF graph.

As indicated by the informal semantics of PSDF, the *init* graph of a hierarchical actor A is executed at the beginning of each iteration of the parent graph (*i.e.,* the graph that contains actor A). Therefore, parameters of the *subinit* and *body* graphs of A that are set by the *init* graph can change after each iteration of the parent graph. In the example of Fig 4, the parameter r in actor A is set once every iteration of G, while p can be changed every iteration of the parent graph of G. At each firing of a hierarchical actor, its *subinit* graph is first invoked (*i.e.,* executed for one iteration), then its *body* graph. Therefore, parameters of the *body* graph and parameters of the *subinit* graph that are connected to dataflow inputs can change at every firing of the actor. In our example, the parameter q in A is changed at every firing of A. So, parameters in PSDF can change at iteration boundaries or, for those used in lower hierarchy levels, inside iterations. Note that control channels (dashed arrows in Fig. 4) are only used to depict parameter dependencies and not the actual transfer of values. Actually, the *init* and *subinit* graphs produce exactly one value per output. These values are used by a scheduler to set the parameter values in other graphs.

Consumption and production rates of the hierarchical actor A are not shown in graph G but they can be inferred from its internal implementation. For instance, the interface input port b is connected to actor H in the *body* graph of A. At each firing of A, the body graph is executed once, and hence H fires r times since the repetition vector of the *body* graph is $[z_H = r, z_F = 2, z_I = 2]$[5]. Therefore, actor H consumes $3r$ tokens at each firing of A. Thus, the consumption rate of port b at the level of G is $3r$. Similarly, the rates of ports a and c are 2 and 6 respectively (denoted in blue in Fig. 4). PSDF requires that these inferred rates depend only on parameters of A that are set by its *init* graph. Hence, these rates remain fixed during an iteration of the parent graph.

[5]Actually, this repetition vector should be divided by $gcd(r, 2)$ to represent the actual vector which is computed dynamically on values. To alleviate notations, we proceed with the simpler, non minimal, version.

To summarize, the execution of our example begins with the top most PSDF graph (*i.e.,* the parent of G, which contains only actor G), the *init* graph of G is first executed for one iteration (*i.e.,* firing actor E). This will set the value of p. Then, the *body* graph of G will be executed. Its execution starts by first firing all the *init* graphs of its hierarchical actors. Here, A is the only hierarchical actor, and its *init* graph will set the parameter r. Thus, an iteration of the *body* graph of G will be $B^2C^{9r}A^3D^9$. Each firing of A in an iteration of the *body* graph of G, starts by executing the *subinit* graph of A (and hence setting the parameter q), and then the *body* graph of A for one iteration $H^rF^2I^2$. All these actions can be represented by the following quasi-static schedule:

$$\left(E \uparrow_p \left(K \uparrow_r (B^2C^{9r}(J \uparrow_q (H^rF^2I^2))^3D^9)\right)\right)^\omega$$

Quasi-static scheduling techniques have been proposed for sub-classes of PSDF graphs (*i.e.,* acyclic graphs and cyclic graphs for which feedback loops can be broken).

PSDF checks (statically if possible, dynamically otherwise) that every (sub)graph is locally synchronous (*i.e.,* rate consistency, liveness and rates and tokens bounds); the execution is terminated in case of violation. Consistency and liveness could be statically checked for all possible configurations in a bottom-up way, starting by checking the deepest levels. Each level may consist of three graphs (*init*, *subinit*, and *body*) which must be checked separately. However, no parametric/symbolic analysis is proposed for consistency, liveness, or performance. Finally, without parametric initial tokens and PRDF-like rates, PSDF could be seen as a hierarchical PRDF with explicit parameter communication. The consistency and liveness symbolic analyses of Section 3 could be reused for such PSDF graphs. As already noted, a naive hierarchical deadlock loses information and may reject graphs that would be accepted if flattened. For hierarchical (non-parametric) SDF, this problem is solved in [Tripakis et al. 2013] using enriched actors called DSSF profiles. It is likely that such profiles could be adapted to PSDF and, more generally, to extend any parametric MoC with hierarchy.

5. VARIABLE RATE DATAFLOW (VRDF)

Variable rate dataflow (VRDF) [Wiggers et al. 2008] can be seen as PRDF extended with (possibly multiple) parameter changes inside the iteration and additional constraints on parameter uses. More precisely, VRDF has the following characteristics:

• Auto-concurrency is not allowed (each actor can be seen as having an implicit self-edge with one initial token).

• A rate is either a constant or a parameter; rates are described by the grammar

$$R ::= k \mid p \quad \text{where} \quad k \in \mathbb{N}^* \text{ and } p \in \mathcal{P} \text{ (parameters take their values in } \mathbb{N})$$

The domain of each parameter may contain zero, but it must contain at least one strictly positive value. This makes VRDF a dynamic MoC for both rate and topology.

• Parameter changes are not restricted to the boundaries of iterations. The value of each parameter is set by a single actor, its *modifier*, which can do so at each of its firings. Due to this reconfiguration mechanism and to the null rates, strong consistency does not longer imply boundedness. Therefore, VRDF imposes many restrictions on parameters usage. As in PRDF, the VRDF reconfiguration transition system is not specified; it could be deterministic or not.

• A parameter p can be used as rates of *at most two* actors, one of them being the modifier. Furthermore, if a parameter is used by a pair of actors, it must be used in the output rates of the modifier and in the input rates of the other actor. Moreover, if two actors A and B use the same parameter, then their solution in the computed symbolic repetition vector (the firings count per iteration) z_A and z_B must be equal.

These restrictions imply that, for any path from two actors A and B using p, if another parameter q occurs on this path, then it must be used once as an input rate and once as an output rate.

Fig. 5 shows a VRDF graph (dashed lines are explained later) which satisfies all the above conditions. Parameter p is used by actors B and D (its modifier), and parameter q is used by actors C (its modifier) and D. The repetition vector is $\vec{z} = [2p, 6, 6, 6, 3p]$.

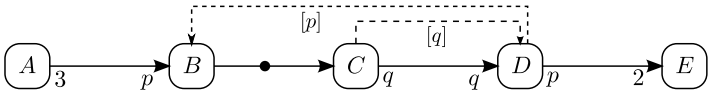
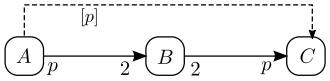


Fig. 5. A simple VRDF graph.

Parameter communication (between modifier and user) must be explicit. As illustrated in Fig. 5, it is denoted by a dashed arrow, with the notation "$[x]$" to indicate that the channel carries the values of parameter x. If a parameter p is used by a pair of actors A (its modifier) and B, then a control channel $A \xrightarrow{1\ 1} B$ has to be added to communicate the values of p. These channels never have initial tokens. Each time a modifier A fires, it modifies the value of p, produces tokens according to the value of p and sends the new value of p over the control channel. When the user B fires, it first consumes the value of p and consumes data tokens according to the value of p. Hence, there is a one-to-one correspondence between the firings of A and those of B. Therefore, whatever the chosen value of p, all produced tokens on a path from A to B will be also consumed (*i.e.,* guaranteeing boundedness). However, due to potential null rates, this correspondence could be broken and an additional constraint is needed. Indeed, consider the following graph:



where the first firing of A sets p to 1: the parameter is sent on the communication channel (A, C), a single token is produced on edge (A, B) and neither B nor C can fire. Suppose that from now on that p is set to 0 for an arbitrary number of firings. The values for p accumulate on (A, C) while B and C are still waiting for a second token. Therefore, the channel (A, C) cannot be bounded. To prevent this situation, VRDF enforces the following constraint: let G_p be the sub-graph made of all paths between two actors X and Y using parameter p, then the repetition vector of G_p must have $z_X = z_Y = 1$. This prohibits the rate 2 in the above example and enforces that rate to be 1. An alternative solution would be to enforce that parameters have large enough values to trigger both a production and a consumption, *i.e.,* in our example $p \geq 2$.

Another issue is that control channels may create deadlocks. For instance, if channel $B \rightarrow C$ in Fig. 5 does not contain an initial token, adding the control channel $D \rightarrow B$ creates a deadlock.

It is shown that a strongly consistent VRDF graph is bounded [Wiggers et al. 2008] (*i.e.,* can execute with bounded buffers). [Wiggers et al. 2008] proposes a conservative memory analysis that computes upper bounds on buffer sizes (size constraints being modeled by backward edges) for chains (which are always consistent). The analysis allows, if possible, a strictly periodic and ASAP execution to achieve a given throughput for all possible parameter reconfigurations.

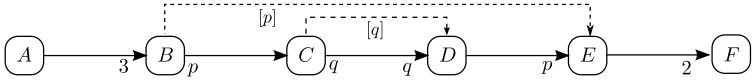
Quasi-static scheduling

There is a clear correspondence between a quasi-static schedule of a PRDF graph and its symbolic repetition vector. Due to unrestricted parameter changes, this is not the case for general VRDF graphs. Let us consider the VRDF graph of Fig. 5, whose repetition vector is $[2p, 6, 6, 6, 3p]$. In PRDF, p and q would be set at the beginning of the iteration, resulting in the following quasi-static schedule:

$$\left(\uparrow_p \uparrow_q A^{2p} B^6 C^6 D^6 E^{3p}\right)^\omega$$

In VRDF, parameters p and q change after each firing of D and C respectively. Since there is a one-to-one correspondence between the firings of C, D, and B, a quasi-static schedule could be $\left((\uparrow_q C \uparrow_p D\ E^f A^g B)^6\right)^\omega$ where f and g denote parametric expressions. For the execution to be bounded, for each parameter configuration, E has to fire as much as possible in order to consume all tokens produced by D, while A has to fire only the necessary number of times to enable one firing of B. However, expressions f and g are required to know the number of tokens in channels $A \to B$ and $D \to E$ at the *beginning* of the current parameter reconfiguration. This depends on previous parameter values and involves dynamic scheduling.

For well-parenthesized graphs, where parameters on each path occurs first as an output rate then as an input rate, a quasi-static schedule always exists. Consider, for instance, a well-parenthesized version of the VRDF graph of Fig. 5 :



Its repetition vector is $[6, 2, 2p, 2p, 2, 1]$ and a valid quasi-static schedule is:

$$\left(A^6(\uparrow_p B\ (\uparrow_q C\ D)^p\ E)^2\right)^\omega$$

This observation holds for any acyclic VRDF graph where every maximal path is well-parenthesized.

Variable-Rate Phased Dataflow (VPDF)

[Wiggers et al. 2011] adapts the main VRDF ideas to CSDF. The resulting MoC is called Variable-Rate Phased Dataflow (VPDF). In this model, each actor could have many (but fixed and finite) *phases* (*i.e.,* cyclic). The sequence of phases is repeated periodically as in CSDF. Production and consumption rates of an actor during one phase can be parametric as in VRDF. The novelty is that each phase of a VPDF actor can be repeated a parametric number of times.

For instance, if the production rate of an actor A is annotated with $[n \star p, m \star 2]$, then this means that A has two phases and it produces p tokens in the first phase and 2 tokens in the second phase. Moreover, each cycle consists of repeating the first phase for n times followed by repeating the second phase for m times.

In addition to the constraints on parameters usage defined in VRDF, VPDF imposes more restrictions, for instance,

— a rate of a phase is parameterized with a different parameter than that of the repetition count of the phase;
— every parameter is associated with a single phase;
— a parameter is assigned one value per iteration through all phases of the actor.

This last constraint is a major difference with VRDF where a parameter can change at each firing of its modifier.

6. SCHEDULABLE PARAMETRIC DATAFLOW (SPDF)

The schedulable parametric dataflow model (SPDF) was proposed in [Fradet et al. 2012]. It can be seen as PRDF with the following extensions:

• Rates are products of positive integers and parameters. They are described by the following grammar:

$$R ::= k \mid p \mid R_1 R_2 \text{ where } k \in \mathbb{N}^* \text{ and } p \in \mathcal{P} \tag{3}$$

A parameter p takes its values in the interval $[1..p_{\max}]$. Since parameters cannot be null, SPDF is a dynamic rate model but not a dynamic topology model. Finally, an SPDF graph is strongly consistent iff it is consistent.

• Parameter reconfiguration in SPDF is not restricted to iteration boundaries. Each parameter p in an SPDF graph is associated with a unique modifier actor. The annotation $p@\pi$ indicates that B is the modifier of parameter p with a *modification period* π. It states that p can be modified every π firings of B.

Periods may be parametric and share the same grammar as rates. Modifications can be seen to be performed at boundaries of local iterations. These inner iterations are *implicit* in SPDF, whereas PSDF requires an *explicit* hierarchical decomposition to model them.

Fig. 6(a) depicts a simple SPDF graph where A is the modifier of p with modification period p, which means that if the value of p is set by the n^{th} firing of A to be p_1, then the next value of p will be set by the $(n + p_1)^{th}$ firing of A.

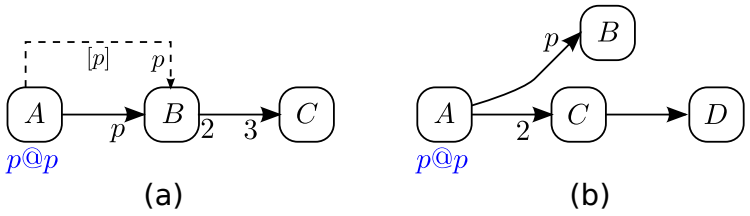


Fig. 6. (a) A period safe SPDF graph. (b) A rejected SPDF graph.

Due to this reconfiguration mechanism, rate consistency alone is not sufficient to guarantee boundedness. The following restrictions on modification periods of parameters are needed to ensure that an iteration returns the graph to its initial state and therefore boundedness.

A parameter p is modified at the boundaries on the local iterations of the subgraph *influenced* by p. This subgraph, called *region* of p and denoted $\mathit{reg}(p)$, is defined by:

$$\mathit{reg}(p) = \{C, D | C \xrightarrow{r \;\; r'} D \in E \;\wedge\; p \in r \vee p \in r' \vee p \in z_C \vee p \in z_D\}\}$$

Let B be the modifier of a parameter p and π be its modification period. The number of modifications during one iteration (also called the modification *frequency*: $\mathit{freq}(p) = z_B/\pi$) must be a positive integer. The produced values of p will be broadcast to all actors influenced by parameter p (the region of p).

Let us consider first SPDF graphs with a single parameter p. If the repetition solutions of all actors in the region can be factorized by $\mathit{freq}(p)$, then for each modification of the parameter, the sub-graph formed by actors in the region performs a *local iteration* and returns to its initial state, and waits for the next value of p. For the SPDF graph in Fig. 6(a), the repetition vector is $\vec{z} = [3p, 3, 2]$. So, $\mathit{reg}(p) = \{A, B\}$ and $\mathit{freq}(p) = 3$. The graph is data safe because the solutions of A and B in the repetition vector can

be factorized by *freq*(p) as $(A^pB)^3C^2$. Hence, after p firings of A, the sub-graph formed by A and B returns to its initial state. Let p_1, p_2, and p_3 be the first three values of p, then the first global iteration is actually $(\uparrow_p A^{p_1}B)(\uparrow_p A^{p_2}B)(\uparrow_p A^{p_3}B)C^2$ and not $(\uparrow_p A^{3p}B^3C^2)$.

The factorization by the frequency is a sufficient condition for boundedness (the graph returns to its initial state after a global iteration) but not a necessary condition. For example, the SPDF graph in Fig. 6(b) is not data safe because $\vec{z} = [2p, 2, p, p]$ and *reg*$(p) = \{A, B, C, D\}$, but the repetition solutions of these actors cannot be factorized by *freq*$(p) = 2$. However, there is definitely a bounded execution of this graph (*i.e.,* an infinite execution that uses only bounded channels).

If there are several parameters in the graph, their regions may interleave. [Fradet et al. 2012] imposes that if two regions *reg*(p) and *reg*(q) interleave (*i.e.,* their intersection is not empty but one region does not include the other) then the repetition solutions of actors in *reg*$(p) \cup$ *reg*(q) must be factorizable by *freq*$(p, q) =$ lcm(*freq*(p), *freq*(q)). Accordingly, the sub-graph formed by actors in *reg*$(p) \cup$ *reg*(q) will return to its initial state after *freq*(p, q)/*freq*(p) modifications of parameter p and *freq*(p, q)/*freq*(q) modifications of parameter q. So, the set of parameters is partitioned into sets of parameters such that two parameters p and q belong to the same partition if and only if the regions of p and q interleave but one region does not include the other. For a parameter partition P, let *reg*(P) denote the union of regions of all parameters in P, while *freq*(P) denote the least common multiplier (lcm) of frequencies of all parameters in P. The repetition solutions of all actors in *reg*(P) should be factorizable by *freq*(P).

Moreover, denoting M_q be the modifier of q, [Fradet et al. 2012] imposes that, if the repetition solution of M_q depends on *another* parameter p that belongs to a partition P, then *freq*(q) must be a multiple of *freq*(P). In order words, since M belongs to *reg*(P), any local iteration of actors in *reg*(P) must modify parameter q an *integer* number of times. This criterion ensures that every modifier is contained in at least one region whose local iterations never finish when a period of that modifier is not yet completed.

With these restrictions, an SPDF graph returns to its initial state after each global iteration.

As in VRDF, each parameter has a unique modifier. To model parameter communications, control channels are added between the modifier and the users (*i.e.,* actors that depend on the parameter), as illustrated in Fig. 6.(a). The reader may refer to [Fradet et al. 2012] for more details on how these channels are added. We only draw attention to the fact that control channels may create deadlocks and need to be considered in the liveness analysis. [Fradet et al. 2012] provides a conservative liveness analysis. An SPDF graph is live if the graph is acyclic when all *saturated* edges are removed, and that adding control channels for parameter communication to this acyclic graph does not create new cycles. As defined in Section 3, a saturated edge is an edge that contains enough initial tokens to fire its consumer for as many times as indicated by its repetition count (its solution).

All these restrictions do not only aim at ensuring boundedness and liveness of accepted SPDF graphs, but also at allowing the generation of quasi-static schedules. For the SPDF graph in Fig. 6.(a), one valid quasi-static schedule is:

$$\left((\uparrow_p A^pB)^3C^2\right)^\omega$$

However, due to parametric numbers of configuration changes (*e.g.,* imagine p changing every q firings with q changing every r firings, *etc.*), static analyses becomes intricate. Even an apparently basic analysis like expressing symbolically the number of tokens produced/consumed by actors during one global iteration can be quite involved.

7. BOOLEAN PARAMETRIC DATAFLOW (BPDF)

The Boolean parametric dataflow MoC (BPDF) has been proposed in [Bebelis et al. 2013; Bempelis 2015] as a simplified version of SPDF but extended to a dynamic topology MoC. It can also be seen an extension of PRDF with Boolean parameters. Communication channels can be dynamically deactivated and activated using Boolean conditions attached to edges. The main characteristics of BPDF are:

• Rate functions are constant positive integers or products of positive integers with parameters, like in SPDF. However, BPDF rate reconfiguration may take place only at iteration boundaries. Therefore, a BPDF graph needs, by definition, to be consistent. Finally, a BPDF graph is strongly consistent iff it is consistent.

• Unlike VRDF, BPDF does not allow null rates. However, it provides a different mechanism to deactivate channels at run-time. Each BPDF channel is annotated by a Boolean condition. If the condition evaluates to false, then the channel is deactivated, meaning that the actors connected by that edge will not write or read on this channel until it is reactivated again. Boolean conditions are described by the grammar:

$$B ::= true \mid false \mid b \mid \neg B \mid B_1 \wedge B_2$$

where b denotes a Boolean parameter in a finite set of Boolean parameters $\mathcal{P}_b$. Boolean parameters make BPDF a dynamic topology model. Compared to null rates, this mechanism has the following advantages:

— A clean separation between rates and channel conditions facilitate analyses such as boundedness. For instance, strongly consistent VRDF graphs are only bounded if strict restrictions are imposed on parameters usage because of special cases that arise from null rates. In contrast, BPDF restrictions are easier to understand by designers and programmers.
— Boolean conditions are a less coarse-grained abstraction than null rates. Indeed, a Boolean parameter can be used in the conditions of several channels, which make deactivation relations explicit in the model. For instance, a mutual exclusion between two channels can be made explicit by annotating them with conditions b and $\neg b$ respectively. Such information can be exploited by analyses for a better accuracy (liveness for instance).

• Boolean reconfiguration is not restricted to iteration boundaries. As for SPDF rate parameters, each Boolean BPDF parameter b is associated with a *unique* modifier. Upon firing, the modifier produces a new value of b, which is propagated to every actor that consumes (resp. produces) from (resp. on) a channel whose condition depends on parameter b. These actors are called the *users* of b and are denoted by $Users(b)$. The set of users of a parameter is closely related to the notion of regions in SPDF.

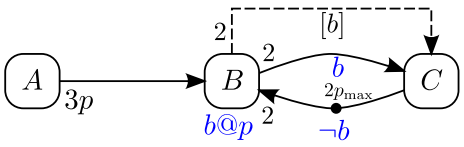


Fig. 7. A BPDF graph.

Fig 7 shows a BPDF graph that uses one integer parameter p and one Boolean parameter b (the dashed edge will be explained later). As in SPDF, the annotation "$b@p$" indicates that a new value of b is produced each p firings of B. So, the n^{th} value of b

is produced by the $(1 + (n-1)p)^{th}$ firing of B. A period may be parametric, with the same grammar as in SPDF. We impose some restrictions on the values of periods to ensure boundedness. Let $\mathit{freq}(b)$ denote the frequency of modification of the parameter b, *i.e.,* the number of produced values of b per iteration. If B is the modifier of b and π is its period, then:

$$\mathit{freq}(b) = \frac{z_B}{\pi} \tag{4}$$

The first restriction is that $\mathit{freq}(b)$ must be an integer (*i.e.,* z_B is a multiple of π). The second restriction is that any BPDF graph must be *period safe* [Bempelis 2015], *i.e.,* for all $b \in \mathcal{P}_b$ and for all $X \in \mathit{Users}(b)$, there exist $k \in \mathbb{N}$ such that $z_X = k.\mathit{freq}(b)$.

Intuitively, $\mathit{freq}(b)$ values of b are produced during each iteration. These values are broadcast to all users. Period safety ensures that each user consumes all the $\mathit{freq}(b)$ values of b in the same iteration. This can be checked with a simple factorization of the repetition vector. For instance, the repetition vector of the BPDF graph in Fig. 7 is $\vec{z} = [1, 3p, 6p]$. The factorization $A(B^pC^{2p})^3$ shows that a period equal to p is safe ($\mathit{freq}(b) = 3$): B will produce a new value b after p firings and C will read it every $2p$ firings. Actually, a period equal to 1 would also be valid ($\mathit{freq}(b) = 3p$), as can be seen by the factorization $A(BC^2)^{3p}$.

Now an important property is that any rate consistent and period safe BPDF graph is bounded [Bempelis 2015]. Intuitively, if a channel $A \xrightarrow{p\ \ q} B$ is annotated by a Boolean condition C, then the number of produced tokens by A during one iteration will be equal to the number of tokens consumed by B in the same iteration, whatever the values of C. The number of firings of A and B do not depend on C but only on whether they write/read on that channel. Actually, when an actor becomes completely disconnected, it continues to fire without producing or reading anything. For MoCs that allow null rates instead, boundedness analysis is more complicated since these models do not impose that production and consumption rates of a channel are both null or both not null, *i.e.,* a channel can be *partially* deactivated. This issue does not arise in BPDF thanks to the Boolean conditions.

The consistency of BPDF graphs, as defined in [Bempelis 2015], is *agnostic* to Boolean conditions; *i.e.,* the symbolic repetition vector is computed assuming that all channels are activated. For instance, if channel $C \rightarrow B$ in Fig. 7 was $B \xrightarrow{2\ \ 1} C$, the graph (without considering the dashed edge) would still be rate consistent because the cycle between B and C is actually a false cycle (mutual exclusive conditions).

• As in PRDF, the integer parameter communication of BPDF is left unspecified. However, the Boolean parameter communication is, as in VRDF, implemented by adding special control channels from modifiers to users (the dashed edges in Fig. 7). These channels do not contain any initial tokens. The rates of these channels do not jeopardize the consistency of the graph but control channels may create deadlocks. Liveness analysis must hence consider the graph augmented with control channels. The algorithm described in Section 3.2 is actually a simplified version of the conservative liveness analysis proposed for BPDF graphs. It assumes that all channel conditions are true and tries to find a schedule using abstract execution. In case of failure, the algorithm uses clustering and finally considers Boolean conditions to detect and to take into account false cycles.

A scheduling framework for BPDF graphs was proposed in [Bebelis et al. 2014]. It takes as input a BPDF graph and a set of constraints aimed at minimizing timing, buffer sizes, power consumption, or other criteria. The BPDF graph is executed according to a *slotted scheduling* policy, such that each actor firing occurs inside a single slots and cannot spread over more than one (but of course several actor firings can

occur inside the same slot). At the beginning of a slot, enabled actors are first filtered *w.r.t.* to the constraints. The selected firings will be performed, each on a different processing unit. When all these firings are completed, the process is repeated for the next slot, and so on. The enabling along with the constraints are pre-compiled as much as possible in order to reduce the scheduling runtime overhead and, in some cases, to produce a quasi-static schedule.

8. PARAMETERIZED AND INTERFACED SYNCHRONOUS DATAFLOW (PiSDF)

Parameterized and interfaced synchronous dataflow (PiSDF) has been proposed in [Desnos et al. 2013] as an extension of SDF with parameters and hierarchy. As PSDF, the corresponding parametrization framework (PiMM) can be applied to any dataflow model with a clear definition of iterations. PiSDF shares many features with PSDF, in particular it inherits its quasi-static scheduling algorithm for a sub-class of graphs. Since PiSDF is closely related to PSDF, we express the graph taken for PSDF (Fig. 4) as a PiSDF graph (Fig. 8). The main differences are:

• Hierarchy is simpler in PiSDF than in PSDF. PiSDF does not distinguish between *init*, *subinit* and *body* graphs. The content of hierarchical actor needs to execute for one iteration each time the actor is fired.

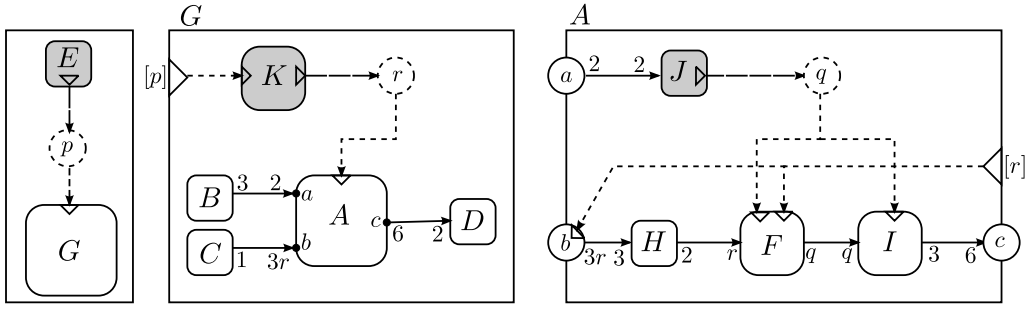


Fig. 8. A PiSDF graph.

As illustrated in Fig.8, parameters are represented by special vertices, and parameter dependencies are represented by dashed arrows. Configuration actors which set parameters are represented in grey. Configuration ports (receiving parameter values) are denoted by triangles ((*e.g.,* $[r]$ in Fig.8) whereas data ports are circles (*e.g.,* a, b, c in Fig.8). A parameter should have a fixed value during the firing of the hierarchical actor to which it belongs. This value can be set by either a configuration actor or another parameter (*e.g.,* a derivative parameter). In Fig. 8, parameters r and q belong to actors G and A receptively. Parameter r is set by the configuration actor K.

As in PSDF, no grammar for parameters is given and null rates seem allowed. PiSDF provides also static parameters, *i.e.,* parameters that are configured only once at the beginning of the execution, and never change after that. So, these parameters do not depend on an actor for reconfiguration.

• PSDF defines an order of execution between the three (*init*, *subinit* and *body*) graphs, which prevents parameter dependencies from creating deadlocks. PiSDF follows the same approach and impose some restrictions on configuration actors. First, a configuration actor fires exactly once per firing of the hierarchical actor to which it belongs, and before any non-configuration actor. Moreover, it can consume data tokens only

from interface input ports of hierarchical actors (*e.g.,* see actor J in Fig. 8). Furthermore, parameters of a configuration actor should be either static (*i.e.,* defined at the beginning of the whole execution) or defined in higher hierarchical levels. So, configuration actors actually model the *subinit* graph in PSDF MoC. Nevertheless, PiSDF allows configuration actors to communicate with the non-configuration actors using ring-buffers (explained later).

• The semantics of interface input and output ports of hierarchical actors in PiSDF is different from their semantics in PSDF. In this latter MoC, these ports do not play any role but to conducts tokens from one hierarchy level to another. Indeed, as the production and consumption rates of a PSDF hierarchical actor are inferred from its internal implementation, there is a perfect correspondence between these rates and the functionality of the internal implementation. This is not always the case in PiSDF as production and consumption rates of a hierarchical actor need to be provided by the designer in the parent graph of the actor. This implies that the parent graph can be scheduled without taking into account the implementation of hierarchical actors. This favors compositionality, but requires interface ports to ensure the correspondence between the external rates and the internal functionality of a hierarchical actor.

The behavior of interface ports is that of interface ports in the Interface-Based SDF (IBSDF) MoC [Piat et al. 2009]. We first precise that interface ports are taking into account when computing the repetition vector of the graph inside a hierarchical actor. This ensures that all produced tokens by an interface input port are consumed by an iteration of the graph, and that an iteration produces enough tokens for an interface output port. Suppose that both the consumption rate of H and the production rate of I in A are equal to 2. The new repetition vector of non-configuration actors will be $[z_H = 3r, z_F = 6, z_I = 6]$. This means that H consumes $6r$ tokens per iteration, while the interface input port reads only $3r$ tokens from the outside world. Therefore, the interface port will behave as a ring buffer, and the $3r$ tokens are read twice per firings of H. Similarly, I produces 12 tokens, while the interface output port provides only 6 tokens to the outside world. Therefore, it behaves as a ring buffer and writes only the *last* 6 tokens to the outside world.

9. FSM-BASED PARAMETERIZED SCENARIO-AWARE DATAFLOW (PFSM-SADF)

The FSM-based parameterized scenario-aware dataflow model, PFSM-SADF, was proposed in [Skelin et al. 2015; 2015] as a parameterization of FSM-SADF. The main objective is to define an expressive parametric MoC whose throughput can be statically analyzed. A PFSM-SADF graph consists of a finite set of scenarios and a nondeterministic finite-state machine that specifies the transitions between scenarios. Each scenario can be seen as a PRDF graph with the following differences:

• Rate functions are positive constant integers or products of positive constant integers with positive integer parameters (as in BPDF Sec. 7).

• For the sake of throughput analysis, approximate parametric execution times are introduced. An actor execution time is an affine function described by the grammar:

$$D ::= k \mid k \cdot p \mid D_1 + D_2 \text{ where } k \in \mathbb{R}^+ \text{ and } p \text{ is a positive real parameter}$$

• Restrictions are imposed for strong consistency and a better analyzability:

(1) Self-loops must contain exactly one token.
(2) If a channel $A \xrightarrow{r_1 \ r_2} B$ contains initial tokens, then the solution of the consumer z_B must be constant (*i.e.,* non parametric) and the edge must be saturated (*i.e.,* its number of initial tokens must be greater than $\max(r_2) \cdot z_B$).

(3) The graph must be compile-time schedulable and must have a single appearance quasi-static schedule of the form $X^{z_X} Y^{z_Y} \ldots$ This entails that the graph obtained by removing the saturated edges must be acyclic.

A PFSM-SADF graph is a tuple (S^p, F^p) where S^p is a finite set of parameterized scenarios and F^p is an FSM over S^p. Each state of the FSM is labeled by a scenario. At the end of each iteration of the graph, a transition is *non-deterministically* chosen from the current FSM state, and a configuration in the scenario of the next state is also *non-deterministically* chosen. So, in comparison with PRDF, two choices are made at boundaries of iterations: the next scenario and the next configuration in that scenario.

Fig. 9 shows a simple PFSM-SADF graph with two scenarios S_1 and S_2. Its FSM contains two states such that the initial state is labeled by scenario S_2. Scenarios may share actors and channels: functionally, this means that tokens produced in one scenario may be used in another scenario. For instance, channel $A \rightarrow B$ is shared between S_1 and S_2, hence tokens present in channel $A \rightarrow B$ at the end of the iteration in scenario S_1 are used in scenario S_2. In this case, the number of initial tokens in a shared channel must be identical in all the concerned scenarios (recall that after each iteration, the channels return to their initial states).

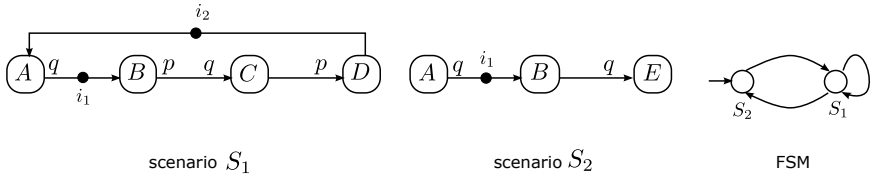


Fig. 9. A PFSM-SADF graph.

Worst-case throughput analysis

A parametric worst-case throughput analysis of PFSM-SADF graphs is proposed in [Skelin et al. 2015; 2014], assuming a self-timed execution. It is based on a max-plus characterization of the graphs. Max-plus algebra has also been used to compute the throughput of SDF and SADF graphs (*e.g.,* [Geilen 2011]).

As defined in [Geilen and Stuijk 2010], the throughput of an FSM-SADF graph is the worst-case (*i.e.,* the smallest) number of completed iterations per time unit that can be achieved by any possible long-run scenario sequence. [Geilen and Stuijk 2010] proposed two techniques for computing the throughput. The first one is based on state space exploration, while the second one is based on max-plus automata theory. This latter technique is extended in [Skelin et al. 2015] to PFSM-SADF.

Using techniques from [Geilen and Stuijk 2010], a worst-case throughput of the PFSM-SADF graph can be computed. For instance, the second method (*i.e.,* using max-plus automata theory) constructs the throughput graph such that the inverse of the maximum-cycle mean (MCM) of this graph is equal to the worst-case throughput of the PFSM-SADF graph. The main idea is to produce and to solve a system of parametric recurrence equations representing timing invariants of iterations. [Skelin et al. 2014] proposed a method, called backward substitution, to solve such systems. Unfortunately, this method does not work in the general case because of parametric ceiling terms. The method is corrected in [Skelin et al. 2015] thanks to conservative approximations of parametric ceiling functions.

Parameterized Set of Modes Core Functional Dataflow (PSM-CFDF)

A similar approach as been applied to CFDF [Hsu et al. 2005]. CFDF actors have sets of modes, each mode fixing constant production and consumption rates. However,

Table I. Characteristics of Parametric Dataflow MoCs

MoC	*Parameters*	*Par. Functions & Range*	*Config. Changes*	*Symbolic Analyses*
PSDF	Rates	$\mathbb{N}^*$	Hierarchical iterations	*Sc*
	Init. tokens	Unspecified		
VRDF	Rates	$R ::= k \mid p \quad p \in \mathbb{N}$	Each firing	*Bu*
SPDF	Rates	$R ::= k \mid p \mid R_1.R_2 \quad p \in \mathbb{N}^*$	Inner iterations	*Co, Li, Sc*
BPDF	Rates	$R ::= k \mid \ p \mid R_1.R_2 \quad p \in \mathbb{N}^*$	Global iterations	*Co, Li, Sc*
	Edges	$B ::= b \mid \neg B \mid B_1 \wedge B_2$	Inner iterations	
PiSDF	Rates	$\mathbb{N}$	Hierarchical iterations	*Sc*
PFSM-SADF	Rates	$R ::= k \mid p \mid R_1.R_2 \quad p \in \mathbb{N}^*$	Global iterations	*Th, La*

Co: Consistency; *Li*: Liveness; *Th*: Throughput; *Bu*: Buffer size; *La*: Latency ; *Sc*: Scheduling

as the number of modes grows, CFDF formulations can become clumsy. PSM-CFDF was proposed in [Lin et al. 2015] as a parametric extension of CFDF. Parametrization allows to regroup modes in sets called parametrized sets of modes (PSMs). Modes in each PSM can be concisely described using parameters.

10. SUMMARY AND COMPARISONS

This section summarizes the main features of parametric MoCs and compares them using a common example.

Table I presents the main characteristics of the MoCs detailed before. The absence of a specific analysis in a model only means that it was not considered in the related work. In many cases, symbolic analyses developed for some MoCs apply directly or can be adapted to others. Parameters are most often changed by actors (modifiers) and transmitted via dataflow channels. Other options could be considered, like changes by a global controller or communication using shared data.

We consider a common example, inspired from video codecs, including several forms of dynamicity. We express it in the different MoCs to illustrate their differences as well as their respective advantages and drawbacks. The structure of the example is depicted in Fig. 10(a). At each firing, B (resp. D and F) produces (resp. consume) a variable number of tokens $q \in \{1, ..., Q\}$. Because of the edge $A \xrightarrow{2\ 1} B$, actor B fires twice in each iteration, hence potentially with two different values of q. Furthermore, the two pipelines starting from B are executed *exclusively*: either the C/D pipeline ($B \longrightarrow C \longrightarrow D \longrightarrow F$) or the E pipeline ($B \longrightarrow E \longrightarrow F$).

• A possible implementation of this example in SDF is presented in Fig. 10(b). Since SDF is a static model, dynamicity is encoded within actors and exchanged data. For rates, the SDF version always uses the maximum rate Q and both pipelines are always active. Additional communication links from B to actors C, D, E, and F (dashed arrows) are used to inform the different actors of (i) the significant number q of tokens

within the Q tokens exchanged, and (ii) which pipeline is active[6]. The iteration is now:

$$AB^2C^{2Q}D^2E^{2Q}F^2 \tag{5}$$

For instance, if the current rate is q and only the C/D pipeline is enabled, B sends this information to actors C, D, E, F, plus q valid tokens within Q to C and Q (possibly dummy) tokens to E. C fires Q times and D selects the significant q tokens among the Q tokens sent by C. Actor F discards the Q tokens sent through the "disabled" pipeline.

This encoding has several drawbacks. First, at each firing, B sends Q tokens to C and E, which are both fired Q times regardless of the actual value of q. Besides, actors C, D, and E are fired at each iteration while the specification states that their execution depends on which pipeline is enabled. Such an encoding implements the worst case and is likely to be extremely inefficient. One may think to optimize the implementation by exchanging only the meaningful number of tokens q. However, in this case the actual rates would be encoded and hidden within dynamic data, which would prevent consistency and performance analyses.

• Fig. 10(f) and (g) depict implementations of the common example in PSDF and PiSDF, respectively. Hierarchy is used to make the subgraph made of actors B, C, D, E, and F (*i.e.,* the body of the macro actor X) to operate with two different values of q within the same iteration. In PSDF, this is achieved thanks to the $X.subinit$ actor that assigns a new value to q and then propagates it to actors B, D, and F. The two pipelines C/D and E must be executed at *each* iteration. It follows that the iteration is:

$$AX^2 \quad \text{with} \quad X = (\uparrow_q BC^qDE^qF) \tag{6}$$

A solution similar as the one used for SDF must be implemented to achieve the dynamic behavior of the two pipelines. Even if not many static analyses have being proposed for PSDF, a few restrictions (*e.g.,* on initial tokens) should allow the symbolic analyses presented in Section 3 to be reused.

In PiSDF, operating with two different values of q is achieved using actors X and I. PiSDF, which allows null parameters, can implement the exclusive choice between the two pipelines by introducing an additional parameter $a \in [0, 1]$ such that if $a = 1$ the C/D pipeline is executed (hence the rates $q.a$ and a in Fig 10(g)), while if $a = 0$ the E pipeline is executed (hence the rates $q.(1-a)$ and $(1-a)$ in Fig 10(g)). It follows that the iteration for this PiSDF graph is:

$$AX^2 \quad \text{with} \quad X = (\uparrow_q\uparrow_a B(C^qD)^a(E^q)^{1-a}F) \tag{7}$$

Hierarchy improves expressivity (*e.g.,* actor sharing) and allows to model inner iterations (within hierarchical actors). Actually PSDF and PiSDF are best seen as a meta-model approach that can be applied to different MoCs. In particular, most of the other MoCs studied in this survey could be made hierarchical following that approach.

• VRDF imposes many constraints on the graph structure and parameters usage. It permits to change parameters at every single firing, which is much more flexible that inner iterations (hierarchical as in PSDF or region-based as in SPDF). Compared to other MoCs, parametric rates are restricted to be a single parameter. Rates such as $2pq$ must be simulated/encoded by introducing new parameters. The relations between such parameters are not explicit, which may preclude some verification and/or analyses. Fig. 10(d) depicts our solution of implementing the common example in VRDF. Parameters q_1 and q_2, both modified by B, implement the dynamic rate. Two parameters are needed since a given parameter can only appear as the production or consumption rate of *at most* two actors. Since parameters can take the value 0, one may

[6]A less explicit solution would encode this information within data tokens without using additional links.

consider to implement mutually exclusive pipelines by adding parametric rates: q_3 as both rates of C (consumption and production), q_4 for both rates of edge (D, F), and q_5 for both rates of E. Then, the exclusive execution of the two pipelines could be ensured provided that:

$$\underbrace{\big((q_3 = q_4 = 1) \wedge (q_4 = q_5 = 0)\big)}_{C/D \text{ pipeline}} \;\; xor \;\; \underbrace{\big((q_1 = q_2 = q_3 = 0) \wedge (q_5 = 1)\big)}_{E \text{ pipeline}} \tag{8}$$

However, ensuring this invariant would require extra communication between B and all the other parameter modifiers (*i.e.,* C, D, E), a solution comparable to the encoding used above for SDF.

At each firing, B reads a new value of q which is used to set both q_1 and q_2. The iteration is therefore:

$$A(\uparrow_q BC^q DE^q F)^2 \tag{9}$$

Unfortunately, the possibility of changing rates at each firing makes the design of precise static analyses for VRDF difficult.

• Fig. 10(e) depicts our solution of implementing the common example in SPDF. Here, B is the modifier of q and the "q@1" annotation indicates that q may change at each firing of B. For this example, the iteration is similar to that of VRDF:

$$A\Big(\uparrow_q B \underbrace{(C^q DE^q)}_{\text{inner iteration}} F \Big)^2 \tag{10}$$

Since SPDF does not provide any means to change the topology, a solution similar as the one used for SDF must be implemented to achieve the dynamic behavior of the two pipelines.

In SPDF, inner iterations may occur inside regions. Regions are more flexible than the fixed hierarchical decomposition of PSDF. For instance, it is possible to have overlapped regions whereas hierarchical actors must be manually described and properly nested. In this sense, SPDF is more expressive although, contrary to PSDF, it does not help the reuse of previously modeled graphs. However, parametric numbers of configurations within an iteration complicate analyses a great deal.

• Fig. 10(c) depicts a BPDF encoding of the common example. The two pipelines must be duplicated because a given parametric rate can only take one value during an iteration. However, BPDF can use two Boolean parameters, b_1 and b_2, to model the mutual exclusion in both pairs of pipelines. As a consequence, the iteration is:

$$A \uparrow_{q_1} \uparrow_{q_2} \big(\uparrow_{b_1} B_1\, b_1 ?\, (C_1^{q_1} D_1 F_1) : (E_1^{q_1} F_1)\big) \big(\uparrow_{b_2} B_2\, b_2 ?\, (C_2^{q_2} D_2 F_2) : (E_2^{q_2} F_2)\big) \tag{11}$$

The parametric rates of BPDF can only be changed between iterations. Compared to SPDF, the absence of inner iterations changing rates simplifies static analyses. On the other hand, BPDF allows inner iterations within regions but only for the Boolean parameters used in enabling conditions of communication channels. This is less problematic since the number of firings is fixed at the beginning of the iteration (even though the firings of completely disconnected actors are dummy firings and can be suppressed in the implementation). They have little impact on analyses, which remain close to those presented in Section 3. In particular, Boolean conditions can often be expanded into a small set of possible configurations. Compared to null rates, Boolean conditions express more easily relations between different channels than in VRDF (*e.g.,* mutual exclusion) and, contrary to PiSDF, they can change within iterations. Actually, the addition of such Boolean conditions could be considered to extend most of the other parametric MoCs presented in this survey.

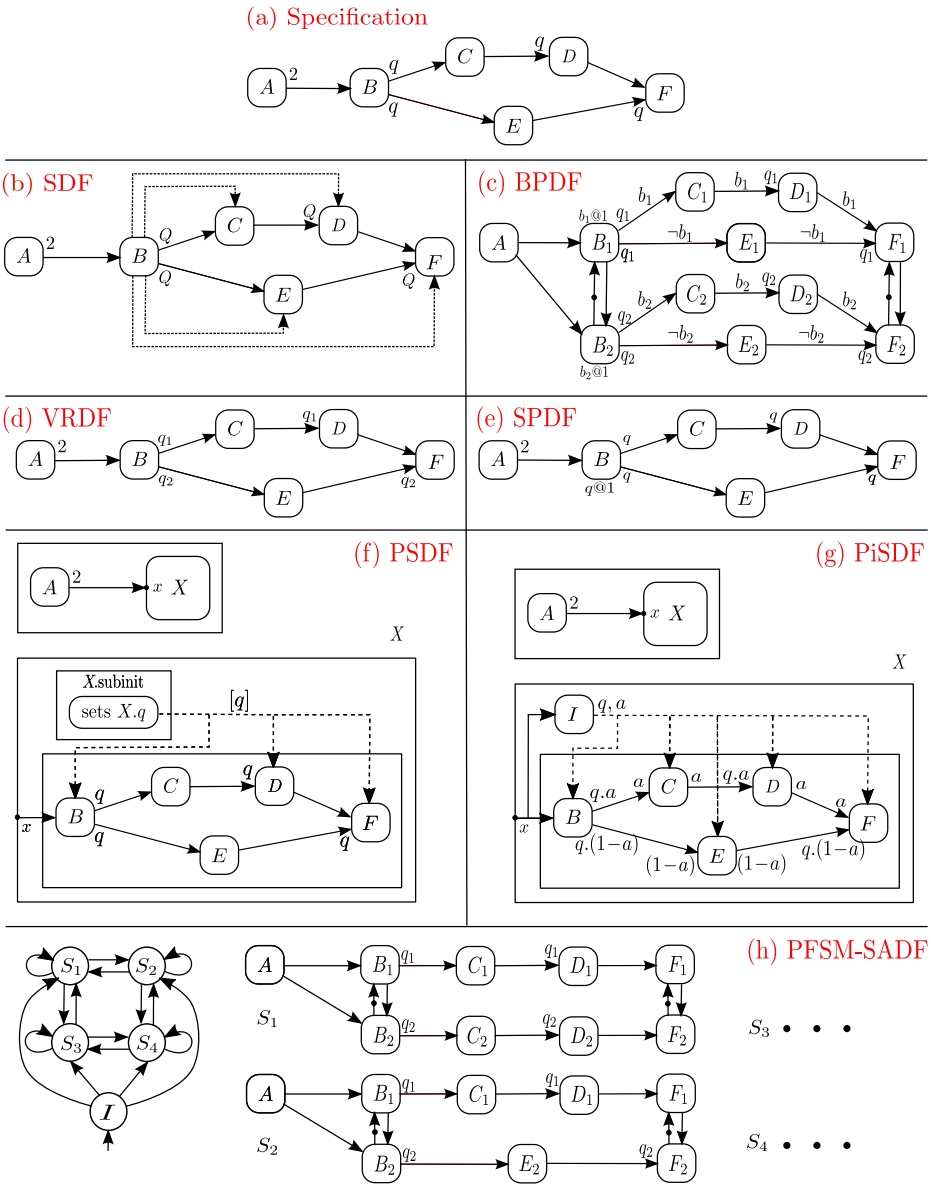


Fig. 10. A common example expressed in all the parametric MoCs.

• Finally, PFSM-SADF combines scenarios with parametric rates changed between iterations only. Fig. 10(h) depicts a solution for the common example in PFSM-SADF. It consists of four scenarios, S_1 to S_4, and one finite state machine (on the left), which selects the active scenario. Each scenario S_i duplicates the graph in the same way as BPDF because PFSM-SADF does not allow rates to take multiple values within a given iteration. For instance, scenario S_1 executes the C/D pipeline with q_1 and then the C/D pipeline with q_2, yielding the following iteration:

$$S_1 ::= A \uparrow_{q_1} \uparrow_{q_2} (B_1 C_1^{q_1} D_1 F_1)\ (B_2 C_2^{q_2} D_2 F_2) \tag{12}$$

The iteration is similar for scenario S_2, which executes the C/D pipeline with q_1 and then the E pipeline with q_2, and so on for S_3 and S_4.

Static analyses of Section 3 could be applied individually to each PFSM-SADF scenario. Compared to BPDF, PFSM-SADF can only activate or deactivate channels at the iteration boundaries by changing scenarios. The relationship between configurations (*e.g.,* that the activation of two channels/paths are mutually exclusive) must be encoded manually. On the other hand, scenarios are also more flexible and handy to describe very different (or unrelated) graphs. The notion of scenarios could be used to equip other MoCs with dynamic topology features.

Each parametric MoC is more expressive than SDF, because parameters allow dynamic changes of rates (while maintaining consistency). However, in terms of expressivity, there is no clear order between the different parametric MoCs presented in this survey. A first subclass of MoCs allows parameter changes within iterations: PSDF and PiSDF do it by using hierarchy, SPDF uses the more flexible notion of regions, and VRDF even allows changes every firing. PSDF, SPDF and VRDF can be regarded as increasingly expressive but their analyzability follows the inverse order. Furthermore, the strong syntactic constraints enforced by VRDF complicates the comparison. A second subclass of MoCs allows dynamic changes of the dataflow graph: BPDF does it by using Boolean parameters on edges, and PFSM-SADF by using scenarios. BPDF is more expressive since, thanks to parameters, it can represent concisely large sets of graphs. On the other hand, PFSM-SADF looks better suited for representing small sets of very different graphs.

In conclusion, each model has its own benefits and drawbacks and any advice or choice depends on the target application. However, hierarchy, Boolean parameters and scenarios can be used to extend most MoCs while being compatible with static analyses. Therefore, if the goal is to improve expressivity while preserving analyzability, then we believe that BPDF extended with hierarchy and scenarios would be a fine trade-off.

REFERENCES

Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. 1996. *Software synthesis from dataflow graphs*. Kluwer Academic Publishers, Norwell, MA, USA.

Vagelis Bebelis, Pascal Fradet, and Alain Girault. 2014. A Framework to Schedule Parametric Dataflow Applications on Many-core Platforms. *SIGPLAN Not.* 49, 5 (2014), 125–134.

Vagelis Bebelis, Pascal Fradet, Alain Girault, and Bruno Lavigueur. 2013. BPDF: A Statically Analyzable DataFlow Model with Integer and Boolean Parameters. In *Proceedings of the 11th ACM International Conference on Embedded Software*. 3:1–3:10.

Evangelos Bempelis. 2015. *Boolean Parametric Data Flow: Modeling - Analysis - Implementation*. Ph.D. Dissertation. Université Grenoble Alpes.

Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya. 2001. Parameterized Dataflow Modeling for DSP Systems. *Trans. Sig. Proc.* 49, 10 (2001), 2408–2421.

Shuvra S. Bhattacharyya, Ed F. Deprettere, and Bart D. Theelen. 2012. Dynamic dataflow graphs. In *Handbook of Signal Processing Systems* (2nd ed.). Springer.

G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. 1996. Cyclo-static dataflow. *IEEE Trans. Sig. Proc.* 44, 2 (February 1996), 397–408.

Adnan Bouakaz, Pascal Fradet, and Alain Girault. 2016. Symbolic Buffer Sizing for Throughput-Optimal Scheduling of Dataflow Graphs. In *Proceedings of the 2016 IEEE 22nd Real-Time and Embedded Technology and Applications Symposium*.

Joseph T. Buck. 1993. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.

Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra S Bhattacharyya, and Slaheddine Aridhi. 2013. PiMM: Parameterized and Interfaced dataflow Meta-Model for MPSoCs runtime reconfiguration. In *13th Int. Conf. on Embedded Computer Systems: Architecture, Modeling and Simulation*. 41 – 48.

Johan Eker and Jörn W. Janneck. 2003. *CAL Language Report*. Technical Report UCB/ERL M03/48. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2003/4186.html

Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia R. Sachs, and Yuhong Xiong. 2003. Taming heterogeneity - the Ptolemy approach. *Proc. IEEE* 91, 1 (2003), 127–144.

Pascal Fradet, Alain Girault, and Peter Poplavko. 2012. SPDF: A Schedulable Parametric Data-flow MoC. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 769–774.

Marc Geilen. 2011. Synchronous Dataflow Scenarios. *ACM Trans. Embed. Comput. Syst.* 10, 2 (2011), 16:1–16:31.

Marc Geilen and Sander Stuijk. 2010. Worst-case Performance Analysis of Synchronous Dataflow Scenarios. In *IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis*. 125–134.

Soonhoi Ha and Hyunok Oh. 2012. Decidable dataflow models for signal processing: synchronous dataflow and its extensions. In *Handbook of Signal Processing Systems* (2nd ed.). Springer.

Chia-Jui Hsu, Ming-Yung Ko, and Shuvra S. Bhattacharyya. 2005. Software Synthesis from the Dataflow Interchange Format. In *Proc. of the Workshop on Software and Compilers for Embedded Systems*. 37–49.

Gilles Kahn. 1974. The semantics of a simple language for parallel programming. In *IFIP Congress*. 471–475.

Hojin Kee, Chung-Ching Shen, Shuvra S. Bhattacharyya, Ian C. Wong, Yong Rao, and Jacob Kornerup. 2012. Mapping Parameterized Cyclo-static Dataflow Graphs Onto Configurable Hardware. *J. Signal Proc. Syst.* 66, 3 (2012), 285–301.

Edward A. Lee. 2001. Computing for embedded systems. In *IEEE Instrumentation and Measurement Technology Conference*. 1830–1837.

Edward A. Lee and David G. Messerschmitt. 1987. Synchronous data flow. In *Proc. of the IEEE*. 1235–1245.

Edward A. Lee and Thomas M. Parks. 2002. Dataflow process networks. In *Readings in hardware/software co-design*. Kluwer Academic Publishers, Norwell, MA, USA, 59–85.

Shuoxin Lin, Lai-Huei Wang, Aida Vosoughi, Joseph R. Cavallaro, Markku J. Juntti, Jani Boutellier, Olli Silvén, Mikko Valkama, and Shuvra S. Bhattacharyya. 2015. Parameterized Sets of Dataflow Modes And Their Application to Implementation of Cognitive Radio Systems. *J. Signal Process. Syst.* 80, 1 (2015), 3–18.

Jonathan Piat, Shuvra S. Bhattacharyya, and Mickaël Raulet. 2009. Interface-based hierarchy for synchronous data-flow graphs. In *IEEE Workshop on Signal Processing Systems*. 145–150.

William Plishker, Nimish Sane, Mary Kiemb, Kapil Anand, and Shuvra S. Bhattacharyya. 2008. Functional DIF for Rapid Prototyping. In *Proc. of the Int. Symp. on Rapid System Prototyping*. 17–23.

Mladen Skelin, Marc Geilen, Francky Catthoor, and Sverre Hendseth. 2014. Worst-cas throughput analysis for parametric rate and parametric actor execution time scenario-aware dataflow graphs. In *International Workshop on Synthesis of Continuous Parameters*. 65–79.

Mladen Skelin, Marc Geilen, Francky Catthoor, and Sverre Hendseth. 2015. Parametrized dataflow scenarios. In *International Conference on Embedded Software, Emsoft'15*. 95–104.

Sundarara Sriram and Shuvra S. Bhattacharyya. 2000. *Embedded multiprocessors: scheduling and synchronization*. Marcel Dekker, Inc., New York, NY, USA.

Sander Stuijk. 2007. *Predictable mapping of streaming applications on multiprocessors*. Ph.D. Dissertation. Eindhoven University of Technology.

William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*. 179–196.

Stavros Tripakis, Dai Bui, Marc Geilen, Bert Rodiers, and Edward A. Lee. 2013. Compositionality in Synchronous Data Flow: Modular Code Generation from Hierarchical SDF Graphs. *ACM Trans. Embed. Comput. Syst.* 12, 3, Article 83 (April 2013).

Lai-Huei Wang, Chung-Ching Shen, and Shuvra S. Bhattacharyya. 2013. Parameterized core functional dataflow graphs and their application to design and implementation of wireless communication systems. In *IEEE Workshop on Signal Processing Systems*. 1–6.

Maarten Wiggers, Marco Bekooij, and Gerard Smit. 2011. Buffer Capacity Computation for Throughput-constrained Modal Task Graphs. *ACM Trans. Embed. Comput. Syst.* 10, 2 (2011), 17:1–17:59.

Maarten H. Wiggers, Marco J. G. Bekooij, and Gerard J. M. Smit. 2008. Buffer Capacity Computation for Throughput Constrained Streaming Applications with Data-Dependent Inter-Task Communication. In *Proc. of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*. 183–194.