SPDF: A Schedulable Parametric Data-Flow MoC (Extended Version)

Pascal Fradet, Alain Girault, Peter Poplavko

SPDF: A Schedulable Parametric Data-Flow MoC (Extended Version)

Pascal Fradet, Alain Girault, Peter Poplavko

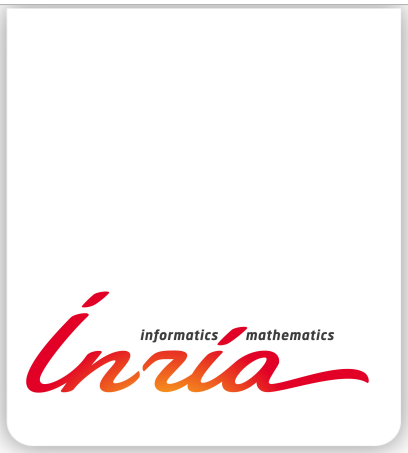
RESEARCH REPORT

N° 7828

December 2011

Project-Team Pop Art

ISSN 0249-6399 ISRN INRIA/RR--7828--FR+ENG



SPDF: A Schedulable Parametric Data-Flow MoC (Extended Version)

Pascal Fradet*, Alain Girault*, Peter Poplavko*†

Project-Team Pop Art

Research Report n° 7828 — December 2011 — 24 pages

Abstract: Dataflow programming models are suitable to express multi-core streaming applications. The design of high-quality embedded systems in that context requires static analysis to ensure the liveness and bounded memory of the application. However, many streaming applications have a dynamic behavior. The previously proposed dataflow models for dynamic applications do not provide any static guarantees or only in exchange of significant restrictions in expressive power or automation. To overcome these restrictions, we propose the *schedulable parametric dataflow (SPDF)* model of computation. We present static analyses and a quasi-static scheduling algorithm. We demonstrate our approach using a video decoder case study.

Key-words: dataflow programming, parametric rates, boundedness, liveness, quasi-static scheduling

---

This report is an extended version of

P. Fradet, A. Girault, and P. Poplavko, *SPDF: A Schedulable Parametric Data-Flow MoC.* Proc. DATE-2012, Design, Automation and Test in Europe. IEEE, 2012.

Besides minutes differences, it adds an appendix with descriptions and proofs omitted from the conference version.

* Inria Grenoble, Pop Art Team
† CRI-PILSI

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

SPDF: A Schedulable Parametric Data-Flow MoC

Résumé : Les modèles de programmation flot de données sont particulièrement bien adaptés aux traitements multimédia sur plateforme multicœurs. La conception de telles applications demande des analyses statiques pour s'assurer de propriétés telles que la vivacité et l'exécution en mémoire bornée. D'autre part, beaucoup d'applications multimédia modernes ont un comportement dynamique. Or, les modèles flot de données adaptés aux applications dynamiques n'offrent pas d'outils d'analyse statique ou seulement en imposant de fortes restrictions d'expressivité et/ou d'automatisation. Pour pallier ces inconvénients, nous introduisons SPDF (*Schedulable Parametric DataFlow*) un nouveau modèle flot de données acceptant des taux de production/consommation dynamiques mais pouvant être analysé et ordonnancé statiquement. Nous illustrons SPDF à l'aide d'une étude de cas: la modélisation d'un décodeur vidéo.

Mots-clés : programmation flot de données, taux paramétriques, mémoire bornée, vivacité, ordonnancement quasi statique

1 Introduction

Multi-core systems are becoming an increasingly important platform for many embedded system designs. To take advantage of multi-cores, programming languages should express thread-level parallelism. Among such languages, *dataflow* languages are prominent for many streaming applications [2].

Recent dataflow programming environments support applications whose behavior is characterized by dynamic variations in resource requirements. The high expressive power of the underlying models makes it challenging to ensure predictable behavior. For example, the CAL actor language [2] or Kahn Process Networks [6] can express many dynamic applications. However, checking *liveness* (*i.e.,* no part of the system will deadlock) and *boundedness* (*i.e.,* can be executed in finite memory) is known to be hard or even undecidable.

This situation is troublesome for the design of high-quality embedded systems. Sufficient criteria for liveness and boundedness have been formulated for less expressive models, which can nevertheless express the core part of many streaming applications. However, such *statically analyzable* criteria come at the cost of significantly constraining modeling and scheduling. For example, parametrical synchronous dataflow (PSDF) [1] imposes a hierarchical discipline which restricts scheduling and analysis.

In this paper, we introduce the *schedulable parametric dataflow (SPDF)* model of computation (MoC) for dynamic streaming applications. SPDF was designed to be statically analyzable for liveness and boundedness, while avoiding the aforementioned restrictions of PSDF and certain essential restrictions of other related models.

The article is organized as follows. In Section 2, we present a well-known basic model – synchronous dataflow (SDF) [7] – which is easily analyzable but restricted to static applications. We then introduce our SPDF model as a parametric and dynamic extension of SDF. In Section 3, we present the static analyses for boundedness and liveness. Section 4 describes compilation, such as the insertion of parameter distribution network and quasi-static scheduling. A video decoding application is presented as a case study in Section 5. Section 6 summarizes our contribution, compares it to related work and hints at future research directions. The appendix describes in more details the implementation of parameter communication and sketches the proofs of the main properties.

2 Model of Computation

We start from SDF – synchronous dataflow [7] – one of the simplest dataflow MoC. Then, we present our MoC (SPDF) as a statically analyzable extension of SDF with dynamic parametrization.

2.1 Basic Model: SDF

In SDF, a program is defined by a directed graph, where nodes – called *actors* – are functional units. The actors have *data ports* connected by *edges* which can be seen as FIFO (first-in first-out) channels. The atomic execution of a given actor – called *actor firing* – consumes data tokens from its incoming edges (its *inputs*) and produces data tokens to its outgoing edges (its *outputs*). The

number of tokens consumed or produced at a given port at each firing is called the *rate*. It is denoted as $r(\pi_m)$ where π_m is a port. In SDF, all rates are constant and known at compile time.

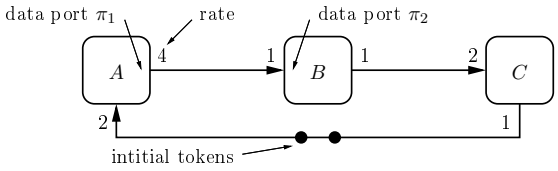


Figure 1: A simple SDF graph.

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*. 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 its initial state after a certain sequence of firings (ensuring boundedness of the FIFOs). The minimal such sequence is called an *iteration*.

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_1, X_2) connecting the ports π_1 and π_2; its balance equation is:

$$\#X_1 \cdot r(\pi_1) = \#X_2 \cdot r(\pi_2) \tag{1}$$

This equation states that the number of firings of the producer X_1, denoted $\#X_1$, multiplied by its rate $r(\pi_1)$, should be equal to the same expression for the consumer X_2. For example, the balance equation for edge (A, B) in Fig. 1 is $\#A \cdot 4 = \#B \cdot 1$.

The existence of solutions of the system of balance equations is referred to as *rate consistency*. The graph of Fig. 1 is rate-consistent, and the solutions are: $\#A = 1$, $\#B = 4$ and $\#C = 2$. Note that multiplying the solutions by the same positive constant makes another set of solutions. One usually considers only the minimal strictly positive integer solutions which are obtained by eliminating common factors.

The minimal solutions determine the number of firings of each actor per iteration. The next step is to determine a static order – the *schedule* – in which those firings can be executed. The schedule is obtained by an abstract computation where an actor is fired only when it has enough input tokens. The graph of Fig. 1 can only start by firing A; then, B has enough input tokens to be fired four times, and finally C twice. Since each actor has been fired the exact number of times requested by its solution, a schedule has been found. We represent it as the string AB^4C^2 where the superscripts denote repetition

count. Another valid schedule for the same graph is AB^2CB^2C which can also be written as $A(B^2C)^2$.

2.2 Our model: SPDF

We extend SDF by allowing rates to be parametric while preserving static schedulability. Let $\mathcal{P}$ be a set of symbolic variables. SPDF rates are defined by the grammar:

$$\mathcal{F} \quad ::= \quad k \mid p \mid \mathcal{F}_1 \cdot \mathcal{F}_2 \qquad \text{where } k \in \mathbb{N}^* \text{ and } p \in \mathcal{P}$$

Actually, SPDF rates are more general and can be integer polynomials and boolean expressions. For simplicity reasons, we limit ourselves to the previous grammar where rates are products of strictly positive integers (k) or *parameters* (symbolic variables) (p). Optionally, each parameter can be constrained to belong to a specific integer interval ($[1, +\infty)$ by default).

Fig. 2 shows a simple SPDF graph where the actors have constant or parametric rates (e.g., $p \cdot q$ for the input rate of C).

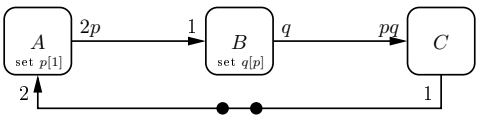


Figure 2: A simple SPDF graph

Unlike the rates of SDF graphs which are fixed at compile time, the parametric rates of an SPDF graph can change dynamically. The changes of each parameter are made by a single actor called its *modifier*. By default, a parameter can be changed between iterations. In SPDF, a modifier may change a parameter more often using the annotation "*set* $p[\alpha]$" where p is the parameter to be set and α is the exact (possibly symbolic) number of firings of the modifier between two parameter changes. We assume that a single modifier and period are provided for each parameter.. In Fig. 2, A and B are the modifiers for p and q; they may change their value every single and p firings, respectively.

Definition 1. *A SPDF graph is a tuple* $(\mathcal{G}, \mathcal{P}, i, d, r, M, \alpha)$*, where:*

- $\mathcal{G}$ *is a directed connected graph* $(\mathcal{A}, \mathcal{E})$ *with* $\mathcal{A}$ *a set of actors and* $\mathcal{E} \subseteq \mathcal{A} \times \mathcal{A}$ *a set of directed edges;*
- $\mathcal{P}$ *is a set of parameters;*
- $i : \mathcal{E} \to \mathbb{N}$ *associates each edge with its number of initial tokens;*
- $d : \mathcal{P} \to 2^{\mathbb{N}^*}$ *returns the interval of each parameter;*
- $r : \mathcal{A} \times \mathcal{E} \to \mathcal{F}$ *returns for each port (represented by an actor and an adjacent edge) its associated (possibly symbolic) rate;*
- $M : \mathcal{P} \to \mathcal{A}$ *and* $\alpha : \mathcal{P} \to \mathcal{F}$ *return for each parameter its modifier and its change period, respectively.*

3 Static Analysis for SPDF

This section presents the three static analyses needed to ensure boundedness and liveness of an SPDF graph. In Section 3.1, we check rate consistency by adapting the analysis of SDF to SPDF. Conditions for consistency and solutions of balance equations are computed in terms of symbolic expressions. In Section 3.2, we check that the change periods of each parameter are safe. Rate consistency and parameter change safety ensure boundedness. Section 3.3 completes the analysis chain by checking for liveness.

3.1 Rate Consistency

As in SDF, we check the rate consistency of an SPDF graph by generating the associated system of balance equations, which are the same equations as in (1), but with coefficients (*i.e.,* the rates) depending on parameters. When parameters are modified only between iterations, rate consistency alone ensures boundedness.

Definition 2 (Rate Consistency)**.** *An SPDF graph is called* rate consistent *if the system of balance equations has solutions for all possible values of all parameters.*

Like in SDF graph, multiple solutions are possible for a rate-consistent SPDF graph. These solutions differ from each other by a multiplication factor. However, we are only interested in the unique (symbolically) *minimal* solutions, which we implicitly use in the definitions and properties throughout this report. In Appendix A, we sketch an algorithm for computing *generic* solutions of balance equations. These solutions are computed, in general, as symbolic expressions in grammar $\mathcal{F}$. From the generic solutions, the minimal solution can be obtained by eliminating the (symbolic) *greatest common divisor* ('*gcd*') of all the solutions. This can be done easily by first decomposing all the solutions into primary factors (prime numbers and parameters), obtaining the expressions in the following form:

$$\underbrace{k_0 \cdot k_1 \cdot k_2 \cdots}_{\text{prime decomposition}} \cdot \underbrace{p_1 \cdot p_1 \cdots p_1}_{\text{the power of } p_1} \cdot \underbrace{p_2 \cdot p_2 \cdots p_2}_{\text{the power of } p_2} \cdots$$

Obviously, the *gcd* of two or more symbolic expressions in this form is obtained as maximal subset of primary factors that is common to all the expressions (i.e. the intersection of multisets of primary factors).

If the undirected version of the SPDF graph is acyclic, a solution always exists. When the SPDF graph contains undirected cycles, the graph may be rate inconsistent. There is, however, a necessary and sufficient condition for the existence of symbolic solutions. Each undirected cycle $X_1, X_2, \ldots, X_n, X_1$ where p_i and q_j denote the rates of edge (X_i, X_j) should satisfy the following condition:

$$(\textit{Cycle condition}) \quad p_1 \cdot p_2 \ldots \cdot p_n = q_1 \cdot q_2 \ldots \cdot q_n \tag{2}$$

This condition enforces that any factor encountered on an "output" port of a cycle should have a *symbolically identical* counterpart on an "input" port on this cycle.

Property 3 (Consistency)**.** *An SPDF graph is rate consistent if its undirected cycles satisfy the cycle condition.*

Proof. See Appendix A □

For example, the graph of Fig. 2 is consistent since its only cycle A, B, C, A satisfies the cycle condition which is $2p \cdot q \cdot 1 = 2 \cdot 1 \cdot pq$. The minimal solutions are $\#A = 1$, $\#B = 2 \cdot p$ and $\#C = 2$, yielding the schedule $AB^{2p}C^2$.

The algorithm either yields for each actor its (symbolic) solution, or returns an unsatisfied cycle condition that can be used by the programmer to fix his SPDF graph.

3.2 Parameter Change Safety

It is always safe to change parameter values between graph iterations [1]. Indeed, the rate consistency and liveness analyses ensure that the graph is bounded and live for any value of the parameters. Since the graph returns to its initial state after each iteration, all parameters can be modified at these stages. Nevertheless, it is sometimes useful to change the parameters more often, i.e., during an iteration. SPDF allows the programmer to specify a faster period using the "*set* $p[\alpha]$" annotation. Yet, not all periods are safe and their consistency must be checked. Consider, for instance, actor B that modifies q in Fig. 2. The period 1 would not be safe since it is only after p firing of B that C can consume its pq tokens. The rate pq would not be well defined if q can change p times before C is fired. On the other hand, the period p is safe since the iteration can be written $A(B^pC)(B^pC)$, with q being changed after each sequence (B^pC).

The criterion ensuring that parameter modification periods are safe relies on the notions of *influence*, *regions* and *local iterations*. Intuitively, the criterion states that a parameter can be modified once per local iteration of the region it influences. For Fig. 2, it can be shown that the region of influence of q consists of actors B and C and that q can be changed after each local iteration (B^pC), that is, after p firings of B.

Definition 4 (Influence)**.** *An edge $e = (A, B)$ is* influenced *by a parameter p, denoted $Infl(e, p)$, if p appears in the rates of e or in the solutions of the balance equations of its source and sink actors. Formally,*

$$Infl(e, p) \Leftrightarrow p \in \#A \vee p \in \#B \vee p \in r(A, e) \vee p \in r(B, e)$$

where $p \in \mathcal{F}$ if p occurs in the symbolic expression $\mathcal{F}$.

The *region* of influence of a parameter is the subset of edges it influences. Since an edge is a relation between actors, a region also specifies a subset of actors.

Definition 5 (Region)**.** *The region of edges $\mathcal{R}(p)$ influenced by p is defined as:* $\mathcal{R}(p) = \{e \mid Infl(e, p)\}$

We will sometimes abuse notation $\mathcal{R}$ to denote also the set of actors connected by the edges of the region. For example, the region of influence of q in Fig. 2 is $\mathcal{R}(q) = \{(B, C)\}$ and the actors in this region are $\{B, C\}$.

The solutions of the system of balance equations are *global solutions* in that they define the number of firings for the global iteration of the whole graph. Local solutions are solutions for a subset of actors; they denote a nested iteration.

Definition 6 (Local solutions)**.** *Let $\mathcal{A}$ be the set of actors of an SPDF graph and $\#X$ be the global solution of X. The local solution of X in the subset $\{X_1, \ldots, X_n\} \subset \mathcal{A}$, denoted $\#_{\mathrm{L}}X$, is obtained by dividing the global solution of X by the greatest common divisor: $\#_{\mathrm{L}}X_i = \frac{\#X_i}{gcd(\#X_1,\ldots,\#X_n)}$.*

For example, the global solutions for Fig. 2 are $\#A = 1$, $\#B = 2p$ and $\#C = 2$, forming the global iteration $AB^{2p}C^2$. The *gcd* of $\#B$ and $\#C$ is 2 and the local solutions for the subset $\{B, C\}$ are $\#_{\mathrm{L}}B = p$ and $\#_{\mathrm{L}}C = 1$. After one local iteration B^pC, all the edges influenced by q return to their initial state. Therefore, q can be changed after each such local iteration, hence after p firings of B, as specified by the "*set* $q[p]$" annotation.

Regions of influence of a given parameter can overlap (i.e., have common edges). Each local iteration of such region may entail firing the same actor a different number times. Such overlapping regions must be grouped so that the modification periods of their parameters are checked on the same subset of actors. Regions are then generalized to a subset of parameters $\mathcal{P}'$ as follows:

$$\mathcal{R}(\mathcal{P}') = \{e \mid \exists p \in \mathcal{P}', \mathit{Infl}(e, p)\}$$

For convenience, we also assume that the region of the empty set of parameters includes all edges of the graph: $\mathcal{R}(\emptyset) = \mathcal{E}$

When a region $\mathcal{R}(\mathcal{P}_2)$ is included within another region $\mathcal{R}(\mathcal{P}_1)$, the periods of the parameters in $\mathcal{P}_2$ can be checked on $\mathcal{R}(\mathcal{P}_2)$. The local iteration of $\mathcal{R}(\mathcal{P}_1)$ will always involve one or several local iterations of the inner region. Hence, the changes of parameters from $\mathcal{P}_1$ are always done between local iterations of $\mathcal{R}(\mathcal{P}_2)$ and are therefore safe for both regions.

Before checking the parameter modification safety criterion, we structure the set $\mathcal{P}$ of all parameters into a *hierarchy tree* of sets of parameters $\mathcal{P}_i$ such that:

- $\mathcal{P}$ is partitioned into non-empty partitions $\mathcal{P}_i$ that are placed at different nodes and leafs of the hierarchy;
- the root of the hierarchy tree is the empty parameter set $\emptyset$;
- if a partition $\mathcal{P}_i$ is a hierarchical child of a non-empty partition $\mathcal{P}_i$, then its region is strictly included in the region of the parent (*i.e.,* $\mathcal{R}(\mathcal{P}_i) \subset \mathcal{R}(\mathcal{P}_j)$); [1]
- the regions of two sets $\mathcal{P}_i$ and $\mathcal{P}_j$ which are not ancestor or descendant of each other are disjoints.

This structuring process is based on two basic steps:

- (*Decomposition*) the first step decomposes the current set of edges (initially $\mathcal{E}$) into disjoint regions. Consider the relation $e_1 \asymp e_2$ which holds if there exists a parameter influencing both edges e_1 and e_2. Then, disjoint regions are the connected components of the graph of the $\asymp$ relation. Each disjoint set of edges corresponds to a region of a disjoint set of parameters;

[1] Note that, by construction, all regions are also non-strictly included in the root region: (*i.e.,* $\mathcal{R}(\mathcal{P}_i) \subseteq \mathcal{R}(\emptyset)$).

- (*Nesting*) the second step finds, for each such independent region $\mathcal{R}(\mathcal{P})$, the largest subset $\mathcal{P}' \subset \mathcal{P}$ such that $\mathcal{R}(\mathcal{P}') \subset \mathcal{R}(\mathcal{P} - \mathcal{P}')$. The set $\mathcal{P} - \mathcal{P}'$ will be the root of the (sub-)tree that will be built by iterating the process (decomposition and nesting) on $\mathcal{P}'$. This process ends when $\mathcal{P}' = \emptyset$.

Fig. 3 represents a graph with two non-empty hierarchy levels: the parent level $\mathcal{P}_1 = \{p\}$ and the child level $\mathcal{P}_2 = \{q\}$. The parameter p influences all edges whereas q influences only (A, D) and (D, C), hence $\mathcal{R}(\mathcal{P}_2) \subset \mathcal{R}(\mathcal{P}_1)$. We can now state the criterion for parameter modification safety.

Definition 7 (Data Safety)**.** *An SPDF graph is* data safe *if, for each parameter p and its hierarchy node $\mathcal{P}_i$ ($p \in \mathcal{P}_i$), $\#M(p)$ is a multiple of $\alpha(p)$ and every actor X_j in $\mathcal{R}(\mathcal{P}_i)$ is such that $\#X_j$ is a multiple of $\#M(p)/\alpha(p)$.*

This criterion ensures that the local iteration count of any region $\mathcal{R}(\mathcal{P}_i)$ – computed by $gcd(\{\#X \mid X \in \mathcal{R}(\mathcal{P}_i\}))$ – is a multiple of the parameter change count, $\#M(p)/\alpha(p)$. Hence, per one change of each parameter p there are multiple local iterations of its hierarchy node $\mathcal{P}_i$. Thus, the change of parameter values can take place "safely", *i.e.,* in between the local iterations, when the data edges are in the initial state.

In Fig. 3, we have $q \in \mathcal{P}_2$, $\#M/\alpha = \#A/1 = 2$, and $\mathcal{R}(\mathcal{P}_2) = \{A, C, D\}$. The solutions for the actors in $\mathcal{R}(\mathcal{P}_2)$ are all multiples of 2: $\#A = 2$, $\#C = 2p$ and $\#D = 2pq$. The annotation "*set* q[1]" in A is thus data safe.

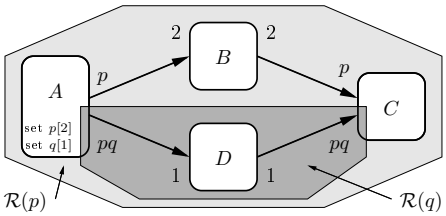


Figure 3: An SPDF graph with two hierarchy levels: $\mathcal{P}_1 = \{p\}$, $\mathcal{P}_2 = \{q\}$.

Definition 8 (Period Safety)**.** *An SPDF graph is* period safe *if, for every pair of parameters p and q such that $\#M(q)$ depends on at least one parameter of the hierarchy node of p, $\#M(q)/\alpha(q)$ is a multiple of $\#M(p)/\alpha(p)$.*

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. E.g., the graph of Fig. 2 is period-safe because even if the solution of $M(q)$ depends on p ($\#M(q) = \#B = 2p$), $\#B/\alpha(q) = 2$ is a multiple of $\#M(p)/\alpha(p) = 1$, so the criterion is satisfied.

Definition 9 (Safety)**.** *An SPDF graph is* safe *if it is both data safe and period safe.*

Boundedness can now be stated for rate consistent and safe SPDF graphs with bounded symbolic parameters.

Property 10 (Boundedness)**.** *All data edges and periods of a rate consistent and safe SPDF graph return to their initial state at the end of a global iteration. Furthermore, if every symbolic parameter can be bounded then the graph can be scheduled in bounded memory.*

Proof. See Appendix C. □

3.3 Liveness

The liveness property for a dataflow graph with parametric rates means that in every infinite execution, each actor fires an infinite number of times. An SPDF model can fail to be live when an actor waits for indefinite time for data tokens from an input edge or for parameter values from a modifier. This happens when a producer of the input tokens or a modifier is itself waiting for the given actor. Thus, this phenomenon is caused by a cyclic dependency. An *acyclic* directed SPDF graph where the parameter communication does not introduce cyclic dependencies then is trivially live. There exist actors that can always fire, thus allowing other actors to fire, and so on until the iteration is complete. However, if there exists a directed cycle, we must check that each cycle contains enough initial tokens. For example, if the (C, A) edge in Fig. 2 had only one token, then A (and therefore B and C) could never fire. Checking the liveness of SDF graphs is done by computing an iteration by abstract execution. It is not clear whether such an approach is applicable to SPDF. Instead, we present a sufficient condition on cycles.

Definition 11 (Saturated edge)**.** *An edge $e = (A, B)$ is said* saturated *is it has enough initial tokens to fire B the needed number of times to complete the iteration. Formally,*

$$i(e) \geq r(B, e) \cdot \#B$$

If the rate $r(B, e)$ or the solution $\#B$ contain symbolic parameters, the inequality must be proved for the upper bounds of those parameters. An edge cannot be saturated if the corresponding inequality involves symbolic parameters with no upper bound.

Definition 12 (Live cycle)**.** *A cycle $X_1, \ldots, X_n, X_1$ is said* live *if it contains at least one saturated edge.*

If $r(X_k, e_k) \cdot \#X_k$ is a symbolic expression, the inequality is checked using the maximum values of the parameters involved. If one of the parameters does not have a declared maximum, then the inequality is considered false. In Fig. 2, the cycle is live since $i(C, A) = 2$, $r(A, (C, A)) = 2$, and $\#A = 1$.

Definition 13 (User)**.** *A* user *U of a parameter p is an actor different from $M(p)$ such that p occurs in $\#U$ or in the rate of one of the ports of U.*

Property 14 (Liveness)**.** *Given a rate-consistent and safe SPDF graph where*

- *all directed cycles are live,*
- *for each parameter p there is a path from its modifier to each user of p in a directed acyclic graph obtained from the SPDF graph by removing only saturated edges,*

there exists a schedule where every actor is fired for an infinite number of times.

Proof. See Appendix D. □

The second requirement ensures that the parameter communication from the modifier to the users does not introduce non-live cycles. Liveness analysis either succeeds or returns to the programmer the faulty cycles (i.e., with not enough initial tokens) or the faulty modifier-user pairs.

We could use, as in [1] [3], less restrictive criteria using local solutions in strongly connected subgraphs. We skip this possibility here for simplicity reasons.

4 Compilation

We first show how to transform any safe SPDF graph into a graph which can be scheduled in bounded memory by dynamic scheduling (Section 4.1). Then, we describe how to generate a quasi-static schedule (Section 4.2).

4.1 Parameter Communication for Bounded Scheduling

The critical aspect for simulating or scheduling SPDF graphs is the communication of the values of parameters from the modifiers to the users. Not only the data paths (i.e. the SPDF graph itself), but also the control paths (i.e. the communication of the parameters), should be bounded and live. Such control paths are inserted automatically by an algorithm presented in this subsection.

Parameter communication is defined in Appendix B. Our algorithm implements this definition by taking the values of the parameters computed by the modifiers and propagating them to the users. This is done by adding extra actors, edges, and ports, forming *parameter distribution networks* (PDNs). This occurs in such a way that the SPDF graph when instrumented by PDNs remains rate consistent, safe and live according to the criteria defined in Section 3. The PDNs link $M(p)$ to all the users of p. These networks are built in three steps.

The first step adds to $M(p)$ a new *output* port, and to each user a new *control* port, respectively to send and to receive the successive values of p. Control ports behave exactly as in BDF [4]: each actor must read input tokens from all its control ports before reading tokens from its regular data ports.

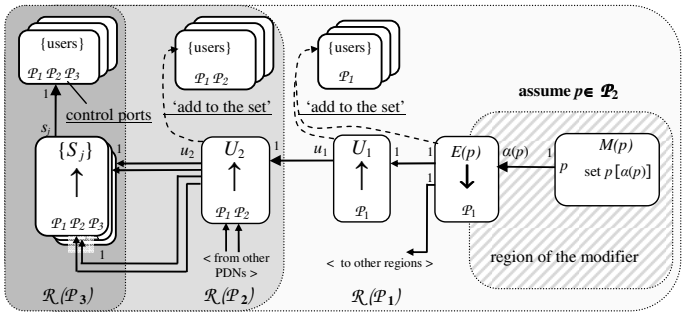


Figure 4: Communication of a parameter from the modifier to the users.

The second step adds a new actor $E(p)$, called the *emitter* of p, and a new edge $e = (M(p), E(p))$, such that $r(M(p), e) = 1$ and $r(E(p), e) = \alpha(p)$. Among the $\alpha(p)$ tokens it receives, $E(p)$ transmits only the first one, the one that

contains the new value of p. To transmit the output token, the emitter has one or more output ports with rate 1. Note that the rates at the edge $(M(p), E(p))$ ensure that , by balance equations:

$$\#E(p) = \#M(p)/\alpha(p) \tag{3}$$

We refer to $E(p)$ as a *downsampler* ($\downarrow$), because it translates multiple tokens (samples) into one. This is illustrated in Fig. 4.

In general, $E(p)$ fires once per certain number of firings of the users, so each user should receive the same value of p repeated a certain number of times. The third step implements this requirement using *upsamplers* ($\uparrow$) that repeat every input token a given number of times. This step depends on the region hierarchy.

The order in which the users are connected to the emitter is defined by the hierarchal location of their ports (see Definition 16 in Appendix B). For parameter p contained in hierarchy node $\mathcal{P}_N$ the user ports can be located in the hierarchy node $\mathcal{P}_N$ or in a node $\mathcal{P}_{N'}$ that is lower in the hierarchy tree. For example, in Fig. 4 we assume that p is in $\mathcal{P}_2$, which has a hierarchical child $\mathcal{P}_3$, where some users of $\mathcal{P}_2$ are located.

The parameter distribution network (PDN) of parameter p to destination region $\mathcal{R}(\mathcal{P}_{N'})$ is the tree of sampler actors and edges from the modifier of p to the users of p whose ports are located in region $\mathcal{P}_{N'}$. For example, in Fig. 4, we illustrate a PDN to destination region $\mathcal{R}(\mathcal{P}_3)$; thus, in this example $N' = 3$. As illustrated in the figure, all the PDNs of parameter p share the same emitter and go to different destination regions.

Let us describe the insertion of one PDN for parameter p to $\mathcal{R}(\mathcal{P}_{N'})$ and then describe how all PDNs are inserted. Let us consider the path in the hierarchy tree $\mathcal{P}_1$, $\mathcal{P}_2$,...,$\mathcal{P}_{N'}$, where $\mathcal{P}_1$ is the location of the emitter output port. We insert a chain of upsampling actors $(U_k)_{k=1...N'-1}$ with upsampling rates u_k, as illustrated in Fig. 4. The input of U_1 is connected to the emitter output, and the input of $U_k, k > 1$ is connected to the output of U_{k-1}. The rates u_k are computed according to the semantics of parameter communication as described in Appendix B (Equality (14)). Having thus inserted the upsampling chain to the destination region of the PDN, now for each internal user X_j, we insert an extra upsampling actor S_j with upsampling rate s_j equal to local solution $\#_{\mathrm{L}} X_j$ in the context of region $\mathcal{R}(\mathcal{P}_{N'})$ (see Fig. 4).

Our PDN insertion algorithm first inserts all the emitters, then visits the hierarchy nodes $\mathcal{P}_i$ and connects the users that have ports located in $\mathcal{R}(\mathcal{P}_i)$ to the emitters of all parameters that influence these ports. For every parameter, a PDN is created to the region of $\mathcal{P}_i$ as destination, using the PDN insertion procedure defined above. The nodes $\mathcal{P}_i$ are visited in a bottom-up order. Although U_k and S_j are new users, they can be located only in the current or higher hierarchy levels. Thus, when adding these new users the lower level regions do not need to be revisited. Note that if $E(p)$ is a user of p we do not communicate parameter p to it by a PDN, as this would create a deadlocked cyclic path and the emitter anyway receives the values of the parameter directly from the modifier.

After inserting the PDN, the final step is to shortcut all the samplers with rate 1. The graph of Fig. 2 with its PDN is shown in Fig. 5. It is interesting to note that by PDN insertion we obtain an SPDF graph where not only data but also control communication is done via FIFO channels. Therefore, SPDF

can be seen as a special case of Kahn process networks, just as SDF, PSDF and other dataflow MoCs.

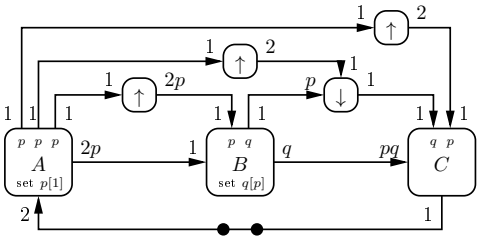


Figure 5: The SPDF graph of Fig. 2 with its PDN.

Unlike general Kahn process networks, the SPDF graphs have bounded-memory executions, as shown by Property 10. It follows that they can be scheduled in bounded memory by the general dynamic scheduling algorithms presented in [9, 5]. However, in the next section we show that for a certain class of SPDF graphs the schedule can be computed statically.

4.2 Quasi-static scheduling

In SPDF, since the firing counts of some actors can be parametric, so is the schedule, which is said to be *quasi-static* [1]. Currently, our quasi-static scheduling algorithm, next to the static analysis criteria for rate consistency, safety and liveness, requires that all parameters p_i can be ordered such that their corresponding emitters $E(p_i)$ are related by:

$$\#E(p_{i+1}) = f_i \cdot \#E(p_i)$$

for some $f_i \in \mathcal{F}$. Observing that $\#E(p_i)$ denotes the modification count of p_i during a global iteration, we can expect our requirement to hold often in practice. A typical streaming application can be represented by nested loops where each parameter is modified exactly once at a certain loop level. The ordering of parameters corresponds to the different loop levels.

Our liveness criterion implies that we can ignore the edges with initial tokens and consider the corresponding acyclic graph. First, for the source (*i.e.,* non-PDN) part of the graph, we generate a string composed of the actors of that graph sorted topologically, e.g., ABC for Fig. 5.

In this string, we replace every actor X by the *wrapper*:

$$\begin{cases} X^{\#X} & \textbf{if } X \text{ is not a user or modifier of any parameter} \\ (\text{set } p_{i_1}; (\text{set } p_{i_2}; \dots (\text{set } p_{i_N}; X^{f'_{N+1}})^{f'_N} \dots)^{f'_2})^{f'_1} & \textbf{otherwise} \end{cases} \tag{4}$$

where p_{i_k} $(k = 1 \dots N)$ are parameters used[2] or modified by X; i_k are the increasing indexes of parameters in the above ordering; 'set p_i' sets a new parameter value for the given actor; $f'_1 = \#E(p_1)$; $f'_k = \frac{\#E(p_{i_k})}{\#E(p_{i_{k-1}})}$ for $k = 2 \dots N$; $f'_{N+1} = \frac{\#X}{\#E(p_{i_N})}$. For Fig. 5, we produce:

$$(\text{set } p; A;) \quad (\text{set } p; (\text{set } q; B^p)^2) \quad (\text{set } p; (\text{set } q; C)^2)$$

[2] see definition 13

Finally, we introduce modifier-to-user communication statements, equivalent to PDNs. The modifier is implicitly connected to each user by a separate queue. It writes to all the queues with a single "push p_i". Each user reads the parameter values by a "pop p_i". In the wrapper for actor X, we replace the "set p_i" by "push p_i" if X is the modifier or by "pop p_i" otherwise. The push are moved *after* the actor invocation[3], because the actor as modifier has to compute the value to be pushed. In our running example, we get:

$$(A; \text{push } p) \quad (\text{pop } p; (B^p; \text{push } q)^2) \quad (\text{pop } p; (\text{pop } q; C)^2)$$

5 Case Study

We have applied SPDF to realistic case studies provided by an industrial partner. Figure 6 shows an SPDF model for a video decoder. The actor "**input**" reads the coded input frame and triggers a variable-length decoder "**vld**" for the 100 macroblocks of the frame. Once per frame (period 100) "**vld**" determines parameter p indicating whether the frame uses motion compensation. The actor "**mv**" determines whether the current macroblock has motion vectors (parameter t). If both conditions hold ($p \cdot t$), motion compensation is performed by the actor "**mc**". The actor "**vld**" triggers the calculation of four luminance blocks, "**lum**", each one computing an l indicating whether it is coded. For coded blocks, inverse discrete cosine transform (IDCT) is performed by the actor "**l-idct**". The actor "**vld**" also determines whether chrominance is coded in macroblock (parameter c). If so, it triggers the execution of IDCT, "**c-idct**", followed by upscaling, "**upsc**", which builds four chrominance blocks out of one. Finally, the four luminance and chrominance blocks of the macroblock are converted one-by-one to RGB color format by the actor "**color**" and sent to the output frame. For each 100 macroblocks, the output frame expects 400 blocks.

Concerning rate consistency, the cycle condition is true for three undirected cycles, so the balance equation algorithm succeeds. Concerning safety, our hierarchy computation algorithm finds three disjoint nodes with parameter sets $\mathcal{P}_1 = \{p, t\}$, $\mathcal{P}_2 = \{l\}$, and $\mathcal{P}_3 = \{c\}$. The video decoder does not have directed cycles and the modifiers are located upstream to the users, so the liveness criterion holds.

Then, the PDN is inserted, shown in grey in Fig. 6. Finally, the quasi-static scheduler examines the periods of the modifiers and sorts the parameters: (p (modified $\times 1$/frame), c ($\times 100$ more), l, t ($\times 4$ more)). Applying our algorithm, we obtain the following schedule:

$$\textbf{input} \quad ((\textbf{vld}; \text{push } c)^{100}; \text{push } p) \quad (\text{pop } p; (\textbf{mv}; \text{push } t)^{400})$$
$$(\text{pop } p; (\text{pop } t; (\textbf{mc})^{pt})^{400}) \quad ((\textbf{lum}; \text{push } l)^{400})$$
$$((\text{pop } l; (\textbf{l-idct})^{l})^{400}) \quad (\text{pop } c; (\textbf{c-idct})^{c})^{100}$$
$$(\text{pop } c; (\textbf{upsc})^{c})^{100} \quad (\text{pop } c; (\text{pop } l; \text{pop } t; \textbf{color})^{4})^{100} \ \textbf{output}$$

Actually, all the parameters (p, t, l, c) have been encoded as booleans. For simplicity reasons, we have not presented this extension, but the whole methodology presented in this work applies to this example without restrictions.

[3] while staying at the same level of parentheses nesting

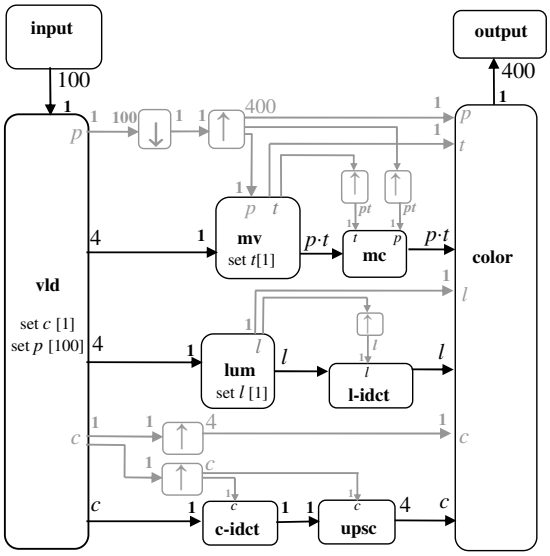


Figure 6: Video decoder (compiler-inserted elements shown in gray).

6 Conclusions

We presented SPDF, a novel MoC for parametric streaming applications enabling static analysis and scheduling. We formulated sufficient and general static criteria for boundedness and liveness. In SPDF, parameter changes are allowed even within iterations. Their safety can be checked and their implementation is made explicit. All this was possible because we could manipulate and compare dynamic values by well-defined static operations on symbolic expressions. The same holds for quasi-static scheduling, which is the first step towards code generation for multi-core systems.

The most closely related MoC is PSDF [1], which requires to manually find the hierarchy levels and enclose them into hierarchical actors, *e.g.,* four levels for Fig. 6. With PSDF, the analysis is not completely static, as [1] applies a run-time analysis at hierarchy boundaries. The hierarchy analysis proposed in [8] requires significant manual help. The Scenario-Aware Data-Flow (SADF) MoC [10] is a dynamic extension of the SDF for which various advanced performance analysis techniques have been developed. Yet, SADF does not define any boundedness analysis if dataflow rates change more often than once per global iteration.

The Variable-Rate Data-Flow (VRDF) MoC [11] introduced support for frequent changes of actor rates. However, VRDF imposes strong structural constraints on the graph. In particular, for each parametric rate p there can be at most one port $\pi_a = (A, e_a)$ in the graph that produces p tokens, which must be matched by exactly one port $\pi_b = (B, e_b)$ that consumes p tokens. Moreover, let $\mathcal{G}'$ be the minimal subgraph that includes all the graph paths from π_a to π_b. Then the VRDF requires that the local solutions of A and B in this subgraph be equal to 1, so that parameter p can change safely at every firing of A and B. This VRDF requirement is obviously significantly more restrictive than our

safety criterion. Moreover, quite a few practical applications do not satisfy this requirement. For example, in our case study in Fig. 6, we see that 'chrominance' parameter c does not satisfy the VRDF requirement, as parameter c may safely change only once per four firings of actor **color**, which has a port that consumes c tokens.

Multiprocessor scheduling for SPDF is an obvious and important extension of our work. Other important future work is SPDF scheduling with dynamic voltage and frequency scaling and performance-memory trade-off exploration. We also intend to explore other forms of dynamicity, such as dynamic graph reconfigurations, while preserving static schedulability.

References

[1] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling of DSP systems. In *ICASSP'00*. IEEE, 2000.

[2] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet. OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems. *SIGARCH Comput. Archit. News*, 36:29–35, June 2009.

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Press, 1996.

[4] J. Buck and E. Lee. Scheduling dynamic data-flow graphs with bounded memory using the token flow model. In *ICASSP'93*, volume I, pages 429–432, Minneapolis (MN), USA, Apr. 1993. IEEE.

[5] M. Geilen and T. Basten. Requirements on the execution of Kahn process networks. In *ESOP'03*, pages 319–334. Springer-Verlag, 2003.

[6] G. Kahn. The semantics of a simple language for parallel programming. In *Information processing*, pages 471–475. North Holland, 1974.

[7] E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, Jan. 1987.

[8] S. Neuendorffer and E. Lee. Hierarchical reconfiguration of dataflow models. In *MEMOCODE'04*, pages 179–188. IEEE, 2004.

[9] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, 1995.

[10] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *IC-SAMOS'11*, pages 404–411. IEEE, 2011.

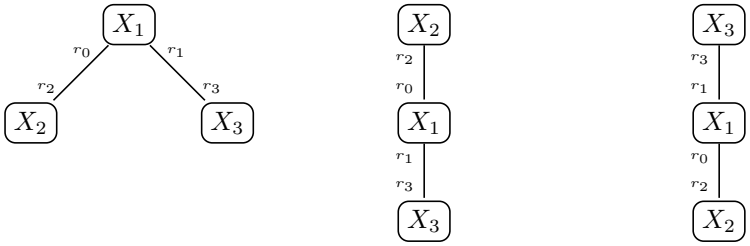
[11] M. Wiggers, M. Bekooij, and G. Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *RTAS'08*, pages 183–194. IEEE, 2008.

Appendix

A Proof of Rate Consistency

Property 3 (Consistency)**.** *An SPDF graph is rate consistent if its undirected cycles satisfy the cycle condition.*

Proof. If the undirected SPDF graph contains no cyclic paths then, for any given actor X_i the SDF graph can be see as an undirected tree rooted in X_i. For instance, the following trees rooted in X_1, X_2 and X_3 respectively, are three representations of the same graph.



We define the *generic solutions* of the balance equations as

$$\#X_i = p_1 \dots p_n \qquad \text{for each actor } X_i$$

where the p_i's are the rates of the "input" ports of the tree rooted in X_i. For the previous graph, such a solution would be

$$\#X_1 = r_2 r_3 \quad \#X_2 = r_0 r_3 \quad \#X_3 = r_1 r_2$$

Let us consider an arbitrary edge $A \overset{p}{\text{——}} \overset{q}{} B$, then the balance equation associated with this edge is

$$\#A \cdot p = \#B \cdot q \tag{5}$$

The generic solutions for A and B are of the form

$$\#A = q \cdot i_1 \cdots i_n \quad \#B = p \cdot i_1 \cdots i_n$$

which are clearly solutions of equation 5. Note that minimal solutions can be obtained by removing all common factors from generic solutions.

In case of a SPDF graph with undirected cycles, we proceed exactly as above on a spanning tree. That tree is obtained by selecting a cycle and removing one of its edge to suppress it. This process is pursued until no cycle remains. As shown in the acyclic case, the balance equations corresponding to the edges of the spanning tree are satisfied by the generic solutions. The remaining equations correspond to edges suppressed to remove cycles. Let us consider a cycle

$$X_1 \overset{p_1}{\text{——}} \overset{q_2}{} X_2 \overset{p_2}{\text{——}} \dots \text{——} \overset{q_n}{} X_n \overset{p_n}{\text{——}} \overset{q_1}{} X_1$$

in the SPDF graph whose edge $X_n \overset{p_n}{\text{——}} \overset{q_1}{} X_1$ does not appear in the spanning tree. We must show that the generic solutions found using the spanning tree

are also solutions for the balance equation associated with this suppressed edge. We know that the generic solutions verify the following balance equations:

$$\#X_i \cdot p_i = \#X_{i+1} \cdot q_{i+1} \qquad \text{for } i = 1 \ldots n-1$$

Then, multiplying all the *lhs* and *rhs* of the $n-1$ equations we get

$$\begin{aligned}
\#X_1 \cdot \ldots \cdot \#X_{n-1} \cdot p_1 \ldots \cdot p_{n-1} &= \#X_2 \cdot \ldots \cdot \#X_n \cdot q_2 \ldots \cdot q_n \\
\text{by removing the common factors} & \\
\#X_1 \cdot p_1 \cdot p_2 \ldots \cdot p_{n-1} &= \#X_n \cdot q_2 \cdot \ldots \cdot q_n \\
\text{and multiplying both sides by } q_1 & \\
\#X_1 \cdot q_1 \cdot p_1 \cdot p_2 \ldots \cdot p_{n-1} &= \#X_n \cdot q_1 \cdot q_2 \ldots \cdot q_n \\
\text{then using the cycle condition} & \\
\#X_1 \cdot q_1 \cdot p_1 \cdot p_2 \ldots \cdot p_{n-1} &= \#X_n \cdot p_1 \cdot p_2 \ldots \cdot p_n \\
\text{and simplifying by } p_1 \cdot p_2 \ldots \cdot p_{n-1} & \\
\#X_1 \cdot q_1 &= \#X_n \cdot p_n
\end{aligned}$$

which is the balance equation corresponding to the suppressed edge.

The cycle condition guarantees that the balance equation of any suppressed edge is satisfied. Hence, the generic solutions satisfy the balance equations for all edges, which guarantees rate consistency of the SPDF graph. □

B Parameter Communication

SPDF dictates a specific way of setting the parameters that influence the dynamic rates. This section describes how the parameters set by the modifiers should be propagated to the users.

We define parameter communication only for SPDF graphs *that satisfy the safety criterion*. As a tool to define it, we consider a modified variant of the SPDF model, which we call *pseudo-SPDF*. Compared to SPDF, this model is characterized by a more relaxed ordering of productions and consumptions of tokens at the edges. In pseudo-SPDF, ports are decoupled from each other and are fired independently. An input port does not necessarily wait until there are enough input tokens in the edge. As a result, some edges can sometimes be seen as carrying a "negative" number of tokens. Nevertheless, every port produces or consumes exactly the same parametric number of tokens as specified in the SPDF graph. The k-th firing of a pseudo-SPDF port results in an exactly equivalent production or consumption at this port by the k-th firing of the actual SPDF actor. An important property of pseudo-SPDF is that if the ports fire the same number of times as specified in the SPDF schedule, then the pseudo-SPDF graph will end up in the same state, i.e., the edges will carry the same number of tokens. By defining the parameter changes in a pseudo-SPDF graph, we define which parameter changes should be experienced by actors in the SPDF MoC.

A *pseudo-schedule* describes the sequential execution of a pseudo-SPDF graph. It is a schedule expressed in terms of ports instead of actors. We write them into curly braces "{...}" to distinguish them from 'regular' schedules. The ports fire the same number of times per iteration as their corresponding SPDF actor. A simple naive pseudo-schedule can be constructed by listing all the ports of the graph in an arbitrary order with a superscript giving the solution of

balance equations of the corresponding actors. Such a pseudo-schedule is called an *initial pseudo-schedule*. For example, the initial pseudo-schedule for Figure 3 is:

$$\{A_{p,q}^2 A_p^2 B^p B^p D^{2pq} D^{2pq} C_{p,q}^2 C_p^2\} \tag{6}$$

Each port is denoted by the actor it belongs to, with no subscript when the rate is constant (e.g., B for the input port and for the output port of B), with a subscript denoting the parameters it depends on (e.g., A_p or $A_{p,q}$ for the two output ports of A), and with a superscript denoting its firing count as in an SPDF schedule (e.g., D^{2pq} for the input port and for the output port of D). Note that two ports can have identical subscript and superscript (e.g., the two ports of B or D). It is irrelevant for us to distinguish between them.

The describe parameter communication within pseudo-schedule, we start by adding virtual ports called *emitter ports*. One emitter port $E(p)$ is added for each parameter p. The role of the emitter port is to propagate the new value of the parameter generated by the modifier. These values are *enqueued* in the output queue attached to the emitter port until they are retrieved and used to instantiate parametric rates and firing counts. By definition, an emitter port fires the same number of times as the modifier sets the new value of its parameter. Therefore, the superscript of the emitter port of p is $\#M(p)/\alpha(p)$, which is in line with Equality (3). For the example of Figure 3, we have $\#M(p)/\alpha(p) = 1$ and $\#M(q)/\alpha(q) = 2$. We insert $E(p)$ and $E(q)^2$ to obtain the following pseudo-schedule:

$$\{E(p)E(q)^2 A_{p,q}^2 A_p^2 B^p B^p D^{2pq} D^{2pq} C_{p,q}^2 C_p^2\} \tag{7}$$

A pseudo-schedule like (7) does not reflect the change of parameters during the global iteration yet. This is done by reordering the pseudo-schedule and grouping ports according to the hierarchy of regions. This grouping of ports is done according to the notion of *containment* and *location* of ports in the hierarchy regions. [4] Region $\mathcal{R}(\mathcal{P}_i)$ *contains* the given port π if a parameter from $\mathcal{P}_i$ occurs in the subscript and subscript of π. Formally:

Definition 15 (Port Region)**.** *A port $\pi = (A, e)$ is contained in a region $\mathcal{R}(\mathcal{P}_i)$ if $\exists p \in \mathcal{P}_i, p \in r(\pi) \vee p \in \#(A)$*

Definition 16 (Port Location)**.** *A port is said to be* located *in the hierarchically lowest region $\mathcal{R}(\mathcal{P}_i)$ that contains it.*

For example, in Figure 3, we had two hierarchical nodes: $\mathcal{P}_1 = \{p\}$ and $\mathcal{P}_2 = \{q\}$, and the root node for an empty parameter set $\emptyset$. The port $A_{p,q}$ is contained in $\mathcal{R}(\{q\})$, $\mathcal{R}(\{p\})$, and $\mathcal{R}(\emptyset)$. The lowest region is $\mathcal{R}(\{q\})$, so $A_{p,q}$ is *located* there.

According to the location of ports, the pseudo-schedule (7) is reordered as follows:

$$\{E(p)E(q)^2(A_p^2 C_p^2 B^p B^p(\underbrace{D^{pq} D^{pq} C_{p,q} A_{p,q}}_{\mathcal{R}(\{q\})})^2)\} \tag{8}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\mathcal{R}(\{p\})}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\mathcal{R}(\emptyset)}$$

[4] See Section 3.2 for the definition of regions

In general, the grouping is done by traversing the hierarchy tree from bottom to top (*e.g.*, first $\{q\}$, then $\{p\}$, and then $\emptyset$ in our example). As a result, we get a string representation of the hierarchy tree, where the lower regions are nested within the higher regions. By computing and factorizing the (symbolic) *gcd* of the solutions (superscripts) inside regions, we make explicit the local iteration of regions. For example, in (8), the greatest common divisor of region $\mathcal{R}(\{q\})$ is 2, which is factorized and placed outside the parentheses, leaving inside the schedule for the local iteration of $\mathcal{R}(\{q\})$.

The last step to obtain a correct pseudo-schedule is to make explicit the actual retrieve and change of parameters. This is done using *parameter switch points*, denoted $\triangle\, p$, for each parameter p. As discussed in Section 3.2, the parameter may only change at the beginning of local iteration of its hierarchical region. A switch point is inserted at the beginning of the local subschedule of $\mathcal{R}(\mathcal{P}_i)$ for every parameter in $\mathcal{P}_i$. In our running example, the result is:

$$\underbrace{\{E(p)E(q)^2(\triangle p A_p^2 C_p^2 B^p B^p(\underbrace{\triangle q \underbrace{D^{pq}D^{pq}C_{p,q}A_{p,q}}_{\mathcal{R}(\{q\})})^2)\}}_{\mathcal{R}(\{p\})}}_{\mathcal{R}(\emptyset)} \tag{9}$$

The new parameter value of p, which was earlier enqueued by $E(p)$, is precisely retrieved at the switch point $\triangle p$. In our running example, two values of parameter q are enqueued at the beginning, by $E(q)^2$, and then dequeued by $\triangle q$, which belongs to the local schedule of region $\mathcal{R}(\{q\})$ executed also twice.

Note that in general a switch point does not retrieve the parameter value at every firing of that point but at a certain period, called *switch period*. The computation of the switch period is defined later.

After grouping the local schedules and inserting the switch points, we finally obtain a correct pseudo-schedule. There are however other correct pseudo-schedules which can be obtained by *transformations*. For example, changing the order of ports of the given region is such a transformation. Another transformation (that we will use later) is refactoring.

Property 17 (Refactoring a pseudo-schedule)**.** *Let $\mathcal{S}_a$ and $\mathcal{S}_b$ be some subschedules of a correct pseudo-schedule, let f_a, f_b, g, and h be expressions in $\mathcal{F}$. Then, the transformation from*

$$\{\ldots(\mathcal{S}_a^{f_a\cdot g}\mathcal{S}_b^{f_b\cdot g})^h\ldots\} \qquad \textit{into} \qquad \{\ldots(\mathcal{S}_a^{f_a}\mathcal{S}_b^{f_b})^{g\cdot h}\ldots\}$$

results in an equivalent pseudo-schedule.

Proof. Sketch: In a correct pseudo-schedule, there is only one switch point for every parameter. Further, this switch point cannot occur between two subschedules that use this parameter. Hence, the parameters that may occur in g have exactly the same value in $\mathcal{S}_a^{f_a\cdot g}$ as in $\mathcal{S}_b^{f_a\cdot g}$. Therefore g has the same value in both cases and we can factorize this common factor by taking it outside the parentheses. □

The following definition is needed to define the superscripts occurring within pseudo-schedules

Definition 18 (Local iteration *w.r.t.* the parent region)**.** *The local iteration count of region $\mathcal{R}(\mathcal{P}_i)$ in the context of its parent region $\mathcal{R}(\mathcal{P}_j)$, $\mathcal{R}(\mathcal{P}_i) \subseteq \mathcal{R}(\mathcal{P}_j)$ is defined as:*

$$f_{\mathrm{r}i} = \frac{gcd\{\#X \mid X \in \mathcal{R}(\mathcal{P}_i)\}}{gcd\{\#X \mid X \in \mathcal{R}(\mathcal{P}_j)\}}$$

This local iteration count gives the number of iterations of $\mathcal{R}(\mathcal{P}_i)$ per each iteration of $\mathcal{R}(\mathcal{P}_j)$.

We can now complete the definition of parameter change by defining the switch periods. From the safety criteria, we know that the solutions of the regions are multiples of $\#E(p)$ for all parameters p that influence these regions. The data criterion ensures this relation between the emitters and the user actors. The period safety criterion complements the data safety criterion such that the emitters themselves can be seen as the users of parameters of other emitters, ensuring that the abovementioned relation holds also in this case.

For a given parameter p contained in hierarchy node $\mathcal{P}_i$, consider the emitter port $E(p)$. From the safety criteria, the emitter port is located in a hierarchy region $\mathcal{R}(\mathcal{P}_j)$ that is strictly higher in the hierarchy tree than $\mathcal{P}_i$. It then follows that there is a path in the hierarchy tree from $\mathcal{P}_j$ to $\mathcal{P}_i$ with $N-1$ edges for some $N > 1$. Without loss of generality, let us assume that $j = 1$, $i = N$, and that the nodes in the hierarchy path are $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_N$. The subschedule that includes only the ports located in the given hierarchical path is of the form:

$$\{E(p)^{f_E} \ldots (\triangle\mathcal{P}_2 \ldots (\triangle\mathcal{P}_3 \ldots \triangle\mathcal{P}_{N-1}(\triangle\mathcal{P}_N \ldots)^{f_{\mathrm{r}N}} \ldots)^{f_{\mathrm{r}3}})^{f_{\mathrm{r}2}}\} \tag{10}$$

where f_E is the local solution of the emitter in region $\mathcal{R}(\mathcal{P}_1)$, $\triangle\mathcal{P}_i$ is the collection of switch points for the parameters in $\mathcal{P}_i$ and $f_{\mathrm{r}N}$ is specified by Definition 18.

To define the switch period of $\triangle p$ properly, we must consider the relative frequency of execution of the emitter port $E(p)$ *w.r.t.* the switch point $\triangle p$. Let us remove from schedule (10) all the irrelevant elements (*i.e.,* all ports except $E(p)$ and all switch points in $\triangle\mathcal{P}_N$ except $\triangle p$) to get:

$$\{E(p)^{f_E}(\triangle\mathcal{P}_2(\triangle\mathcal{P}_3 \ldots \triangle\mathcal{P}_{N-1}(\triangle p)^{f_{\mathrm{r}N}} \ldots)^{f_{\mathrm{r}3}})^{f_{\mathrm{r}2}}\} \tag{11}$$

We now transform the pseudo-schedule to make clear how many times $\triangle p$ is executed per execution of $E(p)$. From the safety criteria, it follows that:

$$\exists u \in \mathcal{F}, \quad u \cdot f_E = f_{\mathrm{r}2} \cdot f_{\mathrm{r}3} \cdot \ldots \cdot f_{\mathrm{r}N}$$

In other words, the right-hand side of this equality is a multiple of f_E. Therefore, we can split every $f_{\mathrm{r}(k+1)}$ for $k = 1 \ldots N-1$ into two factors:

$$f_{\mathrm{r}(k+1)} = u_k \cdot f_{\mathrm{e}k} \tag{12}$$

such that:

$$\begin{aligned} u_1 \cdot u_2 \ldots u_{(N-1)} &= u \\ f_{\mathrm{e}1} \cdot f_{\mathrm{e}2} \ldots f_{\mathrm{e}(N-1)} &= f_E \end{aligned}$$

One of the possible formulas that can be used to compute $f_{\mathrm{e}k}$ and u_k for $k = 1 \ldots N-1$ is given below:[5]

$$f_{\mathrm{e}k} = gcd(f_{\mathrm{r}(k+1)}, f_{\mathrm{E}k}) \tag{13}$$

$$u_k = f_{\mathrm{r}(k+1)}/f_{\mathrm{e}k} \tag{14}$$

[5] Any other valid formula would also result in a pseudo-schedule that is equivalent to (11). Thus a different choice for a formula would not essentially change the behavior.

where $f_{\mathrm{E}1} = f_{\mathrm{E}}$ and $f_{\mathrm{E}(k+1)} = f_{\mathrm{E}k}/f_{\mathrm{e}k}$. Substituting (12) into (11), we get:

$$\{E(p)^{f_{\mathrm{e}1}\cdot f_{\mathrm{e}2}\cdots f_{\mathrm{e}(N-1)}}(\triangle\mathcal{P}_2(\triangle\mathcal{P}_3\ldots(\triangle p)^{u_{(N-1)}\cdot f_{\mathrm{e}(N-1)}}\ldots)^{u_2\cdot f_{\mathrm{e}2}})^{u_1\cdot f_{\mathrm{e}1}}\}$$

Using the transformation from Property 17 N times, we get:

$$\{(E(p)\ (\triangle'\mathcal{P}_2(\triangle'\mathcal{P}_3\ldots(\triangle p)^{u_{(N-1)}}\ldots)^{u_2})^{u_1}\ )^{f_{\mathrm{e}1}\cdot f_{\mathrm{e}2}\cdots f_{\mathrm{e}(N-1)}}\}$$

where $\triangle'\mathcal{P}_k$ are switch points that are different from the switch points $\triangle\mathcal{P}_k$ as they were before the transformations. The periods of these "modified" switch points are multiplied by the factors that we took outside the parentheses. Indeed, $\triangle'\mathcal{P}_k$ switches less frequently than $\triangle\mathcal{P}_k$ by a factor $f_{\mathrm{e}k}\cdot f_{\mathrm{e}(k+1)}\cdots f_{\mathrm{e}(N-1)}$ for $k = 2\ldots(N-1)$.

From this result, we finally obtain the schedule that defines the number of executions of $\triangle p$ per one execution of $E(p)$. This schedule is given below:

$$\{E(p)(\triangle'\mathcal{P}_2(\triangle'\mathcal{P}_3\ldots(\triangle p)^{u_{(N-1)}}\ldots)^{u_2})^{u_1}\} \tag{15}$$

The switch period of $\triangle p$ is defined to be such that at every iteration of schedule (15), the switch point $\triangle p$ will retrieve a value only at the first firing and then not for the remaining firings. This ensures that the value of p is dequeued once per iteration of the schedule above, which is consistent with the fact that it is also enqueued only once, by the firing of $E(p)$.

From schedule (15), it might seem that the switch period is $u = u_1 \cdot u_2 \cdot \ldots u_{(N-1)}$. However, in general, this is not true, because u_k depend on parameters in $\mathcal{P}_2, \mathcal{P}_3, \ldots$, which switch their values during the iteration of this schedule. Thus, the switch periods of $\triangle p$ depends on the switch periods of the parameters strictly higher in the hierarchy. This is done recursively by first calculating the periods for the parameters that are higher in the hierarchy and then deriving the periods lower in the hierarchy. It is unclear whether there is a general way to express the switch period analytically, in a direct way. For this reason we content ourselves by the above indirect definition, using pseudo-schedule (15).

This completes the definition of parameter changes in the SPDF graph. Correct pseudo-schedules define where parameters may change their values and the frequency (consistent with "*set* $p[\alpha]$" annotations) of those changes.

C Proof of Boundedness

Property 10 (Boundedness)**.** *All data edges and periods of a rate consistent and safe SPDF graph return to their initial state at the end of a global iteration. Furthermore, if every symbolic parameter can be bounded then the graph can be scheduled in bounded memory.*

Sketch. Let e be an arbitrary edge $e = (A, B)$ and $\mathcal{R}(\mathcal{P}_i)$ the lowest region in the hierarchy tree that contains e, then $\mathcal{R}(\mathcal{P}_i)$ is the region where both ports of e are *located*. As defined in Appendix B, the communication of parameters in safe graphs ensures that the rates of ports *located* in a given region do not change during the local iteration of that region. Hence, rates of the ports connected to e do not change during a local iteration of $\mathcal{R}(\mathcal{P}_i)$. Moreover, the firing counts of A and B per local iteration are equal to the local solutions of these actors: $\#_{\mathrm{L}}A$ and $\#_{\mathrm{L}}B$. By construction, these solutions satisfy the balance equation for

edge e. Thus, a local iteration of region $\mathcal{R}(\mathcal{P}_i)$ always brings edge e to its initial state. Because the global pseudo-schedule always brings the local iterations of hierarchical nodes to completion, we see that a global iteration also brings edge e to the initial state.

By construction, the PDN actors and edges satisfy the rate consistency and safety criterion. Therefore the above reasoning also holds for the parameter communication edges, including the edges $(M(p), E(p))$, which model the periods. Because all these edges come back to their initial state, we conclude that a global iteration brings all the periods to completion.

If the interval of every parameter is bounded, the firing counts per global iteration and the rates of actors are bounded as well. Hence, a global iteration uses a bounded memory and, because the graph comes back to the initial state, the required memory of the next iteration is independent of the previous iterations. A schedule can execute an indefinitely long sequence of global iterations. Consequently, an indefinitely long execution of the SPDF graph can be scheduled in a bounded memory. □

D Proof of Liveness

Property 14 (Liveness)**.** *Given a rate-consistent and safe SPDF graph where*

- *all directed cycles are live,*
- *for each parameter p there is a path from its modifier to each user of p in a directed acyclic graph obtained from the SPDF graph by removing only saturated edges,*

there exists a schedule where every actor is fired for an infinite number of times.

Sketch. By definition, every live cycle contains a saturated edge, that is, an edge e satisfying the following inequality:

$$i(e) \geq (r(X, e) \cdot \#X)[p_i^{max}/p_i] \tag{16}$$

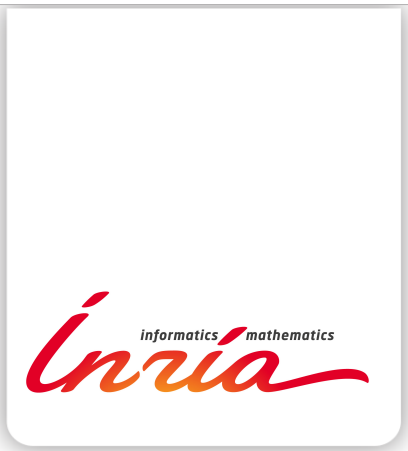
where e is an incoming edge of actor X, p_i denotes the symbolic parameters in the inequality and p_i^{max} their upper bounds, and $X[k/p]$ denotes the substitution of k for p in X. Two important facts hold under the assumptions of the property.

First, the right-hand side of inequality (16) contains an upper bound on the number of tokens consumed in a global iteration. Indeed, even if parameters change within an iteration no one can exceed its own upper bound p_i^{max}. Actually $(r(X, e) \cdot \#X)[p_i^{max}/p_i]$ is the least upper bound on the number of tokens consumed; it is reached by fixing all parameters p_i to p_i^{max} at the beginning of the iteration and not changing them.

Second, when we insert the PDNs into the SPDF graph then all the cycles in the resulting SPDF graph are live. Suppose that the PDNs introduce a non-live cycle in the graph then it contains PDN edges that start at a modifier of some parameter p and ends at some non-PDN user of parameter p. From the second assumption of property 14, all such paths have a corresponding non-PDN path (from the same modifiers to the same users) on a DAG. Furthermore, by definition, the non-live cycle introduced by PDNs consists of non-saturated edges only and so the non-PDN paths of that cycle belong to the DAG. Therefore,

after replacing all PDN paths by paths from the DAG, all edges of the cycle belong to the DAG, which is a contradiction. We conclude that the PDNs cannot introduce non-live cycles.

With these two facts, the statement of the property becomes straightforward. Indeed, in this case, a schedule executing an indefinitely long sequence of global iterations can ignore the edges that satisfy (16). This makes the resulting graph acyclic, which trivially allows such a schedule to bring every iteration to completion by executing the actors of the acyclic graph in a topological order. □



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399