RDF: Reconfigurable Dataflow (extended version)

Pascal Fradet, Alain Girault, Ruby Krishnaswamy,
Xavier Nicollin, Arash Shafiei

RESEARCH REPORT
N° 9227
December 2018
Project-Teams Spades

ISSN 0249-6399   ISRN INRIA/RR--9227--FR+ENG

RDF: Reconfigurable Dataflow (extended version*)

Pascal Fradet[†], Alain Girault[†], Ruby Krishnaswamy[‡],

Xavier Nicollin[†], Arash Shafiei[†‡§]

Project-Teams Spades

Research Report n° 9227 — December 2018 — 19 pages

Abstract: Dataflow Models of Computation (MoCs) are widely used in embedded systems, including multimedia processing, digital signal processing, telecommunications, and automatic control. In a dataflow MoC, an application is specified as a graph of actors connected by FIFO channels. One of the most popular dataflow MoCs, Synchronous Dataflow (SDF), provides static analyses to guarantee boundedness and liveness, which are key properties for embedded systems. However, SDF (and most of its variants) lacks the capability to express the dynamism needed by modern streaming applications. In particular, the applications mentioned above have a strong need for reconfigurability to accommodate changes in the input data, the control objectives, or the environment.

We address this need by proposing a new MoC called Reconfigurable Dataflow (RDF). RDF extends SDF with transformation rules that specify how the topology and actors of the graph may be reconfigured. Starting from an initial RDF graph and a set of transformation rules, an arbitrary number of new RDF graphs can be generated at runtime. A key feature of RDF is that it can be statically analyzed to guarantee that all possible graphs generated at runtime will be consistent and live. We introduce the RDF MoC, describe its associated static analyses, and outline its implementation.

Key-words: Models of computation; Synchronous Dataflow; Reconfigurable systems; Static analyses; Boundednes; Liveness.

* This research report extends the conference version [1] with proofs of the theorems in an appendix. It also uses a different but equivalent formulation of theorem 3.

† Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France. first.last@inria.fr
‡ Orange, France. first.last@orange.com
§ corresponding author

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

RDF : un modèle flot de données reconfigurable (version étendue)

Résumé :

Les modèles de calcul (MoCs) flot de données synchrones sont très utilisés dans les systèmes embarqués pour les applications multimédia, de traitement du signal, de télécommunication et de contrôle automatique. Dans ce style de modèle, une application est spécifiée par un graphe d'acteurs connectés par des liens FIFO de communication. Un des MoCs les plus connus, SDF (pour *Synchronous Dataflow*), permet des analyses statiques qui garantissent l'exécution en mémoire bornée et l'absence d'interblocage, propriétés clés pour les systèmes embarqués. Néanmoins, SDF (et la plupart de ses variantes) ne permet pas d'exprimer la dynamicité requise par les applications embarquées modernes. En particulier, ces applications ont souvent besoin de se reconfigurer pour s'adapter aux changements (par ex., de débit ou de qualité) du flot d'entrée, des objectifs de contrôle ou de l'environnement.

Afin de répondre à ce besoin, nous proposons le MoC RDF (pour *Reconfigurable DataFlow*) qui étend SDF avec des règles de transformations spécifiant comment la topologie et les acteurs du graphe peuvent être reconfigurés dynamiquement. En considérant un graphe SDF initial et un ensemble de règles de transformation, un nombre arbitraire de nouveaux graphes peuvent être produits. La principale qualité de RDF est qu'il peut être analysé statiquement pour garantir que tous les graphes générés dynamiquement s'exécuteront en mémoire bornée et sans interblocage. Nous présentons le modèle RDF, décrivons les analyses statiques associées et décrivons brièvement son implémentation.

Mots-clés : modèles de calcul flot de données synchrones; modèles reconfigurables; analyses statiques; exécution en mémoire bornée, vivacité.

Contents

1 Introduction

Dataflow Models of Computation (MoCs) are convenient for multimedia processing and digital signal processing since they model the application as a network of processing units which is very natural for applications in these domains [2]. One of the most popular dataflow MoCs is Synchronous Dataflow (SDF) [3]. In a nutshell, an SDF graph consists of so-called actors connected by FIFO channels. When it is executed (or fired), an SDF actor consumes a fixed number of data (tokens) on each of its input edges, performs some computation and produces a fixed number of tokens on each of its output edges. These numbers of consumed and produced tokens are *static*, which allows static analyses to check boundedness and liveness of SDF graphs.

Being able to check statically the boundedness and the liveness is a strong advantage, but it comes at the price of forbidding any dynamic changes of the SDF graph. For this reason, several extensions of SDF have been explored such as the parametric production and consumption rates (*e.g.,* PSDF [4], BPDF [5], PiSDF [6]), or allowing limited changes of the topology using scenarios (*e.g.,* SADF [7]). The common points of these variants is to remain statically analyzable [8], a crucial feature for embedded systems. Other MoCs have gone further along the road towards dynamicity (*e.g.,* BDF [9] or DDF [10]), but properties such as boundedness or liveness become undecidable.

One aspect of dataflow MoCs that has not been explored is the dynamic changes to the graph topology. For example, this would be very useful for telecommunication applications (to allocate more pipelines when the number of IP packets increases), embedded computer vision (to change the frame decomposition), automatic control (to change the control law depending on stability criteria).

We propose in this paper a variant of SDF called *Reconfigurable Dataflow (RDF)*. RDF allows dynamic changes to the graph topology thanks to *transformation rules* (expressed as graph rewrite rules) and to a *controller* that applies these rules depending on runtime conditions or measurements. In RDF, the number of graphs that can be produced using transformation rules is potentially *unbounded*. This contrasts with SADF where the number of scenarios is fixed and, in practice, rather small.We show that RDF remains statically analyzable and we propose algorithms to ensure connectivity, boundedness, and liveness of RDF graphs.

The paper is organized as follows. We start by recalling the basic notions of SDF in section 2. RDF is introduced in section 3. Section 4 presents the static analyses ensuring that RDF reconfigurations preserve the connectivity, consistency, and liveness properties. We outline in section 5 the main features of the implementation of RDF. Finally, section 6 presents related work and section 7 concludes. The appendix gathers the proofs of the theorems stated in section 4.

2 Synchronous Dataflow

An SDF graph [3] is a directed graph, where vertices – called *actors* – are functional units. Actors are connected by *edges*, which are 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 (resp. produced) on a given edge at each firing is called the *consumption* (resp. *production*) *rate*. An actor can fire only when *all* its input edges contain enough tokens (*i.e.,* at least the number specified by the consumption rate of the corresponding edge). In SDF, all rates are constant integers known at compile time.

Formally, an SDF graph is defined by a 4-tuple $G = (V, E, \rho, \iota)$ where:

- V is a finite set of actors; among those, we distinguish *source* actors that have no incoming edges, and *sink* actors that have no outgoing edges;

- E is a finite set of directed edges ($E \subseteq V \times V$);
- $\rho : E \to \mathbb{N}\backslash\{0\} \times \mathbb{N}\backslash\{0\}$ is a function that returns for each edge a pair (x, y), where x is the production rate of its origin actor (producer) and y is the consumption rate of its destination actor (consumer);
- $\iota : E \to \mathbb{N}$ is a function that returns for each edge the number of its initial tokens (possibly 0).

When necessary, we will use V_G instead of V to refer to the set of vertices of graph G (and similarly for the other constituents).

Fig. 1 shows a simple SDF graph G_1 with 5 actors. The edge between A_1 and B_1 has a production (resp. consumption) rate of 2 (resp. 3).

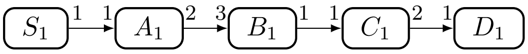

Figure 1: The SDF graph G_1.

Each edge carries zero or more tokens at any moment. The *state* of a dataflow graph is the vector of the number of tokens present on each edge. The *initial state* of a graph is the vector of the number of initial tokens on its edges. For instance, the initial state of G_1 is the vector $[0; 0; 0; 0]$.

The *minimal iteration* of an SDF graph is a smallest set of firings of its actors such that (1) all actors fire at least once, and (2) the graph is returned to its initial state. For instance, the minimal iteration of G_1 is $\{S_1^3, A_1^3, B_1^2, C_1^2, D_1^4\}$, where X^i means that X is fired i times. We note $sol_G(X)$ the number of firings of X in the iteration of the graph G, or $sol(X)$ when no ambiguity can arise. The *basic repetition vector* $\vec{Z}$ indicates the number of firings of actors per minimal iteration. For G_1, it is $\vec{Z}_{G_1} = [3, 3, 2, 2, 4]$ (for actors' ordering $[S_1, A_1, B_1, C_1, D_1]$).

An SDF graph is said to be *consistent* if it admits a repetition vector. The repetition vector is obtained by solving the following *system of balance equations*: each edge $X \xrightarrow{p \quad q} Y$ is associated with the balance equation $sol(X).p = sol(Y).q$, which states that all produced tokens during an iteration must be consumed within the same iteration. The graph is consistent if and only if this system of equations admits a non-null solution [3] (an easy check). An important consequence is that a consistent graph can be executed infinitely with *bounded* memory: all produced tokens are eventually consumed.

The next step is to determine a static order, a *schedule*, in which the firings of the repetition vector can be executed. It is obtained by an abstract computation where an actor is fired only when it has enough input tokens. Such a schedule ensures that the graph returns to its initial state and that each actor is eventually fired. An consistent SDF graph is said to be *live* if it admits a schedule [3].

Among all admissible schedules, we distinguish *single appearance schedules* (SAS) (also called *flat SASs* in [11]) where, once factorized (*i.e.,* any sequence $X; ...; X$ of n consecutive firings of X is replaced by X^n), each actor appears exactly once. For instance, G_1 admits only one SAS: $S_1^3; A_1^3; B_1^2; C_1^2; D_1^4$.

An acyclic SDF graph always admits an SAS, while a cyclic SDF graph admits an SAS if and only if each cycle includes at least one *saturated edge*, that is, an edge (X, Y) that contains enough initial tokens to fire Y at least $sol(Y)$ times. Any SAS S induces a *total order* relation between actors, noted $\prec_S$, such that $X \prec_S Y$ if and only if X appears *before* Y in S. In the context of this paper, we only consider SAS, but RDF can also operate with general schedules.

An SAS can be executed on a single-core chip or on a multi-core chip. On a single-core, it suffices to fire the actors sequentially as specified in the SAS. On a multi-core, each actor must first be allocated to a core, and then on each core an ordering must be chosen among all the actors allocated to it. Actor *allocation* and *ordering* have been the topic of much work. In this paper, we adopt a simple solution called *As Soon As Possible (ASAP)* scheduling, where each actor X is embedded in a private thread `th_X` consisting of the *periodic execution loop* presented in Fig. 2.

```
thread th_X {
   while (true) {
      consume_input_tokens();
      fire_X();
      produce_output_tokens();
   }
}
```

Figure 2: Periodic execution loop for actor X.

The `consume_input_tokens` instruction blocks when (at least) one of the input buffers of X does not contain enough tokens, while the `produce_output_tokens` instruction blocks when (at least) one of the output buffers of X is full. On each core, one such thread `th_X` is started for each actor X allocated to it. This multi-threaded ASAP execution guarantees that the graph can be executed in bounded memory and without deadlock, provided that each buffer has at least the minimal size required for liveness [12].

3 RDF: A Reconfigurable Dataflow MoC

The RDF MoC extends SDF with *actor types* and *transformation rules.* Formally, an RDF application is a pair (G, C) where:

- G is a dataflow graph, basically an SDF graph where each actor is equipped with a type;
- C is a *reconfiguration controller*, a sequence of *transformation programs* that specify *how* an RDF graph may be reconfigured, triggered by conditions that specify *when* the transformations should be applied.

An RDF application can be seen as an initial graph and transformation rules which specify the (potentially infinite) set of possible graphs that can be produced dynamically from the initial graph.

3.1 RDF graph

RDF graphs extend SDF graphs with a set of *actor types* T. A type can be seen as a class of actors. Types allow transformation rules to introduce new actors in the graph as new type instances. An RDF graph is defined as a tuple $G = (V, E, T, \rho, \iota, \tau)$ where V, E, ρ, and ι denote the same items as the ones in SDF (see section 2), T is the finite set of actor types, and $\tau : V \to T$ returns the type of an actor. Although not formally expressed above, it is implicit that actors of the same type have the same numbers of incoming and outgoing edges, the same production and consumption rates, and perform the same computations.

To alleviate the notation, we write $C_1, C_2...$ for actors of type C. The graph of Fig. 1 can be considered as an RDF graph where S_1, A_1, B_1, C_1, and D_1 are actors of types S, A, B, C, and D respectively. It has the same repetition vector and schedules as the SDF version.

3.2 RDF Controller

The controller is specified by a *sequence* of pairs (condition: transformation program): $[cond_1 : P_1; \dots; cond_n : P_n]$.

If one condition $cond_i$ is satisfied, then the controller stops the execution of the RDF graph *at the end of the current iteration*, applies the transformation specified by P_i, and finally resumes the execution. Only one $(cond_i, P_i)$ is selected. If the conditions are not mutually exclusive, the first true condition in the sequence is chosen. Typically, the conditions depend on dynamic non-functional properties (*e.g.,* buffer size, throughput, quality of the input signal, *etc.*). The language for describing these non-functional properties is not part of the MoC nor is it in the scope of this paper.

A transformation program is a combination of *transformation rules* with the following syntax:

$$\begin{array}{rcll} P & ::= & tr & \textit{Transformation rule} \\ & | & P_1 \rhd P_2 : P_3 & \textit{Choice} \\ & | & P^* & \textit{Iteration} \end{array}$$

Individual transformation rules (and their analysis) is the technical heart of RDF. They are presented in the next subsection.

The application of a transformation rule on a given RDF graph G is said to be *successful* if it has *matched* part of G. By extension, an application of a program is considered successful if at least one of the transformation rules it tries to apply has been successful. The choice construction $P_1 \rhd P_2 : P_3$ tries to apply P_1; if it was successful then P_2 is applied next, otherwise P_3 is applied. The iteration P^* applies P as long as it is successful. We write $P_1; P_2$ for the program $P_1 \rhd P_2 : P_2$ which applies P_1 and P_2 in sequence regardless P_1 is successful or not.

To ensure that a controller always preserves the consistency and liveness of the dataflow graphs it transforms, it is sufficient to verify that the initial graph satisfies these two properties *and* that each individual transformation rule preserves them (see section 4).

Another issue, however, is that an iteration P^* may loop infinitely. To guarantee the termination of such iterations, a solution could be to enforce that P decreases some measure (*e.g.,* the number of actors of type T in the graph).

3.3 Transformation rules

An RDF transformation rule is a graph rewrite rule of the form

$$tr : lhs \Rrightarrow rhs$$

which selects a sub-graph matching lhs, and replaces it by the graph specified by rhs. We use the set-theoretic approach of [13] to graph rewriting: the terms lhs and rhs are seen as non empty sets of edges possibly with pattern *variables* matching either types, actor indices, or rates.

As it is standard in programming languages, pattern matching amounts to finding a variable substitution identifying the pattern with a sub-term. In RDF, a pattern lhs matches a sub-graph of G if there is a substitution σ mapping types (resp. indices, rates) variables to actual types (resp. indices, rates) such that the set of edges $\sigma(lhs)$ belongs to G: *i.e.,* $\sigma(lhs) \subseteq G$. The rule removes that sub-graph and replaces it by rhs after substituting its variables by their matches, *i.e.,* $\sigma(rhs)$.

In all examples, we note α, β, ... the pattern variables matching *types*, x, y, ... the pattern variables matching *indices*, and r_1, r_2, ... the pattern variables matching *rates*.

As an example, consider the transformation rule tr_1 depicted in Fig. 3.

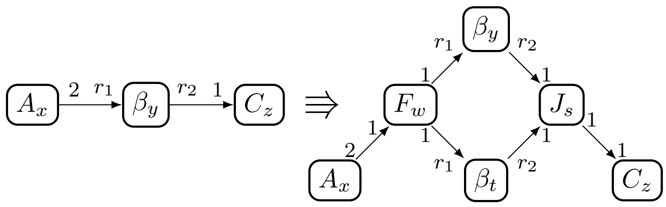

Figure 3: The transformation rule tr_1.

The term β_y matches any actor of any type β, whereas the term A_x matches any actor of type A. When applied to the graph of Fig. 1, the rule matches

$$A_1 \ {}^{2}\!\longrightarrow^{3} B_1 \ {}^{1}\!\longrightarrow^{1} C_1$$

and yields the substitution

$$\sigma = \{x \mapsto 1, \beta \mapsto B, y \mapsto 1, z \mapsto 1, r_1 \mapsto 3, r_2 \mapsto 1\}$$

As a consequence, the rule tr_1 replaces the actor B_1 by a new sub-graph made of B_1 and three new actors of types F, B and J. It transforms the graph of Fig. 1 into the graph of Fig. 4.

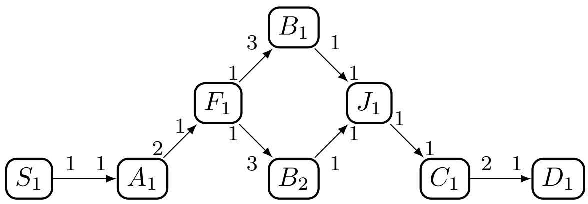

Figure 4: The resulting graph G_2 after applying tr_1 to G_1.

The following conditions should be checked:

(C1) An actor occurring in the *lhs* but not in the *rhs* is *suppressed*. However, to be valid, all incoming and outgoing edges of that actor should occur in the *lhs*. Otherwise, suppressing an actor would create dangling edges. To verify this point, we request the type of removed actors to appear explicitly in the rule. Indeed, when the type is known, the numbers of incoming and outgoing edges are also known and the rule can be checked statically. In the rule tr_1, no actor is suppressed since all matched actors occur in the *rhs*.

(C2) When an actor index variable occurs in the *rhs* but not in the *lhs*, then it yields a *new* actor (instance of the given type) that must therefore be *created*. In contrast, type variables occurring in the *rhs* must always occur (*i.e.,* be defined) in the *lhs*. Indeed, it would be ambiguous to create new instances of unknown types. In the transformation rule tr_1, the terms F_w, β_t and J_s illustrate this case: w, t and s yield new actors, whose types are known because they are either explicit (F, J), or defined in the *lhs* (β).

(C3) Rates and number of incoming and outgoing edges must be consistent with types. This property is easy to check. For instance, no other rate than 2 could decorate the outgoing edge of A_x in tr_1. Rate variables are often superfluous since they are fixed by the type of the actor they are attached to. In such cases, they can be omitted.

A transformation rule $tr : lhs \Rrightarrow rhs$ applied to a graph G can be seen as the set rewrite rule

$$\underbrace{X \cup \sigma(lhs)}_{G} \quad \Rrightarrow \quad \underbrace{X \cup \sigma(rhs)}_{G' = tr(G)} \tag{1}$$

The graph G is seen as the set of edges $X \cup \sigma(lhs)$ where σ is the substitution returned by the matching. When applied to a fresh actor variable in the *rhs*, σ produces a new actor for the necessarily known type, *i.e.,* a new instance of this type. This is the case, for instance, of J_1 or B_2 in Fig. 4.

Initial tokens raise semantic issues. For instance, if a transformation has a *rhs* with initial tokens, we would need a way to specify the origin or values of these tokens. To keep things simple, we allow the initial RDF graph to have tokens but impose that transformations do not manipulate edges with initial tokens.

4 RDF static analyses

The ability to guarantee consistency and liveness is paramount for embedded systems. Hence, improving the expressivity and dynamicity of SDF should not come at the price of losing these static analyses. We present here how connectivity, consistency, and liveness can be analyzed and guaranteed for RDF programs. It is sufficient:

- to check these three properties on the initial graph (SDF static analyses can be reused for that matter);
- to check for each individual transformation rule that, assuming that the considered property holds on the source graph, it still holds on the transformed graph.

An RDF transformation program is said to be valid if all its rules satisfy these checks. Therefore, a valid RDF application transforms, produces, and runs only connected, consistent, and live graphs. We present in turn the conditions that a transformation rule must satisfy to preserve connectivity, consistency, and liveness.

4.1 Connectivity

SDF graphs are always connected, that is, there is an undirected path between every pair of vertices. We write $x \overset{*}{\underset{A}{\longleftrightarrow}} y$ to state that there is an undirected path between actors x and y in graph A. In RDF, a rule removing edges could easily transform a connected graph into several disconnected ones.

Theorem 1 states that, in order to guarantee that connectivity is preserved by the transformation rule $tr : lhs \Rrightarrow rhs$, it is sufficient to ensure that *rhs* is a connected (pattern) graph Note that, on its side, *lhs* can match disconnected subgraphs.

Theorem 1. *Let G be a connected graph and $tr : lhs \Rrightarrow rhs$ be a transformation rule such that*

$$\forall x, y \in rhs, x \overset{*}{\underset{rhs}{\longleftrightarrow}} y$$

then $tr(G)$ is a connected graph.

The proof of Theorem 1, as well as the proofs of Theorems 2 and 3, can be found in the appendix.

Clearly, the transformation tr_1 in Figure 3 preserves connectivity, but the following one

$$\boxed{A_x} \xrightarrow{r_1 \quad r_2} \boxed{B_y} \qquad \Rrightarrow \qquad \boxed{A_x} \xrightarrow{r_1 \quad 1} \boxed{D_z} \quad \boxed{S_w} \xrightarrow{1 \quad r_2} \boxed{B_y}$$

is invalid. Its right-hand term is not connected. Applying this transformation to G_1 would produce two disconnected graphs.

4.2 Consistency

The resulting graph after applying a transformation rule must remain consistent: its system of balance equations should have non-zero solutions. Our condition for consistency, stated in Theorem 2, enforces a *stronger* property: all actors remaining in the transformed graph keep their original solution.

For each transformation rule $tr : lhs \Rrightarrow rhs$, we check that both (pattern) graphs *lhs* and *rhs* are consistent and we compute the (possibly symbolic) solutions of their actors. Actors occurring both in *lhs* and *rhs* should share the same solution. New actors (*i.e.*, occurring only in *rhs*) only need to have a solution.

Theorem 2. *Let G be a consistent graph and let $tr : lhs \Rrightarrow rhs$ be a transformation rule such that lhs and rhs are consistent and*

$$\forall x \in lhs \cap rhs,\ \ sol_{lhs}(x) = sol_{rhs}(x).$$

then $tr(G)$ is consistent.

Note that $sol_A(x)$ denotes the minimal symbolic solution (see [14]) of x in the system of equations corresponding to pattern A.

Example: The transformation rule tr_1 of Fig. 3 preserves consistency. Both the *lhs* and *rhs* are consistent (pattern) graphs and their common actors have the same symbolic solutions. Indeed, the solutions of actors in the *lhs* are

$$sol(x) \qquad sol(y) = \frac{2.sol(x)}{r_1} \qquad sol(z) = \frac{2.r_2.sol(x)}{r_1}$$

and those of actors in *rhs* are: $\quad sol(x) \ \ sol(w) = 2.sol(x)$

$$sol(y) = sol(t) = \frac{2.sol(x)}{r_1} \qquad sol(s) = sol(z) = \frac{2.r_2.sol(x)}{r_1}$$

The common actors x, y and z keep their solutions and the fresh actors w, s, t have also solutions. This rule applied to the graph G_1 yields the consistent graph G_2 (Fig. 4). The actors S_1, A_1, B_1, C_1, and D_1 keep their solutions (3, 3, 2, 2, and 4, respectively) and the solutions of the new actors F_1, B_2 and J_1 are 6, 2, and 2, respectively.

$$\boxed{\alpha_x} \xrightarrow{r_1 \quad 1} \boxed{A_y} \xrightarrow{2 \quad r_2} \boxed{\beta_z} \qquad \Rrightarrow \qquad \boxed{\alpha_x} \xrightarrow{r_1 \quad 3} \boxed{B_w} \xrightarrow{1 \quad r_2} \boxed{\beta_z}$$

Figure 5: The transformation rule tr_2.

On the other hand, the transformation tr_2 in Fig. 5 is invalid. The reason is that, even though *rhs* is consistent, the solution of actor z changes from $\frac{2.r_1.sol(x)}{r_2}$ to $\frac{r_1.sol(x)}{3.r_2}$. We cannot be sure that this solution is a natural number. The transformation applied to G_1 produces a consistent graph but all solutions change ($sol(S_1) = 9$, $sol(B_1) = 1$, *etc.*).

In general, such rules can produce inconsistent graphs. For instance, when applied to the graph of Fig. 6a, tr_2 would produce the inconsistent graph of Fig. 6b. We have $sol(H_1) = 2$ in the initial graph, and yet H_1 has no solution in the transformed graph. The reason is to be found in the edge (E_1, H_1) which enforces a constraint on the solution of H_1 that cannot be seen in the transformation rule.

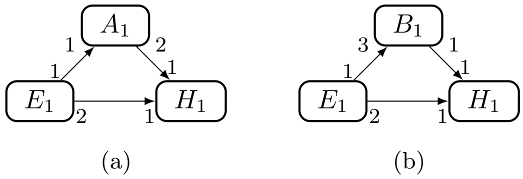

Figure 6: Consistent (a) and inconsistent (b) graphs.

4.3 Liveness

A consistent graph is live if it can be scheduled. We present here conditions to preserve liveness for graphs with single appearance schedules (SAS). The general case (*i.e.,* a schedule exists, but is not an SAS) can also be dealt with, but it is more involved and would require more space to present.

For each transformation rule $tr : lhs \Rrightarrow rhs$, we need to check that rhs is live (acyclic) and that tr does not add a path between common actors of lhs and rhs that did not exist before. These checks ensures that tr does not introduce new cycles.

Theorem 3. *Let G be a live graph with an SAS and $tr : lhs \Rrightarrow rhs$ a transformation rule such that rhs is live and*

$$\forall x, y \in lhs \cap rhs, x \xrightarrow[rhs]{+} y \Rightarrow x \xrightarrow[lhs]{+} y$$

then $tr(G)$ is live and admits an SAS.

The transformation rule tr_1 of Fig. 3 preserves liveness. The rhs does not introduce new paths between actors occurring both in lhs and rhs (*i.e.,* between A_x, β_y and C_z).

On the other hand, the transformation tr_3 in Fig. 7 is invalid. Actor Y_y is connected to Z_z in the rhs but not in the lhs. If the only schedule in the initial graph was one were Z_z needed to be fired before Y_y, then rule tr_3 would produce a deadlocked (*i.e.,* non live) graph.

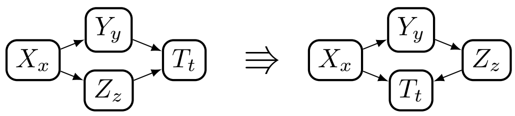

Figure 7: The transformation rule tr_3 (all rates are 1).

Such a case is shown in Fig. 8. The rule tr_3 would transform the live graph of Fig. 8a into the deadlocked graph of Fig. 8b.

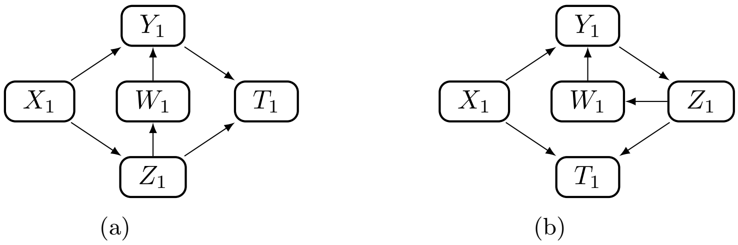

Figure 8: Live (a) and deadlocked (b) graphs (all rates are 1).

5 Implementation

Actors are executed according to an as soon as possible (ASAP) policy. An actor can fire as soon as it has enough tokens on its incoming edges (see Section. 2 and Fig. 2). Actors can therefore execute in parallel independently of each other. Synchronization is ensured by communication buffers. New actors introduced by reconfigurations just need to know their input and output buffers and to follow the execution loop pattern of Fig. 2.

Yet, reconfigurations cannot be performed at any moment. Transforming the dataflow graph in the middle of an iteration or when actors are not in the same iteration would raise many semantic issues. A reconfiguration should only occur in a consistent state, that is, *after* an iteration has completed and the graph has returned to its initial state.

To simplify the presentation, we assume (*i*) that the initial graph has no initial tokens (they could be taken into account but the implementation is more involved), (*ii*) that it has single source and sink actors (every dataflow graph can be transformed to meet this criterion by adding dummy source and sink actors), and (*iii*) that none of the transformation rules change these two actors.

The controller (which runs inside its own thread) continuously watches whether one of its reconfiguration condition is satisfied (see Section 3.2). Whenever this occurs, before applying the associated transformation, the graph must return to its initial state, and all actors must have completed the same iteration. To do this, the source and sink actors keep track of their iteration number and of their number of firings in the current iteration. The controller requests the source actor to answer with its current iteration number k and to stop at the end of that iteration. Then, the controller requests the sink actor to stop at the end of its kth iteration and afterwards, to answer with an acknowledgment. At this point, the controller knows that the graph is in its initial state. All actors have completed their kth iteration; [1] the source actor waits for a signal to resume whereas all others actors are blocked on empty input buffers. The controller performs the reconfiguration and resumes the execution of the source actor (and therefore of the transformed graph altogether). The execution proceeds as before, each actor firing as soon as its incoming edges have enough tokens.

6 Related work

To the best of our knowledge, no existing dataflow MoC allows both the dynamic reconfiguration (in the general sense) of the graph topology and static analyses for boundedness and liveness. Still, several dataflow MoCs allow a limited form of topology changes, including SADF [7] and BPDF [5], while still remaining statically analyzable.

[1] Assumptions (*i*) and (*ii*) ensure that no actor may have already started its $(k+1)$th iteration at this point.

SADF [7] models reconfigurability as a set of pre-defined configurations (called scenarios), coupled with a non-deterministic finite-state machine that specifies the transitions between scenarios. The number of available topologies is statically fixed and specified in the source model. Analyzing an SADF model consists in applying the standard analyses of SDF to each scenario.

BPDF [5] models reconfigurability by adding Boolean conditions to FIFO channels. When a condition switches to false (resp. true) the channel is *disabled* (resp. *enabled*). Boundedness and liveness remain statically analyzable, and static or quasi-static schedules can be produced [15].

Reconfigurability using rewriting rules has also been studied for Petri nets (see [16] for a recent overview). In the general case, reconfigurable Petri nets do not preserve properties such as liveness, boundedness, or reversibility. In [17], a restricted class of transformations (called INRS) is proposed that preserves these properties. It has been applied to design Petri net controllers for the supervision of reconfigurable manufacturing systems. Model checking of reconfigurable Petri nets has been considered by converting the net and the set of rewriting rules into a Maude specification [18]. This approach allows the absence of deadlocks to be verified.

7 Conclusion

In this paper, we addressed the question of dynamic reconfigurations of SDF graphs. To this aim, we introduced the RDF MoC consisting in a dataflow graph (an SDF graph with typed actors) and a controller (a sequence of transformation programs triggered by conditions). The transformation programs determine *how* the RDF graph is reconfigured and the conditions specify *when* these reconfigurations take place. Our RDF MoC provides static analyses to guarantee that reconfigurations preserve boundedness and liveness properties. Finally, we outlined the main characteristics of an RDF implementation.

Further work is needed in two directions. Firstly, a useful application of reconfigurations would be to duplicate lines of computation (*e.g.,* to increase parallelism when computational demand grows). This requires to extend RDF with *variable arity actors* able of (de)multiplexing inputs and outputs for a varying number of computation lines. Secondly, a reconfiguration entails to stop the pipelined execution, to remove or create actors and communication links and, finally, to restart the execution. These costs should be evaluated by implementing RDF on a multi-core platform and using realistic use cases. This knowledge would be particularly useful to tune the conditions for reconfigurations.

References

[1] P. Fradet, A. Girault, R. Krishnaswamy, X. Nicollin, and A. Shafiei, "RDF: Reconfigurable Dataflow," in *DATE 2019 - Design, Automation & Test in Europe Conference & Exhibition*, Florence, Italy, Mar. 2019.

[2] G. Kahn, "The semantics of a simple language for parallel programming," *Information Processing*, vol. 74, pp. 471–475, 1974.

[3] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[4] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Trans. on Signal Processing (TSP)*, vol. 49, no. 10, pp. 2408–2421, 2001.

[5] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur, "BPDF: A statically analyzable dataflow model with integer and boolean parameters," in *International Conference on Embedded Software, EMSOFT'13*, 2013, pp. 1–10.

[6] K. Desnos, M. Pelcat, J.-F. Nezan, S. Bhattacharyya, and S. Aridhi, "PiMM: Parameterized and interfaced dataflow meta-model for MPSoCs runtime reconfiguration," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS'13*. Samos Island, Greece: IEEE, Jul. 2013, pp. 41–48.

[7] M. Geilen, "Synchronous dataflow scenarios," *ACM Trans. on Embedded Computing Systems (TECS)*, vol. 10, no. 2, p. 16, 2010.

[8] A. Bouakaz, P. Fradet, and A. Girault, "A survey of parametric dataflow models of computation," *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 2, Mar. 2017.

[9] J. Buck and E. Lee, "Scheduling dynamic data-flow graphs with bounded memory using the token flow model," in *International Conference on Acoustics, Speech, and Signal Processing, ICASSP'93*, vol. I. Minneapolis (MN), USA: IEEE, Apr. 1993, pp. 429–432.

[10] E. Lee, S. Neuendorffer, and G. Zhou, *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[11] S. S. Battacharyya, E. A. Lee, and P. K. Murthy, *Software Synthesis from Dataflow Graphs*. Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[12] O. Moreira, T. Basten, M. Geilen, and S. Stuijk, "Buffer sizing for rate-optimal single-rate data-flow scheduling revisited," *IEEE Trans. on Computer*, vol. 59, no. 2, pp. 188–201, 2010.

[13] J.-C. Raoult and F. Voisin, "Set-theoretic graph rewriting," in *Graph Transformations in Computer Science*. Springer, 1994, pp. 312–325.

[14] P. Fradet, A. Girault, and P. Poplavko, "SPDF: A Schedulable Parametric Data-Flow MoC (Extended Version)," INRIA, Research Report RR-7828, Dec. 2011. [Online]. Available: https://hal.inria.fr/hal-00666284

[15] V. Bebelis, P. Fradet, and A. Girault, "A framework to schedule parametric dataflow applications on many-core platforms," in *International Conference on Languages, Compilers and Tools for Embedded Systems, LCTES'14*. Edinburgh, UK: ACM, Jun. 2014.

[16] J. Padberg and L. Kahloul, "Overview of reconfigurable Petri nets," in *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig*, 2018, pp. 201–222.

[17] J. Li, X. Dai, Z. Meng, and L. Xu, "Improved net rewriting systems-extended Petri nets supporting dynamic changes," *Journal of Circuits, Systems, and Computers*, vol. 17, no. 6, pp. 1027–1052, 2008.

[18] J. Padberg and A. Schulz, "Model checking reconfigurable Petri nets with Maude," in *Graph Transformation - 9th International Conference, ICGT*, 2016, pp. 54–70.

A Appendix

We recall the following facts and notations:

- A graph is seen as a set of edges and transformations as set rewritings. A transformation $tr : lhs \Rrightarrow rhs$ applied to a graph G consists in finding a substitution σ such that $G = X \cup \sigma(lhs)$. The graph is then rewritten into $tr(G) = X \cup \sigma(rhs)$.
- We write $x \xrightarrow[A]{} y$ for an edge between actors x and y belonging to graph A (set of edges) and use the corresponding transitive closure $x \xrightarrow[A]{+} y$ (resp. reflexive transitive closure $x \xrightarrow[A]{*} y$) to denote paths in A. We write $x \xleftrightarrow[A]{} y$ to denote that there is an edge from x to y or from y to x in graph A. We use the corresponding transitive closure $x \xleftrightarrow[A]{+} y$ (resp. reflexive transitive closure $x \xleftrightarrow[A]{*} y$) to denote an undirected path between x and y in A.
- We say that an actor x belongs to graph A (and write $x \in A$) if there is an edge in A having x as initial or terminal vertex.

Theorem 1. *Let G be a connected graph and $tr : lhs \Rrightarrow rhs$ be a transformation rule such that:*

$$\forall x, y \in rhs, x \xleftrightarrow[rhs]{*} y \qquad (\mathrm{C}^{rhs})$$

then $tr(G)$ is a connected graph.

Proof. Let x and y be two *distinct* actors $\in tr(G)$; we must prove that $x \xleftrightarrow[tr(G)]{+} y$. We consider tr as the set rewriting $G = X \cup \sigma(lhs) \Rrightarrow X \cup \sigma(rhs) = tr(G)$. Note that Cond. (C^{rhs}) implies that forall x, y in $\sigma(rhs)$, we have $x \xleftrightarrow[\sigma(rhs)]{*} y$.

We distinguish the following exclusive cases: **(A)** x and y are in $\sigma(rhs)$; **(B)** x and y are not in $\sigma(rhs)$; **(C)** x is in $\sigma(rhs)$ whereas y is not. The last case ($y \in \sigma(rhs)$ and $x \notin \sigma(rhs)$) is identical to case **(C)**.

Case (A): $x \in \sigma(rhs)$ and $y \in \sigma(rhs)$.

By Cond. (C^{rhs}) we have $x \xleftrightarrow[\sigma(rhs)]{+} y$ for any two distinct actors x and y of rhs. We therefore conclude that $x \xleftrightarrow[tr(G)]{+} y$.

Case (B): $x \notin \sigma(rhs)$ and $y \notin \sigma(rhs)$.

Actors x and y belong to X and therefore to G. Since G is a connected graph we have $x \xleftrightarrow[G]{+} y$. Recall that an actor belonging to lhs but not to rhs is removed from the graph. Therefore neither x nor y belong to $\sigma(lhs)$. The undirected path between x and y in G must start and finish with an edge in X, meaning that it has the following form :

$$x \xleftrightarrow[X]{+} x_1 \xleftrightarrow[\sigma(lhs)]{+} x_2 \xleftrightarrow[X]{+} \dots \xleftrightarrow[\sigma(lhs)]{+} x_n \xleftrightarrow[X]{+} y \qquad \text{with } n \geq 0$$

Since $x_1, \dots, x_n$ belong to X *and* belong to $\sigma(lhs)$, they also belong to $\sigma(rhs)$. Indeed, recall that, by definition of tr, actors occurring in (edges of) X cannot be suppressed by tr.

By Cond. (C^{rhs}), we have $x_1 \xleftrightarrow[\sigma(rhs)]{+} x_n$ and, edges in X being untouched by tr, we have $x \xleftrightarrow[X]{+} x_1 \xleftrightarrow[\sigma(rhs)]{+} x_n \xleftrightarrow[X]{+} y$. We therefore conclude that $x \xleftrightarrow[tr(G)]{+} y$.

Case (C): $x \in \sigma(rhs)$ and $y \notin \sigma(rhs)$.

As in Case **(B)**, y belongs to X hence to G and does not belong to $\sigma(lhs)$. However, either x occurs in $\sigma(lhs)$ or does not. We consider both cases in turn.

Sub-Case (C$_1$): $x \in \sigma(lhs)$.

Since y belongs to the connected graph G, we have $x \underset{G}{\overset{+}{\longleftrightarrow}} y$. This path can be of the following two forms:

$$x \underset{X}{\overset{+}{\longleftrightarrow}} x_1 \underset{\sigma(lhs)}{\overset{+}{\longleftrightarrow}} x_2 \underset{X}{\overset{+}{\longleftrightarrow}} \dots \underset{\sigma(lhs)}{\overset{+}{\longleftrightarrow}} x_n \underset{X}{\overset{+}{\longleftrightarrow}} y \qquad \text{with } n \geq 0$$

$$x \underset{\sigma(lhs)}{\overset{+}{\longleftrightarrow}} x_1 \underset{X}{\overset{+}{\longleftrightarrow}} x_2 \underset{\sigma(lhs)}{\overset{+}{\longleftrightarrow}} \dots \underset{\sigma(lhs)}{\overset{+}{\longleftrightarrow}} x_n \underset{X}{\overset{+}{\longleftrightarrow}} y \qquad \text{with } n \geq 0$$

On the one hand, since x_n belongs to X and to $\sigma(lhs)$, it also belongs to $\sigma(rhs)$ and, by hypothesis, x also belongs to $\sigma(rhs)$. Therefore, by Cond. (C^{rhs}), $x \underset{\sigma(rhs)}{\overset{+}{\longleftrightarrow}} x_n$, hence $x \underset{tr(G)}{\overset{+}{\longleftrightarrow}} x_n$.

On the other hand, edges in X being untouched by tr, we have $x_n \underset{X}{\overset{+}{\longleftrightarrow}} y$, hence $x_n \underset{tr(G)}{\overset{+}{\longleftrightarrow}} y$. Putting both facts together, we therefore conclude that $x \underset{tr(G)}{\overset{+}{\longleftrightarrow}} y$.

Sub-Case (C$_2$): $x \notin \sigma(lhs)$.

In that case x is a fresh actor created by tr. But there must be another actor x_i in $\sigma(rhs)$ belonging also to $\sigma(lhs)$. Otherwise, it would mean that all actors in $\sigma(lhs)$ were suppressed by tr. This would only be possible if they were not linked to any other actor in G, so if lhs had matched the whole graph. Since y belongs to $tr(G)$ and not to $\sigma(rhs)$ this cannot be the case.

As a consequence, by Cond. (C^{rhs}) there is a path $x \underset{\sigma(rhs)}{\overset{+}{\longleftrightarrow}} x_i$. We can use the same reasoning as in Sub-Case **(C$_1$)** to show that there is a path $x_i \underset{tr(G)}{\overset{+}{\longleftrightarrow}} y$. By transitivity, we therefore conclude that $x \underset{tr(G)}{\overset{+}{\longleftrightarrow}} y$. □

Theorem 2. *Let G be a consistent graph and let $tr : lhs \Rrightarrow rhs$ be a transformation rule such that lhs and rhs are consistent and*

$$\forall x \in lhs \cap rhs,\ sol_{lhs}(x) = sol_{rhs}(x) \qquad (\mathrm{C}^{sol})$$

then $tr(G)$ is consistent.

Note that we write $sol_A(x)$ to denote the *minimal* solution of actor x in the system of equations corresponding to the graph (or pattern pattern) A. If A is a SDF graph, this solution is an integer; if A is a pattern (with possibly parametric rates) the solution can also be computed and is, in general, symbolic. The reader may consult [14] for a definition of the minimal symbolic solutions of parametric systems of equations.

Proof. First, consider a graph G (a set of edges between actors) than can be partitioned into two disjoint subsets of edges (two subgraphs) G_1 and G_2, such that $G = G_1 \cup G_2$ and $G_1 \cap G_2 = \emptyset$. As far as balance equations are concerned, the system of equations of G is the union of the systems of equations of G_1 and G_2. If G is consistent (*i.e.,* its system of balance equation has a solution) then clearly G_1 and G_2 are also consistent. For any actor x such that $x \in G_1$ or $x \in G_2$, $sol_G(x)$ is also a solution of x in G_1 or G_2. This solution may be not minimal for the system of balance equations of G_1 or G_2 because G may enforce additional constraints, but we have:

$$\exists k, \forall x \in G_i, sol_G(x) = k sol_{G_i}(x), \quad i \in \{1, 2\}$$

Dually, if G_1 and G_2 are consistent and if there exist two integers k_1 and k_2 such that, for any common actor x, $k_1 sol_{G_1}(x) = k_2 sol_{G_2}(x)$, then G is also consistent. The solutions $k_1 sol_{G_1}(x)$ and $k_2 sol_{G_2}(x)$ are also solutions for the system of equations of G. The minimal (*i.e.,* coprime) pair of integers k_1 and k_2 gives the minimal solutions for G.

Lemma 1 formalizes this fact.

Lemma 1. *Let G be an SDF graphs partitioned into G_1 and G_2. We have:*

$$G \text{ is consistent } \Leftrightarrow \begin{cases} & G_1 \text{ is consistent} \\ \wedge & G_2 \text{ is consistent} \\ \wedge & \exists (k_1, k_2) \in \mathbb{N} \times \mathbb{N}, \forall x \in G_1 \cap G_2 \\ & k_1 sol_{G_1}(x) = k_2 sol_{G_2}(x) \end{cases}$$

Now, let G be a consistent graph, let tr be a transformation rule satisfying Cond. (C^{sol}) described as:

$$\underbrace{X \cup \sigma(lhs)}_{G} \Rrightarrow \underbrace{X \cup \sigma(rhs)}_{tr(G)}$$

The condition $sol_{lhs}(x) = sol_{rhs}(x)$ means that the common minimal symbolic solutions of the balance of the graphs *lhs* and *rhs* are syntactically equal. It follows that any graph matching the *lhs* (resp. *rhs*) using a substitution σ accepts the solutions $\sigma(sol_{lhs}(x))$ (resp. $\sigma(sol_{rhs}(x))$. These concrete solutions may not be minimal though.

Since G is consistent, by Lemma 1, X and $\sigma(lhs)$ are also consistent and there exist k_1 and k_2 such that, for any actor x in $X \cap \sigma(lhs)$, we have:

$$k_1 sol_X(x) = k_2 sol_{\sigma(lhs)}(x)$$

Furthermore, let (k_1^m, k_2^m) be the minimal (coprime) pair of (k_1, k_2). We thus have:

$$\forall x \in X,\ sol_G(x) = k_1^m sol_X(x) \quad \text{and} \quad \forall x \in \sigma(lhs),\ sol_G(x) = k_2^m sol_{\sigma(lhs)}(x)$$

Cond. (C^{sol}) ensures that the solutions of common actors in $\sigma(lhs)$ and $\sigma(rhs)$ are the same. The common actors between X and $\sigma(rhs)$ belong also to $\sigma(lhs)$ (the others are fresh actors), therefore k_1^m and k_2^m can be used to equalize the solutions. As a result, for any shared actor between X and $\sigma(rhs)$, we have:

$$k_1^m sol_X(x) = k_2^m sol_{\sigma(rhs)}(x)$$

and, by Lemma 1, the graph $tr(G)$ is consistent. Furthermore, since k_1^m and k_2^m are coprime, they correspond to the minimal solutions of $tr(G)$:

$$\forall x \in X,\ sol_{tr(G)}(x) = k_1^m sol_X(x) \quad \text{and} \quad \forall x \in \sigma(rhs),\ sol_{tr(G)}(x) = k_2^m sol_{\sigma(rhs)}(x)$$

$\square$

Remark: We could have chosen a weaker condition for Theorem 2, namely $\exists k, sol_{lhs}(x) = k sol_{rhs}(x)$. This would allow a transformation to weaken some constraints (*e.g.,* by removing edges) so that the minimal solutions of the *rhs* are possibly smaller than the solutions of *lhs*. In that case, consistency would be still preserved, the solutions of all actors would remain valid, but they might not be minimal anymore.

Theorem 3. *Let G be a live graph with an SAS and $tr : lhs \Rrightarrow rhs$ a transformation rule such that*

$$rhs \text{ is live and } \forall x, y \in lhs \cap rhs, x \xrightarrow[rhs]{+} y \Rightarrow x \xrightarrow[lhs]{+} y \qquad (\mathrm{C}^{live})$$

then $tr(G)$ is live and admits an SAS.

Proof. We first prove (Lemma 2) that a transformation respecting Cond. (C^{live}) cannot create new cycles.

Lemma 2. *Let $tr : lhs \Rrightarrow rhs$ a transformation rule satisfying Cond.* (C^{live}) *then*

$$\forall G,\ x \xrightarrow[tr(G)]{+} x \Rightarrow x \xrightarrow[G]{+} x$$

Proof. Consider the rewriting $G = X \cup \sigma(lhs) \Rrightarrow X \cup \sigma(rhs) = tr(G)$, there are two cases:

1. $x \in X$

   The path $x \xrightarrow[tr(G)]{+} x$ is made of alternating subpaths from X and $\sigma(rhs)$. It can take one of the following forms depending on whether the path starts and terminates with a subpath in X or in $\sigma(rhs)$:

$$\begin{array}{c}
x \xrightarrow[X]{+} x_1 \xrightarrow[\sigma(rhs)]{+} x_2 \xrightarrow[X]{+} \dots \xrightarrow[\sigma(rhs)]{+} x_n \xrightarrow[X]{+} x \\
x \xrightarrow[X]{+} x_1 \xrightarrow[\sigma(rhs)]{+} x_2 \xrightarrow[X]{+} \dots \xrightarrow[X]{+} x_n \xrightarrow[\sigma(rhs)]{+} x \\
x \xrightarrow[\sigma(rhs)]{+} x_1 \xrightarrow[X]{+} x_2 \xrightarrow[\sigma(rhs)]{+} \dots \xrightarrow[\sigma(rhs)]{+} x_n \xrightarrow[X]{+} x \\
x \xrightarrow[\sigma(rhs)]{+} x_1 \xrightarrow[X]{+} x_2 \xrightarrow[\sigma(rhs)]{+} \dots \xrightarrow[X]{+} x_n \xrightarrow[\sigma(rhs)]{+} x
\end{array}$$

   Actors $x, x_1, \dots, x_n$ belong to X: $x \in X$ by hypothesis and each x_i is either the initial or terminal vertex of an edge in X. Subpaths in X, $x_i \xrightarrow[X]{+} x_j$, are unchanged by tr and therefore occur also in G. For subpaths in $\sigma(rhs)$, $x_i \xrightarrow[\sigma(rhs)]{+} x_j$, we know that $x_i \in X$ and $x_j \in X$. Note that an actor in $\sigma(rhs)$ is either a fresh actor created by tr, or belongs also to $\sigma(lhs)$. Since $x_i \in X$ and $x_j \in X$, then x_i and x_j must also belong $\sigma(lhs)$. In that case, Cond. (C^{live}) enforces that the path $x_i \xrightarrow[\sigma(lhs)]{+} x_j$ exists. Therefore, in each of the above cases, we have $x \xrightarrow[G]{+} x$.

2. $x \notin X$

   The path $x \xrightarrow[tr(G)]{+} x$ can take one of the two following forms:

$$\begin{array}{l}
x \xrightarrow[\sigma(rhs)]{+} x_1 \xrightarrow[X]{+} x_2 \xrightarrow[\sigma(rhs)]{+} \dots \xrightarrow[X]{+} x_n \xrightarrow[\sigma(rhs)]{+} x \\
x \xrightarrow[\sigma(rhs)]{+} x
\end{array}$$

   In the first case, we apply the same reasoning as before. All x_is (except x) belong to X and $x_1 \xrightarrow[G]{+} x_n$. We also have $x_n \xrightarrow[\sigma(rhs)]{+} x_1$ with $x_1 \in X$ and $x_n \in X$. Since x_1 and x_n also belong to $\sigma(lhs)$, Cond. (C^{live}) ensures that $x_n \xrightarrow[\sigma(lhs)]{+} x_1$. Hence we have $x \xrightarrow[G]{+} x$.

The second case is impossible. Indeed, Cond. (C^{live}) enforces *rhs* to be live and since *tr* can only manipulate edges without initial tokens, $\sigma(rhs)$ must be acyclic.

□

We now return to the proof of Theorem 3. A consistent SDF graph admits an SAS (or a flat SAS following the terminology of [11]) iff all cycles have a *saturated* edge, that is, an edge with enough initial tokens to permit its destination actor to complete all its firings in this SAS for one iteration. Indeed, consider a cycle $x_0 \longrightarrow x_1 \longrightarrow \dots x_n \longrightarrow x_0$ in a graph G with an SAS. Then, the first actor of that cycle occurring in the SAS, say x_i, must perform all its firings consecutively before any other (in particular x_{i-1}) can fire. The edge $x_{i-1} \longrightarrow x_i$ must therefore be saturated with initial tokens.

Since transformation *tr* does not introduce new cycles (Lemma 2), nor removes (matches) any edge with initial tokens, nor changes the solution of actors (Theorem 2), all cycles remain with a saturated edge in $tr(G)$. We therefore conclude that $tr(G)$ is live and admits an SAS. □

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