RDF: A Reconfigurable Dataflow Model of Computation

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

RDF: A Reconfigurable Dataflow Model of Computation

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

RESEARCH REPORT
N° 9439
Dec. 2021
Project-Team Spades

ISSN 0249-6399 ISRN INRIA/RR--9439--FR+ENG

RDF: A Reconfigurable Dataflow Model of Computation

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

Project-Team Spades

Research Report n° 9439 — Dec. 2021 — 33 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 first and 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 and when 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 present its implementation and some experimental results.

Key-words: Models of computation; Synchronous dataflow; Reconfigurable systems; Graph rewriting; Static analyses; Boundedness; Liveness

* Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
† Orange Labs

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

RDF: Un modèle de calcul flot de données reconfigurable

Résumé :

Les modèles de calcul (MoCs) flot de données synchrones sont très utilisés dans les systèmes embarqués et 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 RDF (pour *Reconfigurable DataFlow*) un MoC qui étend SDF avec des règles de transformations spécifiant comment la topologie du graphe flot de données peut être reconfiguré 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, les analyses statiques associées, sa mise en œuvre et quelques expérimentations.

Mots-clés : Modèles de calcul; SDF; systèmes reconfigurables; réécriture de graphes; analyses statiques; exécution en mémoire bornée; vivacité

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 [1]. One of the first and most popular dataflow MoCs is Synchronous Dataflow (SDF) [2]. In a nutshell, an SDF graph consists of so-called actors connected by FIFO channels. When it is executed (or fired in SDF terminology), an SDF actor consumes a fixed number of data (referred as 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 fixed integers, which allows static analyses to check boundedness and liveness of SDF graphs.

Being able to check statically these properties is a strong advantage of SDF, but it comes at the price of forbidding dynamic changes of the graph. For this reason, several extensions of SDF have been explored, such as the parametric production and consumption rates (*e.g.,* PSDF [3], BPDF [4], PiSDF [5]), allowing limited changes of the topology using scenarios (*e.g.,* SADF [6]) or the possibility to dynamically enable and disable the edges of the graph (BPDF [4]). The common points of these variants is to remain statically analyzable [7], a crucial feature for embedded systems. Other MoCs have gone further along the road towards dynamicity (*e.g.,* BDF [8] or DDF [9]), 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 to be handled increases), embedded video processing (to add filters as the light conditions change), 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. 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 describe conditions on transformations 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 Sec. 2 before presenting the RDF MoC in Sec. 3. Sec. 4 describes the conditions on transformations and static analyses ensuring that RDF reconfigurations preserve connectivity, consistency, and liveness. We present in Sec. 5 the main features of the implementation of RDF and provide some experimental results in Sec. 6. Finally, Sec. 7 presents related work and Sec. 8 concludes. The appendix gathers the proofs of the theorems stated in Sec. 4.

This article extends and revises the work presented in [10]. Since then, RDF has been equipped with variable arity actors, an implementation has been developed and experiments have been conducted. Sections 3.3, 5 and 6 are novel and other sections have been extended or rewritten. Explanations and examples have been added throughout. Additional details can be found in a PhD thesis [11].

2 Synchronous Dataflow

An SDF graph [2] is a directed graph, where vertices – called *actors* – are functional units. Actors are connected by *edges*, which represent FIFO communication channels. The atomic execution of a given actor – referred to as 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 non-null 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: $E \subseteq V \times V$;
- $\rho : E \to \mathbb{N}^* \times \mathbb{N}^*$ 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 and B has a production rate of 2 and a consumption rate of 3.

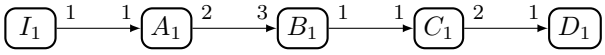


Figure 1: The SDF graph G_1.

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 $(I^3, A^3, B^2, C^2, D^4)$, 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 $\vec{Z}_{G_1} = [3, 3, 2, 2, 4]$ assuming the actor ordering $[I, A, B, C, D]$).

An SDF graph is said to be *consistent* if it admits a repetition vector. The repetition vector is obtained by solving a *system of balance equations*. Each balance equation of that system corresponds to an edge of the graph. The 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 its system of balance equations admits a non-null solution [2], which is easy to check statically. An important advantage 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 of the actors' firings, a *schedule*, in which the firings in the repetition vector can be executed without deadlock. 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. A consistent SDF graph is said to be *live* if it admits a schedule [2].

Among all admissible schedules, we distinguish *single appearance schedules* (SAS)[1] where, once factorized[2], each actor appears exactly once. For instance, G_1 admits exactly one SAS: $\{I^3; A^3; B^2; C^2; D^4\}$.

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) containing enough initial tokens to fire Y at least $sol(Y)$ times. 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 is first allocated to a core, and then on each core an ordering is chosen among all the actors allocated to it. In this paper, we consider *As Soon As Possible (ASAP)* scheduling, where each actor X is embedded in a private thread `act_X` consisting of the *periodic execution loop* presented in Fig. 2.

```
thread act_X {
  while (true) {
    consume_input_tokens();
    fire();
    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 `actor_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 (which is easy to compute statically [13]). The dataflow semantics guarantees *functional determinism* whatever the order in which the actors are fired [1]. Moreover, provided enough cores and sufficiently large buffers, ASAP scheduling permits to achieve the maximal throughput.

3 RDF: A Reconfigurable Dataflow MoC

The RDF MoC extends SDF with *transformation rules*, *actor types*, and *explicit ports*. A transformation rule describes how the current dataflow graph is modified. Actors and communication links can be moved, removed, and/or added. Adding new actors motivates the introduction of actor types. A type can be seen as a class of actors having the same functionality. Types allow transformation rules to add new actors in the graph as new type instances. For instance, a video application may require the dynamic introduction of several noise filter actors at different places in the graph. This may be done by introducing new actors in the graph, instances of the noise filter type. Transformation rules and type instances allow the number of actors and possible RDF graphs to be *unbounded*. RDF also introduces explicit actor ports to allow transformation rules to select specific edges more easily. For instance, ports allow two outgoing edges of the same actor and bearing the same producing rate to be discriminated.

An RDF application is specified as a pair (G, C) where:

[1] Also called *flat SASs* in [12].

[2] Any sequence $X; ...; X$ of n consecutive firings of X is replaced by X^n.

- G is a dataflow graph, basically an SDF graph where each actor is equipped with a type;
- C is a *reconfiguration controller*, which consists of a set of transformation rules that specifies *how* an RDF graph may be transformed, and of a reconfiguration program using conditions to specify *when* the transformation rules should be applied.

An RDF application starts by executing its initial graph, until a condition is true and some transformation rules are applied, resulting in a new graph that is executed, and so on and so forth. The transformation rules allow a potentially infinite number of graphs to be produced dynamically from the initial graph.

3.1 RDF graph

RDF graphs extend SDF graphs with a set of *actor types* T and a notion of *ports*. Formally, an RDF graph is defined as a tuple $G = (T, V, E, \iota)$ where

- $T \subseteq Id_T \times \mathbb{N} : k_i \times \mathbb{N} : k_o \times ([1, k_i] \to \mathbb{N}^*) \times ([1, k_o] \to \mathbb{N}^*)$ is a finite set of types consisting of a unique identifier, a number of input and output ports, and two functions returning the rate associated with input and outpout ports respectively. A type $t = (i, k_i, k_o, f_i, f_o)$ is composed of
  - an identifier i (a capital letter in this article);
  - two integers k_i and k_o denoting its numbers of input and output ports respectively;
  - two functions f_i and f_o returning the rate associated with their input and output port argument respectively.

  The auxilary functions *idof*, *nbin*, *nbout*, *finr*, and *foutr* return respectively the identifier, number of input ports, number of output ports, input rate function, and output rate function of their type argument. For instance, $finr(T)(nbin(T))$ returns the rate of the last input port of type T;
- $V \subseteq T \times \mathbb{N}^*$ is a finite set of actors, each one consisting of a type ($\tau \in T$) and an index ($i \in \mathbb{N}^*$). In the following, we use capital letters for type identifiers and we denote an actor of type X and index i by X_i. The functions *typeof* and *indof* return the type and index of their actor argument. Among actors, we distinguish *source* actors that have no incoming ports, and *sink* actors that have no outgoing ports;
- $E \subseteq (V \times \mathbb{N}^*) \times (V \times \mathbb{N}^*)$ is a finite set of directed edges. The edge $((a, i), (b, j))$ connects the ith output port of actor a to the jth input port of actor b. We will also denote an edge between a and b in the graph G by $a \underset{G}{\longrightarrow} b$ and the fact that actors a and b are connected in G (*i.e.,* $a \underset{G}{\longrightarrow} b$ or $b \underset{G}{\longrightarrow} a$) by $a \underset{G}{\longleftrightarrow} b$;
- $\iota : E \to \mathbb{N}$ is a function that returns, for each edge, the number of its initial tokens (possibly 0).

We consider only well-formed graphs, that is, graphs properly connected and typed. In RDF, we check that initial graphs are well-formed and that transformations preserve well-formedness. Formally,

Definition 1 (Well-formedness)**.** *An RDF graph is* well formed *if it is (weakly) connected, all its actors are fully linked, and all its edges are valid.*

An RDF graph G is (weakly) connected if there exists an undirected path between any two actors a and a', which we write $a \xleftrightarrow[G]{*} a'$. Formally,

Definition 2 (Graph connectivity)**.** *An RDF graph $G = (T, V, E, \iota)$ is (weakly) connected if* $\forall (a, a') \in V \times V, \quad a \xleftrightarrow[G]{*} a'$

Actors are *fully linked* if all their ports belong to a unique edge. Formally,

Definition 3 (Actors fully linked)**.** *An RDF graph $G = (T, V, E, \iota)$ has its actors fully linked if*

$$\forall b \in V,\ \forall\ 1 \leq i \leq nbin(typeof(b)),\ \forall\ 1 \leq o \leq nbout(typeof(b)),$$
$$\exists!(a, o') \in V \times \mathbb{N}^*, ((a, o'), (b, i)) \in E \quad \wedge \quad \exists!(c, i') \in V \times \mathbb{N}^*, ((b, o), (c, i')) \in E$$

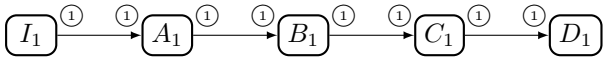
Edges are *valid* if they connect actors only through ports permitted by the actors' type. Formally,

Definition 4 (Edge validity)**.** *An RDF graph $G = (T, V, E, \iota)$ has valid edges if*

$$\forall ((a, o), (b, i)) \in E, 1 \leq o \leq nbout(typeof(a)) \wedge 1 \leq i \leq nbin(typeof(b))$$

To facilitate the reading, RDF graphs are often represented as in SDF with implicit ports and explicit rates. The graph of Fig. 1, can be seen as an RDF graph where S_1, A_1, B_1, C_1, and D_1 are actors with index 1 and types S, A, B, C, and D respectively. It has the same repetition vector and schedules as its SDF counterpart.

Another representation closer to the formal definition, with explicit ports, is given in Fig. 3.



with $foutr(I)(1) = 1, finr(A)(1) = 1, foutr(A)(1) = 2, finr(B)(1) = 3, foutr(B)(1) = 1, \ldots$

Figure 3: The RDF graph corresponding to G_1 with explicit ports.

3.2 Reconfiguration Controller

The RDF controller specifies when and how the dataflow graph is modified. The basic operations are *transformation rules* which are specified as graph rewrite rules. The *reconfiguration program* combines these transformation and specifies the conditions for their application. We present these two components in turn.

3.2.1 Transformation rules

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

$$tr : lhs \Rrightarrow rhs,$$

which selects a sub-graph matching the pattern *lhs*, and replaces it by the graph specified by *rhs*. We use the set-theoretic approach of [14] 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 ports.

Pattern variables require us to introduce the set $\mathcal{V}_t$ of type variables, the set $\mathcal{V}_i$ of actor index variables, and the set $\mathcal{V}_p$ of port variables. With these sets, we define:

- the set of pattern nodes as $\tilde{V} \subseteq (T \cup \mathcal{V}_t) \times (\mathbb{N}^* \cup \mathcal{V}_i)$, and
- the set of pattern edges as $\tilde{E} \subseteq (\tilde{V} \times (\mathbb{N}^* \cup \mathcal{V}_p)) \times (\tilde{V} \times (\mathbb{N}^* \cup \mathcal{V}_p))$.

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, ports) variables to actual types (resp. indices, ports) such that the set of edges $\sigma(lhs)$ belongs to G: *i.e.,* $\sigma(lhs) \subseteq G$. The rule removes the matched 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. For instance, A_x matches any actor of type A and α_x matches any actor.

For the same reasons as we represent graphs with rates instead of explicit ports, we use patterns matching rates instead of ports. In the case of ambiguity, we may use explicit port index (such as ① in Fig. 3) or port variables p_1, p_2, ... in transformation rules.

As an example, consider the transformation rule tr_1 depicted in Fig. 4.

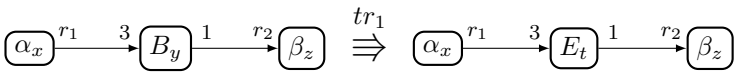


Figure 4: The transformation rule tr_1.

The terms α_x and β_x matches any actor of any type, whereas the term B_y matches any actor of type B.

When applied to the graph of Fig. 1 (seen as an RDF graph), the *lhs* of the rule matches the sub-graph

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

and yields the substitution

$$\sigma = \{\alpha \mapsto A, x \mapsto 1, r_1 \mapsto 2, y \mapsto 1, r_2 \mapsto 1, \beta \mapsto C, z \mapsto 1\}.$$

The rule tr_1 replaces the actor B_1 by a new actor of type E. When a transformation introduces a new actor, its index is chosen so that the actor's name is fresh. Since no E actor occurs in G_1, the variable t in tr_1 can be instantiated with index 1. As a result, tr_1 transforms the RDF graph G_1 of Fig. 1 into the RDF graph G_2 of Fig. 5.

Figure 5: The resulting graph $G_2 = tr_1(G_1)$.

The numbers of incoming and outgoing ports must be consistent with types. Actors occurring in the *lhs* and *rhs* must have the same number of edges in both parts (Cond. (**C1**)). Actors occurring only in *lhs* or *rhs* must be fully linked: they must have explicit types and all their ports connected (Conds (**C2**) and (**C3**)). The following conditions must be respected by each transformation rule:

- (**C1**) Actors occurring in both sides must have the same edges and ports connected in the *rhs* and in the *lhs*. For an actor with an unknown type (*i.e.,* denoted by a pattern variable), since it was fully linked before the transformation, it remains so afterwards.

  In tr_1, both α_x and β_z keep the same edges and rates in *lhs* and *rhs*.

- (**C2**) An actor occurring in the *lhs* but not in the *rhs* is *suppressed*. To be valid, all incoming and outgoing edges of that actor should appear 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 tr_1, the actor that is suppressed has type B and has only one ingoing and outgoing edge. It must be checked that actors of type B have only one ingoing and outgoing edges.

- (**C3**) When an actor index variable occurs in the *rhs* but not in the *lhs*, it represents a *new* actor (instance of the given type) that must therefore be *created*. We enforce the type of such actors to be explicit and check that it is fully linked.

  In tr_1, E_t represents a new actor with an explicit type E. It must be checked that actors of type E have only one ingoing and outgoing edges.

These conditions are easily checked syntactically on each transformation. Additional constraints are required to guarantee that transformations preserve connectivity, consistency, and liveness. They are presented in Sec. 4.

RDF transformations can be formalized by representing a dataflow graph G as set of edges, and the rule $tr : lhs \Rrightarrow rhs$ applied to G 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 the set of edges $X \cup \sigma(lhs)$ where σ is the substitution returned by the pattern matching. The resulting graph $G' = tr(G)$ is G where the sub-graph $\sigma(lhs)$ has been replaced by $\sigma(rhs)$. The context X (*i.e.,* the graph or set of edges "surrounding" the matched part) remains unchanged.

Initial tokens raise semantic issues. For instance, if the *rhs* of a transformation rule contains 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 edges with initial tokens but impose that transformations do not manipulate them. In other words, an edge with initial tokens cannot be matched nor created.

3.2.2 Reconfiguration programs

RDF transformation rules may be composed freely. The controller describes how to compose transformations into reconfiguration programs and when to apply these programs. A controller is specified by a *sequence* of pairs "(condition : reconfiguration program)" separated by semicolons:

$$[cond_1 : P_1; \ldots; cond_n : P_n].$$

If a condition $cond_i$ is satisfied, then the controller stops the execution of the RDF graph *at the end of the iteration*, applies the transformations specified by P_i, and finally resumes the execution. Only one pair $(cond_i, P_i)$ 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.

The simplest option for specifying reconfiguration programs is to consider them made of a single transformation. This is the language we used to perform our experiments (see Sec. 6).

Many other, more expressive, options are possible. We describe in the following one such option. A reconfiguration program can be a combination of *transformation rules* with the following syntax:

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

The application of a transformation rule on a given RDF graph G is said to be *successful* if it has *matched* a sub-graph 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 P_1 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 can also write $P_1; P_2$ for the program $P_1 \rhd P_2 : P_2$, which try to apply P_1 then P_2 in sequence regardless of the success or not of P_1.

To ensure that a controller always preserves connectivity, consistency, and liveness of the dataflow graphs it transforms, it is sufficient to verify that the initial graph satisfies these properties *and* that each individual transformation rule preserves them. This will be the topic of Sec. 4.

This expressive language raises another issue, however: 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 Variable arity actors

An important application of RDF is to permit dataflow programs whose parallelism level can vary dynamically when needed by the environment (for instance according to some performance measure). Consider the dataflow graph G_3 of Fig. 6 that applies a filter F_1 on a flow of image macroblocks.

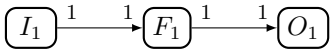


Figure 6: The RDF graph G_3 with a single computing line.

When the resolution of the images in the video flow increases, it might be needed to increase the computational power and change the graph G_3 into the new graph G_4 of Fig. 7.

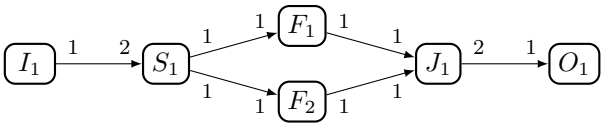


Figure 7: The RDF graph G_4 with two computing lines.

In Fig. 7, the split actor S_1 reads two image blocks and distributes them towards the two filters F_1 and F_2, while the join actor J_1 reads the two resulting blocks and passes them to actor O_1. Provided enough hardware computing resources, the actors $F1$ and F_2 can be fired in parallel and the throughput is thus improved compared to the initial RDF program.

Should a third computation line be needed, one would have to introduce new split and join actors so as to distribute and read the three blocks, as shown in Fig. 8. The split actor S_1' now reads three image blocks and distributes them to its outputs, so its type differs from that

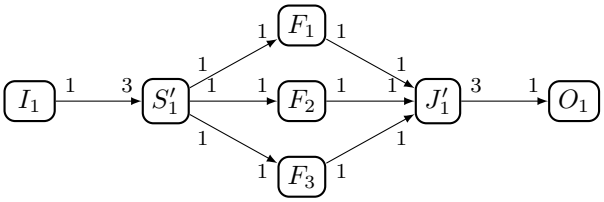


Figure 8: The RDF graph G_5 with three computing lines. The split actor S_1' differs from the split actor S_1 from Fig. 7.

of S_1. Similarly, the type of the join actor J_1' differs from that of J_1. The graphs G_4 and G_5 illustrate the complexity of modifying the graph topology to increase (and reduce) dynamically the number of computing lines:

- It requires an arbitrary number of split actor types like that of S_1 in Fig. 7 and S_1' in Fig. 8 to distribute an arbitrary number of tokens read from its single input to all its outputs (and similarly for the join actor types like that of J_1 and J_1').

- It requires an arbitrary number of transformation rules, because the rule used to increase the number of computing lines from 1 to 2 differs from the rule used to increase the number of computing lines from 2 to 3 differs from the rule used to increase the number of computing lines from 2 to 3 (and similarly for the rules decreasing the number of computing lines).

To solve these issues, RDF provides a specific type named V for *variable arity* actors, shown in Fig. 9. To the best of our knowledge, RDF is the first dataflow MoC to offer such a variable arity actor.

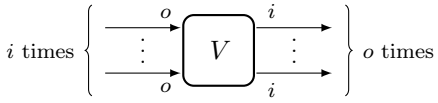


Figure 9: The variable arity actor type V (most general form).

When an actor of type V has i incoming edges and o outgoing edges, the consumption rate on each of its incoming edges is o and the production rate on each of its outgoing edges is i. At each firing, such an actor consumes and produces $i \cdot o$ tokens. It does not perform any computation; it just reads tokens from its input ports and distributes them evenly on its output ports.

The variable arity actor V_k is associated with two unique parameters: i_k representing both the input rates and the number of output ports, and o_k representing both the output rates and the number of input ports. Solving the system of balance equations is performed *symbolically* with those parameters, as in parametric variants of SDF [7]; the resulting iteration is also parametric. In this way, consistency is checked for all possible values of parameters.

Whenever a transformation adds or removes one or more edges of a variable arity actor V_k, the following modifications must occur:

- the number of ports $nbin(V_k)$ and $nbout(V_k)$ are updated;

- the value of the two parametric rates i_k and o_k are updated such that $i_k = nbout(V_k)$ and $o_k = nbin(V_k)$;

- ports are implemented as *lists* and adding a new edge involves adding a new port at the end of the list, while removing the edge of the ℓth port makes the $\ell+1$th port become the ℓth and so on; finally, the functions *finr* and *foutr* are updated such that:
  - $\forall 1 \leq \ell \leq i_k, \mathit{finr}(V_k)(\ell) = o_k$,
  - $\forall 1 \leq \ell \leq o_k, \mathit{foutr}(V_k)(\ell) = i_k$.

  To allow this updating, the functions *nbin*, *nbout*, *finr*, and *foutr* must now take as their first argument an *actor* instead of a *type*. Indeed, before introducing variable arity actors, all the actors of a given type T had exactly the same number of input and output ports. This is not the case anymore with variable arity actors.

Variable arity actors entail additional conditions on transformation rules. Indeed, suppressing a variable arity actor would require to select all its edges, the number of which cannot be statically known. Creating a new variable arity actor is also difficult since it involves introducing new parameters that play a role in the solutions of connected actors. We therefore enforce the following new condition on rules:

- (**C4**) Variable arity actors cannot be suppressed nor be created by transformation rules. All variable arity actors must appear in the initial graph.

Furthermore, a variable arity actor V_k is fully linked if (i) the value of its parameter i_k (resp. o_k) is equal to the number of its outgoing (resp. incoming) edges, and (ii) it has at least one incoming and one outgoing edge. Condition (i) is enforced by construction, as explained above. To enforce Condition (ii) for all possible RDF graphs generated dynamically, we add the following new condition:

- (**C5**) If a transformation rule removes an incoming (resp. outgoing) edge from a variable arity actor, then this actor must occur in the *rhs* with still at least one incoming (resp. outgoing) edge.

The rule t_{dec} in Fig. 11 removes a line of treatment and an outgoing and incoming egde of two variable arity actors (S_1 and J_1) while respecting that constraint.

Two special cases of this generic type V are particularly useful: the *split* actor type S and the *join* actor type J, depicted in Fig. 10 (and already seen in Fig. 11). The split actor type S has a single input with rate q, which is also the number of its outputs whose rates are 1. In other words, S is a special case of V where $p = 1$. The join actor type J has p inputs with rates 1 and a single output whose rate is p. In other words, J is a special case of V where $q = 1$. For an actor S_k (resp. J_k), we note its corresponding parameter s_k (resp. j_k). We only use splits and joins as variable arity actors in our examples and experiments.

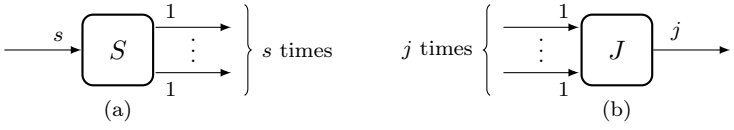


Figure 10: Variable arity actor types: (a) split S and (b) join J.

Other form of variable arity actors might be considered. Consider for example an actor *Sum* whose functionality consists in summing a collection of integer inputs. Such an actor could be represented as a variable arity actor returning the sum of of its inputs on its single output

whereas its number of inputs could be changed freely by transformations. Actually, any actor the functionality of which is defined on a list of inputs and/or outputs could be implemented as this form of variable arity actor. This generalization is presented in details in [11].

3.3.1 A complete RDF application

Fig. 11 shows an RDF application, with its initial graph G_6, two transformation rules tr_{inc} and tr_{dec}, and a controller using two conditions for reconfigurations. This RDF application uses two variable arity actors, namely the split actor S_1 and the join actor J_1. The reconfiguration controller applies the transformation rule tr_{inc} as soon as the throughput of the last actor P_1 drops below a threshold value equal to 20, and the transformation rule tr_{dec} as soon as the number of tokens present in the buffer from V_1 to S_1 drops below a threshold value equal to 10 and there are few data to process.

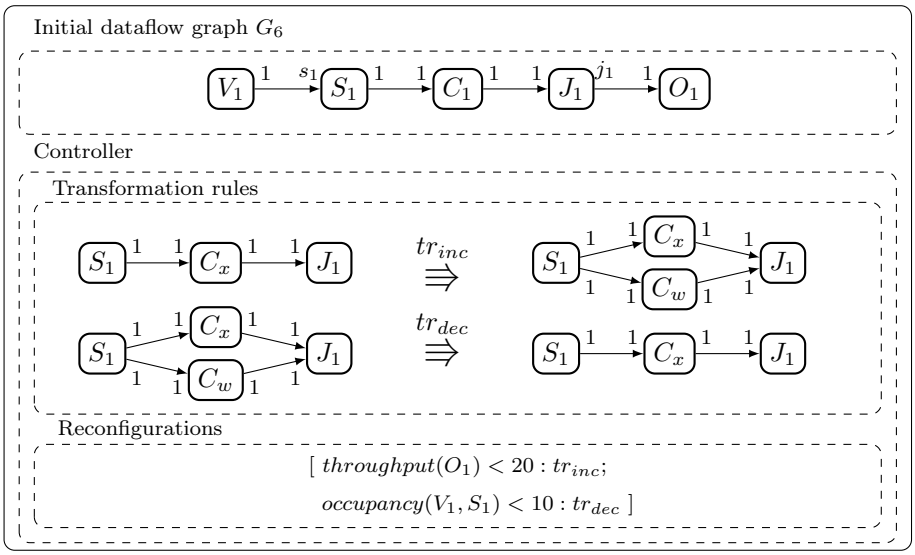


Figure 11: An adaptive video processing application.

Applying tr_{inc} twice to G_6 yields the graph G_7 shown in Fig. 12, with $j_1 = 3$ and $s_1 = 3$, to be compared with the previous graph G_5 from Fig. 8. Rule tr_{inc} creates a new output port for actor S_1 (and similarly for J_1). This is only allowed for variable arity actors of course. As explained above, the ports are implemented as lists and the functions *nbin*, *nbout*, *finr*, and *foutr* must be updated each time an input or output port is created or suppressed by a transformation rule. The parametric iteration for the initial dataflow graph of Fig. 11 is $(I_1^{s_1} S_1 F_1 F_2 J_1 O_1^{j_1})$. When the transformation tr_{inc} is applied, the values of the parameters are updated and are propagated to all the actors with a number of firings per iteration depending on them (here, I_1 and O_1).

This RDF application implements an adaptive video processing application performing edge detection on a video stream with variable image quality. It will be used as a case study in Sec. 6.

Note that the following initial graph

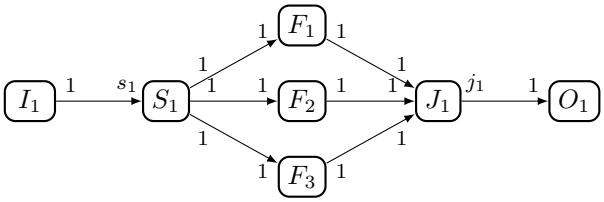
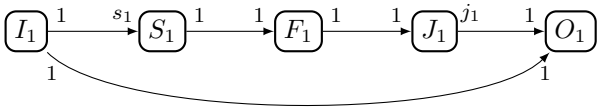


Figure 12: The RDF graph $G_7 = tr_{inc}(tr_{inc}(G_6))$; we have $j_1 = 3$ and $s_1 = 3$.



would be inconsistent since the new edge between I_1 and O_1 requires that the parameters j_1 and s_1 to be equal. Event if that condition is satisfied in the initial graph, it might be invalidated by some transformation (*e.g.,* adding an edge between S_1 and a new sink actor). A simple extension would be to allow consistency checking (see Sec. 4.2) to produce such additional constraints and to statically check that all the transformations respect these constraints. In the following, we do not consider this extension and ensure consistency by checking that transformations do not change the solutions of existing actors.

4 RDF static analyses

The ability to guarantee statically consistency and liveness is of paramount importance for embedded systems. For this reason, improving the expressivity and dynamicity of SDF should not come at the price of losing these static analyses. This is the main technical issue of RDF. We present in this section how well-formedness, consistency, and liveness can be analyzed and guaranteed for RDF applications.

Of course, we want to guarantee that these three properties hold for all possible RDF graphs a given RDF application can generate at run-time. Our key contribution here is to show that it is sufficient:

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

An RDF application is said to be valid if all its transformation rules satisfy these checks. Therefore, a valid RDF application transforms, produces, and runs only well-formed, consistent, and live graphs. We present in turn the conditions that a transformation rule must satisfy to preserve these three properties.

4.1 Well-formedness

We show that a well-formed graph (see Def. 1) remains so after a transformation respecting the previous conditions **(Ci)**. We have to show that all actors remain fully linked, all edges remain valid, and the graph remains weakly-connected.

All actors in the transformed graph remain fully linked since:

- Cond. (**C1**) ensures that all the other actors of the *rhs* keep the same ports connected, so they remain fully linked.
- Cond. (**C3**) ensures that newly introduced actors are fully linked.
- By construction, all the ports of a variable arity actor remain connected after a transformation (see Sec. 3.3).
- Finally, all the actors not present in the transformation rule remain untouched in the graph.

All edges in the *rhs* (new and remaining) can be checked to connect valid ports. Cond. (**C2**) ensures that removing an actor cannot create dangling edges. Therefore, all edges occurring in the graph remain valid.

SDF graphs are always connected, that is, there always exist an undirected path between every pair of vertices. In contrast, an RDF transformation 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 ($x \overset{*}{\underset{rhs}{\longleftrightarrow}} y$ states that there is an undirected path between x and y in *rhs*). Note that, in contrast to *rhs*, *lhs* may be disconnected, and therefore match disconnected subgraphs.

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

$$\forall x \neq y \in rhs, x \overset{+}{\underset{rhs}{\longleftrightarrow}} y \qquad (\mathbf{C}^{conn})$$

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

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

Well-formedness follows from the preservation of complete linkage, validity, and connectivity.

Corollary 1. *Let G be a well-formed graph and $tr : lhs \Rrightarrow rhs$ be a transformation rule satisfying the syntactic constraints of Sec.3.2.1 and ($\mathbf{C}^{conn}$), then $tr(G)$ is a well-formed graph.*

Clearly, the transformation tr_1 in Fig. 4 on page 8 preserves connectivity, but the rule tr_2 shown in Fig. 13 is invalid because its *rhs* is not a connected graph.

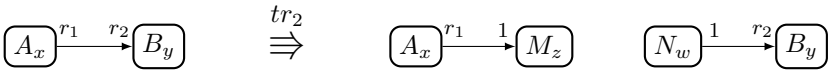


Figure 13: The invalid transformation rule tr_2.

Applying this transformation to G_1 of Fig. 1 would produce two disconnected graphs.

4.2 Consistency

The graph resulting from a transformation rule must remain consistent, meaning that its system of balance equations should have *non-null solutions*. Our condition for consistency, stated in Theorem 2, enforces a *stronger* property: all actors remaining in the transformed graph must 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 have exactly the same solution. New actors (*i.e.,* occurring only in *rhs*) only need to have a non-null solution, which is ensured by the fact that *rhs* must be consistent.

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*

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

then $tr(G)$ is consistent.

Note that $sol_{pat}(x)$ denotes the *minimal* symbolic solution (see [15]) of x in the system of equations corresponding to the pattern graph *pat*. If *pat* is a pure SDF graph, then this solution is an integer. If *pat* has pattern variables matching rates, then the solution can also be computed and is, in general, symbolic. If *pat* has variable arity actors, then it also contains actors with parametric solutions. It is quite simple to deal with symbolic systems of equations and to define their minimal symbolic solutions [15].

Example: The transformation rule tr_1 of Fig. 4 (p. 8) preserves consistency. Indeed, both the *lhs* and *rhs* are consistent pattern graphs, and their common actors have the same symbolic solutions. Moreover, the solutions of the *lhs* actors are:

$$sol_{lhs}(x) \qquad sol_{lhs}(y) = \frac{r_1\, sol_{lhs}(x)}{3} \qquad sol_{lhs}(z) = \frac{r_1\, sol_{lhs}(x)}{3\, r_2}$$

while those the *rhs* actors are:

$$sol_{rhs}(x) \qquad sol_{rhs}(t) = \frac{r_1\, sol_{rhs}(x)}{3} \qquad sol_{rhs}(z) = \frac{r_1\, sol_{rhs}(x)}{3\, r_2}$$

The actors common to the *lhs* and *rhs* (x and z) keep their solutions, while the fresh actor t has a non-null solution. Besides, since $sol_{rhs}(t) = sol_{lhs}(y)$ it is an integer solution.

Applied to the graph G_1 from Fig. 1, it yields the consistent graph $G_2 = tr_1(G_1)$ shown in Fig. 5. The actors I_1, A_1, C_1, and D_1 keep their solutions (3, 3, 2, and 4, respectively), while the solution of the new actor E_1 is 2.

On the contrary, the transformation rule tr_3 of Fig. 14 is invalid. The reason is that, even though *rhs* is consistent, the actor y with solution $\frac{r_1\, sol(x)}{3}$ is replaced by actor u with solution $\frac{r_1\, sol(x)}{5}$, so we cannot be sure that this new solution is an integer.

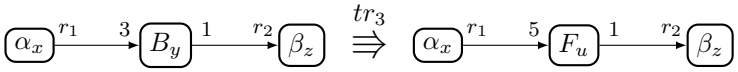


Figure 14: The invalid transformation rule tr_3.

For instance, the graph $tr_3(G_1)$ is consistent but some of the solutions change, with $sol(I_1)$ and $sol(A_1)$ becoming 5.

Rules such as tr_3 can produce inconsistent graphs. For instance, when applied to the graph G_8 of Fig. 15a, tr_3 would produce the inconsistent graph G_9 of Fig. 15b. The reason is that the edge (E_1, H_1) enforces a constraint on the system of balance equations that does not appear in the transformation rule alone.

Figure 15: (a) Consistent graph G_8. (b) Inconsistent graph $G_9 = tr_3(G_8)$.

Note that we could have chosen a weaker condition for Theorem 2, namely

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

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 could remain the same although they would not be minimal anymore.

4.3 Liveness

A consistent graph is live if it can be scheduled. We present here conditions on transformation rules so that they 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. Recall also that, as stated in Sec. 3.2.1, transformation rules do not match nor create edges with initial tokens.

For each transformation rule $tr : lhs \Rrightarrow rhs$, it suffices to check that *rhs* is live (*i.e.,* acyclic) and that tr does not add a path between common actors of *lhs* and *rhs* that did not exist before. These two conditions ensures that tr cannot introduce new cycles in the graph.

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

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

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

The transformation rule tr_1 of Fig. 4 (p. 8) preserves liveness. Indeed, its *rhs* is live (it admits the schedule $[\alpha_x^{3r_2}; E_t^{r_1 r_2}; \beta_z^{r_1}]$) and it does not introduce new paths between actors occurring both in *lhs* and *rhs* (namely between α_x and β_z).

On the other hand, the transformation tr_4 in Fig. 16 is invalid.

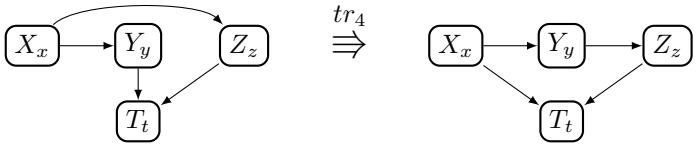


Figure 16: The transformation rule tr_4. All rates are 1.

Indeed, actor Y_y is connected to Z_z in the *rhs* but not in the *lhs*. If the only schedule in the initial graph is one where Z_z needs to be fired before Y_y, then rule tr_4 would produce a deadlocked (*i.e.,* non live) graph. Such a case is shown in Fig. 17.

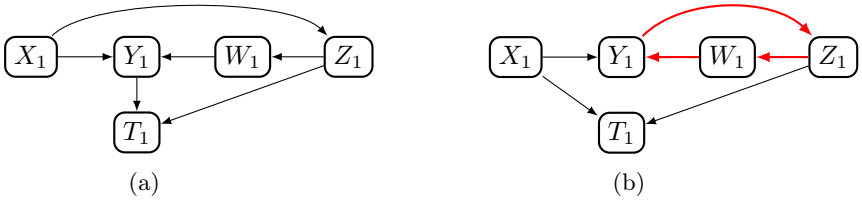


Figure 17: (a) Live graph G_{10}. (b) Deadlock graph $G_{11} = tr_4(G_{10})$. All rates are 1.

The rule tr_4 transforms the live graph G_{10} of Fig. 17a into the deadlocked graph G_{11} of Fig. 17b, where the new (and blocking) directed cycle created by tr_4 is highlighted in red.

5 Implementation

We have implemented a prototype of RDF to perform experiments, to evaluate the reconfiguration costs, and to explore its practicability. In this section, we present the main characteristics of our prototype. In particular, we describe (i) how an RDF application is executed in normal mode, *i.e.,* between reconfigurations; (ii) the steps needed to perform a reconfiguration; (iii) the kinds of conditions the controller may use; (iv) how the pattern matching of a transformation is implemented efficiently, and finally (v) how to deal with the placement of actors on a multi-core architecture when actors can be added or removed dynamically.

5.1 Standard execution

The initial graph is implemented by creating each actor as an instance of its type, and by allocating a circular buffer for each of its output ports. For an actor A and an output port with rate p, the size of the allocated buffer is $p \cdot sol(A) \cdot sizeof(token)$. This is enough to achieve maximal throughput [13], but smaller bounds are known for special classes of graphs [16, 17]. The types of tokens as well as the values of the initial tokens, which are not part of the MoC, must be specified in the application. The types of the tokens allow to compute their size. The initial tokens are pushed in the circular buffer they belong to. Then, the communication links (the edges of the RDF graph) are created by providing to each input port a reference to the buffer it reads from.

Depending on the architecture (*e.g.,* single or multiple servers), actors can communicate (*i.e.,* read and write buffers) through shared memory or message passing. Our prototype runs on a single multi-core processor and uses only buffers in shared memory to communicate. Finally, one thread is created for each actor as well as for the controller.

Actors are executed according to an as soon as possible (ASAP) policy, meaning that each actor fires as soon as it has enough tokens on all its incoming edges. When it does so, it extracts from each of its input buffers a number of tokens equal to the input rate of this buffer, it processes them according to its functionality, and finally it writes output tokens into its output buffers. Note that at this step it may have to wait to have enough room in its output buffers.

Provided enough resources, all actors can run in parallel independently of each other. Synchronization is ensured by communication buffers. Using the ASAP schedule and properly sized buffers, the maximal throughput is reached.

5.2 Reconfigurations

Reconfigurations cannot be performed at any moment. Indeed, transforming the dataflow graph in the middle of an iteration, or when all actors are not in the same iteration, would raise many semantic issues. Therefore, a reconfiguration should only occur when the RDF graph is in a coherent state, that is, after an iteration has completed and the graph has returned to its initial state (meaning implicitly that all actors have completed the same iteration).

Our prototype uses reconfiguration programs (see Sec. 3.2) made of a single transformation rule. The controller (which runs inside its own thread) is thus of the form

$$[cond_1 : lhs_1 \Rrightarrow rhs_1; \ldots; cond_n : lhs_n \Rrightarrow rhs_n]$$

It periodically watches whether one of its reconfiguration condition $cond_i$ is satisfied. In our prototype and experiments, the reconfiguration conditions are mainly performance metrics (throughput, latency, buffer occupancy) measured at runtime. Many others criteria (*e.g.,* arity of split actors, internal variables of an actor, absolute time or delays) could be considered as well. Whenever a condition is true, the controller retrieves the rule corresponding to this condition and checks whether the *lhs* matches the current graph. If so, the transformation can be applied and the reconfiguration starts. Whenever several conditions are true, only the first matching rule in the controller list is selected.

As explained above, before applying a transformation, the graph must return to its initial state, and all actors must have completed the same iteration. To implement this, all actors keep track of their iteration number and of their number of firings within the current iteration. Since actors run in separate threads, at a given time they may not necessarily belong to the same iteration. Here are the main steps of a reconfiguration:

- When the controller has decided to apply a transformation rule, it prompts all actors for their iteration number; it then computes the maximum iteration number received and asks the actors to proceed until the end of this maximum iteration.

- On their side, all the actors stop at the end of their current iteration when they are prompted for their iteration number, *i.e.,* they finish all their firings but not not start a new iteration; when they receive from the controller the maximum iteration number, they either do nothing (if that number was indeed their own last iteration number) or they resume firing until they reach the end of the maximum iteration (otherwise). Then, they all send an acknowledgment to the controller.

- When all actors have stopped and sent their acknowledgment to the controller, the graph is its initial state and the transformation can be applied. The subgraph matching the *lhs* is replaced by the instantiated *rhs*. This may involve removing actors and deallocating the buffers suppressed by the rule, and creating new actors with fresh names and threads, allocating new buffers for the created actors, and connecting them.

- Finally, the controller asks all actors to resume their execution. The computation proceeds as before, each actor firing as soon as its incoming edges have enough tokens.

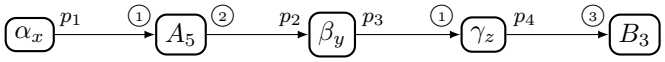
Other, more local, options for reconfiguration could be implemented as well. For instance, stopping only the matched actors while ensuring that the matched edges are empty seems feasible and more efficient.

5.3 Pattern matching

Graph pattern matching is, in general, a NP-complete problem [18] that involves costly graph traversals with potentially many backtrackings. However, in our context of dynamic graph reconfiguration, this operation should be performed as fast as possible, ideally with a time complexity linear in the size of the *lhs* of the rule under consideration and without any backtracking.

To achieve this, we enforce that a *lhs* has at least one fully named actor *i.e.,* neither an actor variable such as α_x nor an actor type with an index variable such as B_x which can serve as a root. Further, we enforce that all edges of the *lhs* can be traversed starting from such roots by following explicit ports. Informally, each edge of the *lhs* must be reachable from a root actor via a *non-ambiguous* (undirected) path. That path is non unambiguous, because either the outgoing or incoming port of each edge is a fully identified one, *i.e.,* not a port variable.

Consider, for instance, the following pattern which may appear as the *lhs* of a transformation rule:



This pattern has two explicitly named actors than can be directly selected in the graph: A_5 and B_3. From these roots, α_x, β_y, and γ_z and the four edges of the *lhs* can be selected unambiguously. Indeed, the edges (α_x, A_5) and (A_5, β_y) can be selected from A_5 by following the input port 1 and output port 2 of A_5 respectively. The edge (γ_z, B_3) can be selected (and actor γ_z determined) by following the input port 3 of B_3, and then the edge (β_z, γ_z) from the input port 1 of γ_z.

If the actor A_5 was not named (*e.g.,* A_x), then the pattern matching would have to consider all typed A actors of the graph. Similarly, if the output port of A_5 was a variable, then the pattern matching would have to consider all possible ports. In both cases, this may involve failure and backtracking.

Note that our constraints can be relaxed. For instance, if the actor type A had a single input port, then the pattern would not need to make it explicit. Similarly, if the second output port of type A was the only one to have the rate 4, then the pattern could make use that rate instead. The key idea here is that the pattern matching should always be able to proceed without performing choices.

Our approach guarantees that pattern matching can be achieved by traversing the pattern without backtracking, *i.e.,* with a time complexity linear in the size of the *lhs*. In practice, we have not noticed a loss of expressive power while observing these constraints. It is always possible to make patterns precise enough by introducing dummy named actors to act as pointers on the graph and as roots in patterns.

5.4 Placement strategy

Actors should be placed to maximize parallelism and minimize communication costs (which may ne high on distributed architectures). With more actors than resources, one should also take care of load balancing. A simple heuristics is to place actors created by transformations on the core that minimize the load and communication cost. These can be expressed in terms of the execution and production costs of the actor during an iteration, values given by the type and solution of the actor.

Our experiments were conducted on a multi-core, single socket server, where communication costs remain small. Preliminary experimental results showed that the previous heuristics was not significantly better than Linux's Completely Fair Scheduler (CFS), even with transformations

drastically changing the initial graph. We then relied on CFS, by running each actor in a separate thread with equal priority. When a transformation creates new actors, the Linux scheduler appears to place these new threads to optimize load balancing.

6 Experimental results

We experimented our prototype on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 12 cores and Linux. We present an experimental evaluation of reconfiguration costs, show typical RDF transformations that change the throughput, and describe a small case study. All experiments consider graphs with a single source and sink actors.

Instead of measuring the throughput (number of iterations per time unit) $\mathcal{T}_G$ of a graph G, we consider a closely related measure: the number of tokens produced by the graph per time unit denoted by $\mathcal{H}_G$. For a graph with a single sink actor O_1, this measure is equal to the number of tokens consumed by O_1 per time unit.

The maximal throughput of an SDF graph is determined by the actor whose execution takes the most time in an iteration. For an acyclic graph with a set of actors V it is equal to:

$$\mathcal{T}_G = \frac{1}{\max_{v \in V} sol(v) \cdot t(v)}$$

For a graph with a single sink actor consuming n tokens per iteration, the number of tokens per time unit is

$$\mathcal{H}_G = n \cdot \mathcal{T}_G$$

This measure is more informative than the throughput since the graph and the computation performed during an iteration may change dynamically. Consider, for instance, the initial graph and transformation tr_{inc} of Fig. 11. That transformation does not change the throughput of the initial graph since a fully parallel ASAP iteration takes the same time. However, the number of tokens produced (and consumed) double.

In the following, we often use the term throughput to refer to the number of tokens produced.

6.1 Reconfiguration costs

An important point to evaluate is the cost of applying a transformation and the global reconfiguration cost. Indeed, RDF would lose part of its interest for streaming applications if reconfiguring takes too long. This cost can be decomposed in two parts:

- the cost of the transformation itself, *i.e.,* matching the *lhs* and replacing it by the *rhs*, possibily creating/removing actors and communication links.
- the cost of pausing the graph execution and restarting it until it reaches again its steady state and maximal throughput.

In order to measure the transformation costs, we considered the following dual transformation rules:

$$I \to O \Rrightarrow I \to A_{x_1} \to ... \to A_{x_n} \to O$$

$$I \to A_{x_1} \to ... \to A_{x_n} \to O \Rrightarrow I \to O$$

for various values of n. Experiments show that the matching and transformation costs are linear in the size of the rule. This was expected since there is no backtracking while matching and

(de)allocating actors and buffers is main part of the costs. The transformation costs range from around $1ms$ for $n = 10$ to $4ms$ for $n = 40$.

To evaluate the total reconfiguration costs we used initial graphs of the form

$$I \to A_{x_1} \to ... \to A_{x_n} \to O$$

and the dummy simple transformation $I \to A_1 \Rrightarrow I \to A_1$ The difference of duration between execution of the graph with that single transformation and the duration without reveals the cost of pausing and restarting the graph. For that setting it is around $100ms$.

In conclusion, the cost of a reconfiguration remains low enough to be used in a video streaming application, as shown in the case study later. If it was higher, a solution to make the reconfiguration seamless would be to introduce output buffers to continue the streaming and prevent glitches during reconfigurations.

6.2 Synthetic transformations

A standard use of the RDF model is to increase the throughput of an application when needed. For instance, a change of resolution could suggest to add or remove parallel levels of computation in a video streaming application.

We use the initial graph G_{10} of Fig. 18 (all unspecified rates are 1) with a split and join actors. The execution times of I_1 and O_1 are $10ms$ each, S_1 and J_1 take $2ms$ each, and A_1 takes $50ms$.

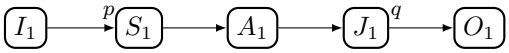


Figure 18: The dataflow graph G_{10}.

Two transformation rules tr_5 and tr_6 are used (see Fig. 19). At time instances $5s$ and $10s$, the transformation rule tr_5 is applied and whereas tr_6 is applied at time 20s and 25s.

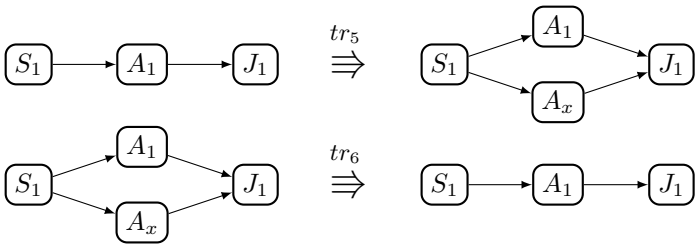


Figure 19: Transformation rules tr_5 and tr_6

The throughput of the graph is shown in Fig. 20. As it was expected, after the first and second transformation, the throughput increases from 20 to nearly 40 and 60 tokens/second. By applying the second transformation twice, the throughput returns to its initial value. If the throughput is not exactly multiplied by 2 and then 3 it is of course due to the light overhead of the split and join actors which have to distribute and gather tokens. Provided sufficient resources, These two rules are sufficient to adapt the application to any throughput demand. Other dynamic MoC like SADF would have to plan for a fixed number of configurations beforehand.

We use this kind of transformation in our case study presented next.

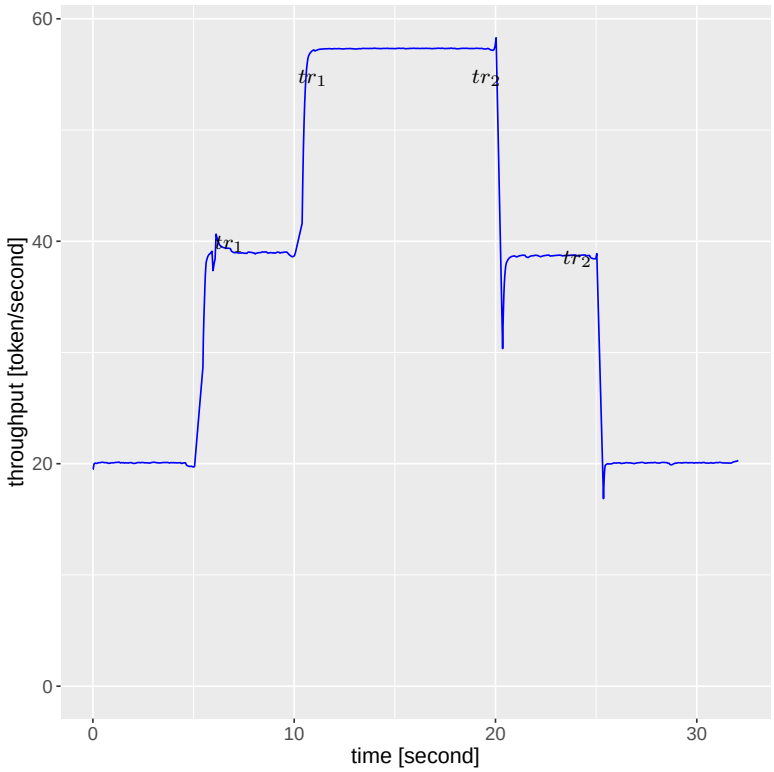


Figure 20: Throughput of G_{10} applying successively tr_5 and tr_6 transformations

6.3 Case study

We have used RDF to implement Canny edge detection, an application which captures a video stream from a source, decodes it, detects the edges in the decoded images, and finally displays images reduced to edges with a constant frame rate. The input video quality can change dynamically. When the quality increases, the processing (decoding & edge detection) takes more time and a static application may fail to maintain the same frame rate. We show that using transformation rules triggered by conditions on throughput, our RDF application can accommodate dynamic changes of video quality. This application has already been introduced in Fig. 11 in Sec. 3.3.1 .

We have implemented this example only partially. Our application reads the video stream from a file and we simulate the increase of processing time using a dummy actor whose execution time artificially increases. The RDF dataflow graph contains 6 actors (see Fig. 21): V_1 captures and decodes the input video stream, the decoded images are send to S_1 a split actor that dispatches p images to its p output ports. The actor C_1 receives an image and extracts its edges using a canny edge detector, D_1 is the dummy actor with a dynamically increasing execution time, J_1 is a join actor, and P_1 displays the output edge image.

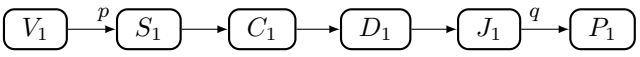


Figure 21: The initial RDF graph G_c of the case study (Canny edge detection)

The transformation in Fig. 22 adds one line of treatment and increases the arity (and parameters) of S_1 and J_1. The controller applies this transformation when the throughput goes below $20fps$.

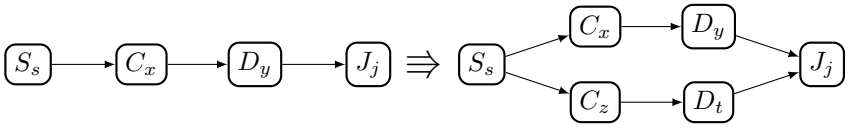


Figure 22: Transformation tr_c

In our experiment, the execution time of actor D_1 increases from $30ms$ to $60ms$ at the $100th$ iteration and then to $120ms$ at the $200th$ iteration. Without any transformation (*e.g.,* using SDF), the throughput decreases from $25fps$ to $17fps$ and then to $8fps$.

In Fig. 23, we see the throughput measured at the sink actor. At the beginning, the actor V_1, whose execution time is $42ms$, is the most costly actor in an iteration and determines the throughput $\mathcal{H}_G = \frac{1}{0.042} = 24fps$. When the execution time of actor D_1 increases to $60ms$, it becomes the most costly actor; the throughput tends to $\mathcal{H}_G = \frac{1}{0.06} = 16.6fps$. Around iteration 110, the throughput goes below the threshold and the transformation tr_c is applied. Actor V_1 becomes again the most costly actor since its solution after reconfiguration is 2 and its cost becomes 84 per iteration. Therefore, $\mathcal{H}_G = 2 \cdot \frac{1}{2 \cdot 0.042} = 24fps$ tokens/second.

At iteration 200, the execution time of actors D_1 and D_2 increases to $120ms$. Again the throughput starts to decrease to $\mathcal{H}_G = 2 \cdot \frac{1}{0.12} = 16.6fps$. The transformation rule is applied for the second time and throughput returns to $\mathcal{H}_G = 3 \cdot \frac{1}{3 \cdot 0.042} = 24fps$ tokens/second.

The reconfiguration costs are small enough so that the two reconfigurations are hardly visible in the video. This case study shows that RDF can be used to design easily adaptive image processing that can maintain the throughput at a desired value even with dynamic resolution changes.

7 Related work

Many different dataflow MoCs have been proposed in the few last decades. More recently, several *parametric* dataflow MoCs have been presented as an interesting trade-off between expressiveness and analyzability. Among the existing parametric MoCs let us cite PSDF [19], VRDF [20], SPDF [21], BPDF [4], PiSDF [22], and PFSM-SADF [23]. They all offer a controlled form of dynamism under the form of parameters (*e.g.,* parametric rates) along with run-time parameter configuration.

Among those, BPDF [4] can model dynamic topology changes 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 [24].

An different approach is taken by SADF [6] and its parametric version PFSM-SADF [23]. They model 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 a (PFSM-)SADF model consists in applying the standard analyses of SDF to each scenario.

Both BPDF and SADF only allow a fixed, and usually small, number of graph topologies. Imagine a video application that may need to apply a collection of n filters depending on the

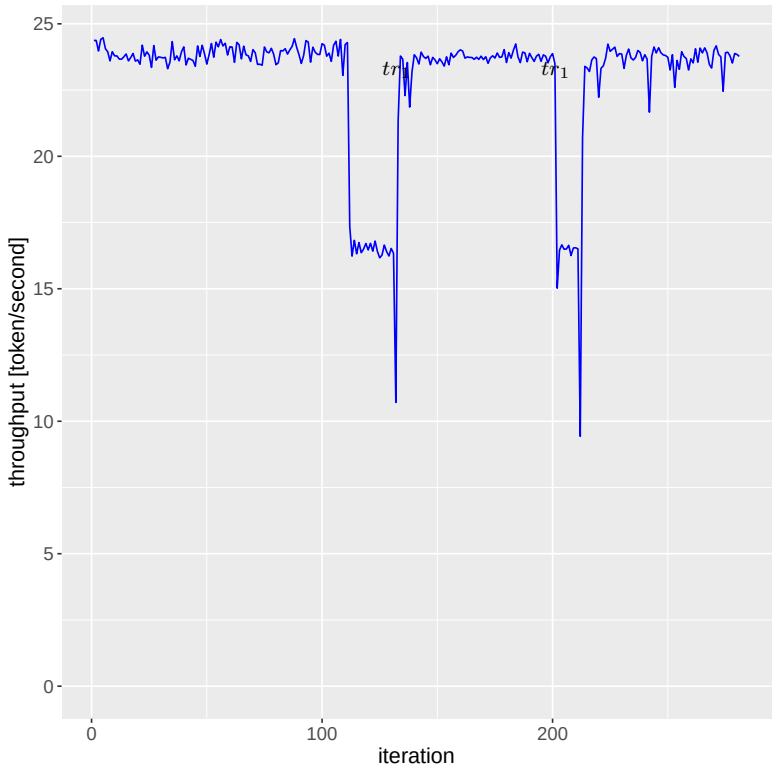


Figure 23: Throughput of the edge detection with transformation tr_c

characteristics of the video stream. Since there are 2^n combinations of such filters, describing so many graph configurations would be cumbersome in such MoCs.

To the best of our knowledge, RDF is the only dataflow MoC allowing both the dynamic reconfiguration (in the general sense) of the graph topology and static analyses for boundedness and liveness. It permits to generate an unbounded number of consistent and live graphs which do not have to be planned nor declared in advance.

Reconfigurability using rewriting rules has also been studied for Petri nets (see [25] for an overview). In the general case, reconfigurable Petri nets do not preserve properties such as liveness, boundedness, or reversibility. In [26], 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 [27]. This approach allows the absence of deadlocks to be verified.

8 Conclusion

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 reconfiguration programs triggered by conditions). The transformation rules determine *how* the RDF graph is reconfigured and the conditions specify *when* these reconfigurations take place. A key feature and advantage of RDF is that it retains static analyses to

guarantee boundedness and liveness properties of all possible graphs produced by dynamic reconfigurations. Finally, we outlined the main characteristics of our RDF prototype implementation and presented some experiments.

Several extensions of RDF come to mind. First, RDF rates could be generalized to accept parameters. Such rates are different from the parametric rates of variable arity actors which denote their number of edges. In parametric MoCs, a rate parameter may be changed, usually between iterations, to an arbitrary value. We expect this generalization to be relatively straightforward. Indeed, in such models, static analyses become parametric but remain similar to those of SDF. Second, it is likely that some constraints we enforced to guarantee boundedness and liveness could be relaxed. A clear candidate is the condition prohibiting transformations to manipulate edges with initial tokens. Finally, the pattern and transformation language might be enhanced. Consider the problem of duplicating a line of actors between a split and a join whereas this line may change overtime (by adding/removing actors). This would require to have as many transformation rules as there are versions of the line. Allowing to match and duplicate (unambiguously) some part of the graph without enumerating all its actors would come handy.

References

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

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

[11] A. Shafiei, "RDF : A reconfigurable dataflow model of computation," Ph.D. dissertation, Université Grenoble Alpes, Dec. 2021.

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

[13] 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.

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

[15] 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

[16] A. Bouakaz, P. Fradet, and A. Girault, "Symbolic buffer sizing for throughput-optimal scheduling of dataflow graphs," in *Real-Time and Embedded Technology and Applications Symposium, RTAS'16*, Vienna, Austria, Apr. 2016, pp. 199–208.

[17] ——, "Symbolic analyses of dataflow graphs," *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 2, p. 39, 2017.

[18] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[19] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for dsp systems," *Trans. Sig. Proc.*, vol. 49, no. 10, pp. 2408–2421, 2001.

[20] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit, "Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication," in *Proc. of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, 2008, pp. 183–194.

[21] P. Fradet, A. Girault, and P. Poplavko, "Spdf: A schedulable parametric data-flow moc," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2012, pp. 769–774.

[22] K. Desnos, M. Pelcat, J.-F. Nezan, S. S. Bhattacharyya, and S. Aridhi, "PiMM: Parameterized and Interfaced dataflow Meta-Model for MPSoCs runtime reconfiguration," in *13th Int. Conf. on Embedded Computer Systems: Architecture, Modeling and Simulation*, 2013, pp. 41 – 48.

[23] M. Skelin, M. Geilen, F. Catthoor, and S. Hendseth, "Parametrized dataflow scenarios," in *Proceedings of the 12th International Conference on Embedded Software*, 2015, pp. 95–104.

[24] 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.

[25] 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.

[26] 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.

[27] 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 first recall the following facts and notations:

- A graph is seen as a set of edges and a transformation rule as a set rewriting. 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 $tr(G) = X \cup \sigma(rhs)$.
- In any transformation $tr : lhs \Rrightarrow rhs$, both lhs and rhs are non empty.
- The set of edges of G is partitioned into X and lhs, meaning that $G = X \cup lhs$ and $X \cap lhs = \emptyset$. Note that X and lhs may have some actors in common because any node connected to one or more edges of X and to one of more edges of lhs appears both in X and in lhs.
- We write $x \underset{A}{\longrightarrow} y$ for a directed edge from x to y belonging to graph A (set of edges) and use the corresponding transitive closure $x \overset{+}{\underset{A}{\longrightarrow}} y$ (resp. reflexive transitive closure $x \overset{*}{\underset{A}{\longrightarrow}} y$) to denote paths in A. We write $x \underset{A}{\longleftrightarrow} y$ 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 $x \overset{+}{\underset{A}{\longleftrightarrow}} y$ (resp. reflexive transitive closure $x \overset{*}{\underset{A}{\longleftrightarrow}} y$) 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 weakly connected graph and $tr : lhs \Rrightarrow rhs$ be a transformation rule such that*

$$\forall x \neq y \in rhs, x \overset{+}{\underset{rhs}{\longleftrightarrow}} y \qquad (\mathbf{C}^{conn})$$

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

Proof. Let x and y be two distinct actors in $tr(G)$; we must prove that $x \overset{+}{\underset{tr(G)}{\longleftrightarrow}} y$. As said above, we consider tr as the set rewriting $G = X \cup \sigma(lhs) \Rrightarrow X \cup \sigma(rhs) = tr(G)$. Note that Cond. $(\mathbf{C}^{conn})$ implies that forall x, y in $\sigma(rhs)$, we have $x \overset{+}{\underset{\sigma(rhs)}{\longleftrightarrow}} y$.

We distinguish the following exclusive cases: **(A)** x and y are in $\sigma(rhs)$; **(B)** neither x nor y are 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 $y \in \sigma(rhs)$.

We have already stated $x \overset{+}{\underset{\sigma(rhs)}{\longleftrightarrow}} y$ for any two distinct actors x and y of rhs. Since $\sigma(rhs) \subseteq tr(G)$, we therefore conclude that $x \overset{+}{\underset{tr(G)}{\longleftrightarrow}} y$.

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

Recall that an actor belonging to lhs but not to rhs is removed from the graph. Since neither x nor y are removed by tr (by assumption they are in $tr(G)$), necessarily none of them belong to $\sigma(lhs)$. It follows that they both belong to X, and therefore to G. Since G is weakly connected, we thus have $x \overset{+}{\underset{G}{\longleftrightarrow}} y$.

Since both x and y belong to X, this undirected path between x and y in G must start and finish with an edge in X, meaning that it consists of an alternation of subpaths in X and

subpaths in $\sigma(lhs)$, starting and ending with a subpath in X. Formally, this path must have one of the two following forms:

- $x \xleftrightarrow[X]{+} x_1 \xleftrightarrow[\sigma(lhs)]{+} x_2 \xleftrightarrow[X]{+} x_3 \xleftrightarrow[\sigma(lhs)]{+} \cdots \xleftrightarrow[\sigma(lhs)]{+} x_n \xleftrightarrow[X]{+} y$ with $n \geq 1$;
- or $x \xleftrightarrow[X]{+} y$.

The second case trivially implies $x \xleftrightarrow[tr(G)]{+} y$ since $X \subseteq tr(G)$. We therefore focus on the first case. For each $1 \leq i \leq n$, x_i belongs to two edges, one in X and one in $\sigma(lhs)$, hence x_i belongs to X and cannot be suppressed by tr (thanks to Cond. (**C2**)). It follows that, for each $1 \leq i \leq n$, x_i belongs to $\sigma(rhs)$.

Now, by Cond. ($\mathbf{C}^{conn}$), $\sigma(rhs)$ is weakly connected, hence we have $x_1 \xleftrightarrow[\sigma(rhs)]{+} x_n$. Furthermore, edges in X being untouched by tr, we thus have $x \xleftrightarrow[X]{+} x_1 \xleftrightarrow[\sigma(rhs)]{+} x_n \xleftrightarrow[X]{+} y$. Since $GX \cup \sigma(rhs)$, 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, x does not necessarily belong to $\sigma(lhs)$. We consider both cases in turn.

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

Since y belongs to the weakly connected graph G, we have $x \xleftrightarrow[G]{+} y$. Similarly to Case **(B)**, this path consists of an alternation of subpaths in X and subpaths in $\sigma(lhs)$, starting with a subpath in $\sigma(lhs)$ and ending with a subpath in X:

- $x \xleftrightarrow[\sigma(lhs)]{+} x_1 \xleftrightarrow[X]{+} x_2 \xleftrightarrow[\sigma(lhs)]{+} \cdots \xleftrightarrow[\sigma(lhs)]{+} x_n \xleftrightarrow[X]{+} y$ with $n \geq 1$

On the one hand, since x_n belongs to X and to $\sigma(lhs)$, it also belongs to $\sigma(rhs)$ (same reasonning as in Case **(B)**). Furthermore, by hypothesis x also belongs to $\sigma(rhs)$. Therefore, by Cond. ($\mathbf{C}^{conn}$), $x \xleftrightarrow[\sigma(rhs)]{+} x_n$, hence $x \xleftrightarrow[tr(G)]{+} x_n$.

On the other hand, edges in X such as $x_n \xleftrightarrow[X]{+} y$ being untouched by tr, we have $x_n \xleftrightarrow[tr(G)]{+} y$. Putting both facts together, we conclude that $x \xleftrightarrow[tr(G)]{+} y$.

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

In that case, x is a fresh actor created by tr.

By definition, $lhs \neq \emptyset$ (see Section 3.2.1), so let $z \in lhs$. $G = X \cup \sigma(lhs)$ begin weakly connected, we have $z \xleftrightarrow[G]{+} y$. This path is an alternation of subpaths in X and subpaths in $\sigma(lhs)$, starting with a subpath in $\sigma(lhs)$ and ending with a subpath in X:

- $z \xleftrightarrow[\sigma(lhs)]{+} x_1 \xleftrightarrow[X]{+} x_2 \xleftrightarrow[\sigma(lhs)]{+} \cdots \xleftrightarrow[\sigma(lhs)]{+} x_n \xleftrightarrow[X]{+} y$ with $n \geq 1$

Necessarily x_n is not touched by tr (because it belongs to X and lhs in G), hence x_n belongs both to X and to rhs in $tr(G)$. This implies two things. On the one hand we have $x_n \xleftrightarrow[X]{+} y$, implying $x_n \xleftrightarrow[tr(G)]{+} y$. On the other hand, rhs being connected, thanks to Cond. ($\mathbf{C}^{conn}$), we thus have $x \xleftrightarrow[\sigma(rhs)]{+} x_n$, implying $x \xleftrightarrow[tr(G)]{+} x_n$.

By transitivity, we therefore conclude that $x \overset{+}{\underset{tr(G)}{\longleftrightarrow}} y$. □

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*

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

then $tr(G)$ is consistent.

Proof. First, consider a graph G (a set of edges) that can be partitioned into two disjoint subsets of edges (two subgraphs) G_1 and G_2, *i.e.,* $G = G_1 \cup G_2$ 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:

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

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 graph partitioned into G_1 and G_2. We have:*

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

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

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

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 $tr(G)$:

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

The proof holds for variable arity actors. The condition and reasoning deal with symbolic solutions which can accommodate parameters of X actors.

□

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

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

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

Proof. It is well known that any consistent acyclic SDF graph has a single appearance schedule [12]. We therefore focus on cycles and first prove the following lemma which states 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 $\sigma(rhs)$:

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

   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:

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

   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 [12]) 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 can 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