RDF: Reconfigurable Dataflow

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

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

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

***Index Terms*—Models of computation; Synchronous dataflow; Reconfigurable systems; Static analyses; Boundedness; Liveness.**

I. 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 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), an SDF actor consumes a fixed number of data (tokens) on each of its input edges, performs some computation and produces a fixed number of tokens on each of its output edges. These numbers of consumed and produced tokens are *static*, which allows static analyses to check boundedness and liveness of SDF graphs.

Being able to check statically the boundedness and the liveness is a strong advantage, but it comes at the price of forbidding any dynamic changes of the SDF graph. For this reason, several extensions of SDF have been explored such as the parametric production and consumption rates (*e.g.,* PSDF [3], BPDF [4], PiSDF [5]), or allowing limited changes of the topology using scenarios (*e.g.,* SADF [6]). 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 increases), embedded computer vision (to change the frame decomposition), automatic control (to change the control law depending on stability criteria).

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

The paper is organized as follows. We start by recalling the basic notions of SDF in section II. RDF is introduced in section III. Section IV presents the static analyses ensuring that RDF reconfigurations preserve the connectivity, consistency, and liveness properties. We outline in section V the main features of the implementation of RDF. Finally, section VI presents related work and section VII concludes.

Proofs of the theorems stated in this paper as well as some additional material can be found in a companion research report [10].

II. Synchronous Dataflow

An SDF graph [2] is a directed graph, where vertices – called *actors* – are functional units. Actors are connected by *edges*, which are FIFO channels. The atomic execution of a given actor – called actor *firing* – consumes data tokens from all its incoming edges (its *inputs*) and produces data tokens to all its outgoing edges (its *outputs*). The number of tokens consumed (resp. produced) on a given edge at each firing is called the *consumption* (resp. *production*) *rate*. An actor can fire only when *all* its input edges contain enough tokens (*i.e.,* at least the number specified by the consumption rate of the corresponding edge). In SDF, all rates are constant integers known at compile time.

† : Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France. first.last@inria.fr

‡ : Orange, France. first.last@orange.com

∗ : Corresponding author

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

- V is a finite set of actors; among those, we distinguish *source* actors that have no incoming edges, and *sink* actors that have no outgoing edges;
- E is a finite set of directed edges ($E \subseteq V \times V$);
- $\rho : E \to \mathbb{N}\backslash\{0\} \times \mathbb{N}\backslash\{0\}$ is a function that returns for each edge a pair (x, y), where x is the production rate of its origin actor (producer) and y is the consumption rate of its destination actor (consumer);
- $\iota : E \to \mathbb{N}$ is a function that returns for each edge the number of its initial tokens (possibly 0).

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

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

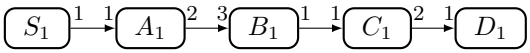


Fig. 1: The SDF graph G_1.

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

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

An SDF graph is said to be *consistent* if it admits a repetition vector. The repetition vector is obtained by solving the following *system of balance equations*: each edge $X \xrightarrow{p \quad q} Y$ is associated with the balance equation $sol(X).p = sol(Y).q$, which states that all produced tokens during an iteration must be consumed within the same iteration. The graph is consistent if and only if this system of equations admits a non-null solution [2] (an easy check). An important consequence is that a consistent graph can be executed infinitely with *bounded* memory: all produced tokens are eventually consumed.

The next step is to determine a static order, a *schedule*, in which the firings of the repetition vector can be executed. It is obtained by an abstract computation where an actor is fired only when it has enough input tokens. Such a schedule ensures that the graph returns to its initial state and that each actor is eventually fired. An consistent SDF graph is said to be *live* if it admits a schedule [2].

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

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

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

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

Fig. 2: Periodic execution loop for actor X.

The `consume_input_tokens` instruction blocks when (at least) one of the input buffers of X does not contain enough tokens, while the `produce_output_tokens` instruction blocks when (at least) one of the output buffers of X is full. On each core, one such thread `th_X` is started for each actor X allocated to it. This multi-threaded ASAP execution guarantees that the graph can be executed in bounded memory and without deadlock, provided that each buffer has at least the minimal size required for liveness [11].

III. RDF: A Reconfigurable Dataflow MoC

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

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

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

A. RDF graph

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

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

B. RDF Controller

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

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

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

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

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

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

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

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

C. Transformation rules

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

$$tr : lhs \Rrightarrow rhs$$

which selects a sub-graph matching lhs, and replaces it by the graph specified by rhs. We use the set-theoretic approach of [12] to graph rewriting: the terms lhs and rhs are seen as sets of edges possibly with pattern *variables* matching either types, actor indices, or rates.

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

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

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

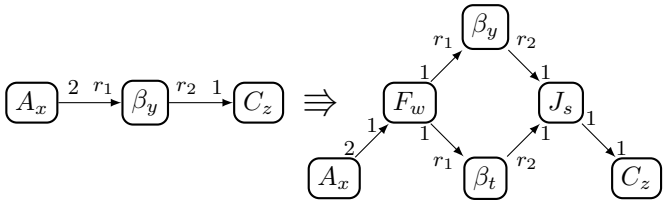


Fig. 3: The transformation rule tr_1.

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

$$A_1 \xrightarrow{2} {}^{3} B_1 \xrightarrow{1} {}^{1} C_1$$

and yields the substitution

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

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

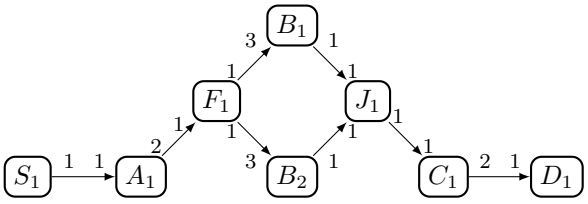


Fig. 4: The resulting graph G_2 after applying tr_1 to G_1.

The following facts should be pointed out:

- An actor occurring in the lhs but not in the rhs is *suppressed*. However, to be valid, all incoming and outgoing edges of that actor should occur in the lhs. Otherwise, suppressing an actor would create dangling edges. To verify this point, we request the type of removed actors to appear explicitly in the rule. Indeed, when the type is known, the numbers of incoming and outgoing edges are also known and the rule can be checked statically. In the rule tr_1, no actor is suppressed since all matched actors occur in the rhs.
- When an actor index variable occurs in the rhs but not in the lhs, then it yields a *new* actor (instance of the given type) that must therefore be *created*. In contrast, type variables occurring in the rhs must always occur (*i.e.,* be defined) in the lhs. Indeed, it would be ambiguous to create new instances of unknown types. In the transformation rule tr_1, the terms F_w, β_t and J_s illustrate this case: w, t and s yield new actors, whose types are known because they are either explicit (F, J), or defined in the lhs (β).
- Rates and number of incoming and outgoing edges must be consistent with types. This property is easy to check. For instance, no other rate than 2 could decorate the outgoing edge of A_x in tr_1. Rate variables are often superfluous since they are fixed by the type of the actor they are attached to. In such cases, they can be omitted.

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

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

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

Initial tokens raise semantic issues. For instance, if a transformation has a rhs with initial tokens, we would need a way to specify the origin or values of these tokens. To keep things simple, we allow the initial RDF graph to have tokens but impose that transformations do not match nor return edges with initial tokens.

IV. RDF STATIC ANALYSES

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

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

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

A. Connectivity

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

Theorem 1 states that, in order to guarantee that connectivity is preserved by the transformation rule $tr : lhs \Rrightarrow rhs$, it is sufficient to ensure that rhs is a connected graph.

Theorem 1. *Let G be a connected graph and $tr : lhs \Rrightarrow rhs$ be a transformation rule such that $\forall x, y \in rhs, x \overset{*}{\underset{rhs}{\longleftrightarrow}} y$. Then $tr(G)$ is a connected graph.*

The proof of theorem 1, as well as proofs of theorems 2 and 3, can be found in [10].

Clearly, the transformation tr_1 preserves connectivity, but the following one

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

B. Consistency

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

For each transformation rule $tr : lhs \Rrightarrow rhs$, we check that both 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 the same solution. New actors (*i.e.,* occurring only in rhs) only need to have a solution.

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

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

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

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

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

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

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

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

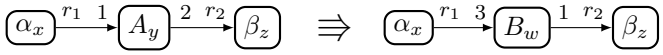
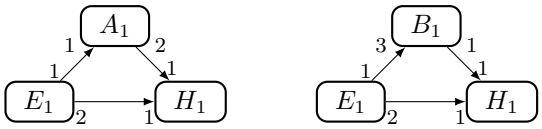


Fig. 5: The transformation rule tr_2.

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



(a) (b)

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

C. Liveness

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

For each transformation rule $tr : lhs \Rrightarrow rhs$, Theorem 3 checks that, for each SAS of lhs, there is an SAS of rhs such that the actors common to lhs and rhs appear in the same order (see $\prec$ in Section II) in both schedules.

Theorem 3. *Let G be a live graph with an SAS and $tr : lhs \Rrightarrow rhs$ be a transformation rule. $tr(G)$ is live and admits an SAS if, for each SAS S of lhs, there is an SAS S' of rhs such that*

$$\forall x, y \in lhs \cap rhs, x \prec_S y \Rightarrow x \prec_{S'} y.$$

The transformation rule tr_1 of Fig. 3 preserves liveness. The lhs has a single SAS $A_x^a; \beta_y^b; C_z^c$ and the rhs has the SAS $A_x^a; F_w^f; \beta_y^b; \beta_t^b; J_s^j; C_z^c$. The actors x, y and z appear in the same order in both schedules so the liveness condition is satisfied.

On the other hand, the transformation tr_3 in Fig. 7 is invalid. The lhs has two SAS $X_x^1; Y_y^1; Z_z^1; T_t^1$ and $X_x^1; Z_z^1; Y_y^1; T_t^1$ and the rhs has a single SAS $X_x^1; Y_y^1; Z_z^1; T_t^1$. The first SAS of the lhs has no corresponding SAS in the rhs. Therefore, if the only SAS in the initial graph was one were Z_z needed to be fired before Y_y, then rule tr_3 would produce a deadlocked (*i.e.,* non live) graph.

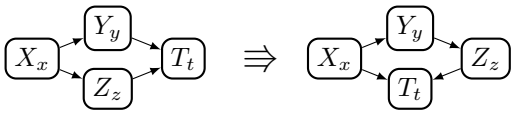
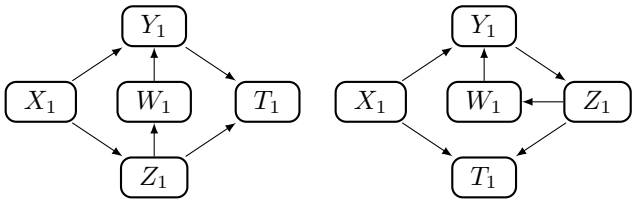


Fig. 7: The transformation rule tr_3 (all rates are 1).

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



(a) (b)

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

V. Implementation

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

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

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

The controller (which runs inside its own thread) continuously watches whether one of its reconfiguration condition is satisfied (see Section III-B). Whenever this occurs, before

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

VI. Related work

To the best of our knowledge, no existing dataflow MoC allows both the dynamic reconfiguration (in the general sense) of the graph topology and static analyses for boundedness and liveness. Still, several dataflow MoCs allow a limited form of topology changes, including SADF [6] and BPDF [4], while still remaining statically analyzable.

SADF [6] models reconfigurability as a set of predefined configurations (called scenarios), coupled with a nondeterministic finite-state machine that specifies the transitions between scenarios. The number of available topologies is statically fixed and specified in the source model. Analyzing an SADF model consists in applying the standard analyses of SDF to each scenario.

BPDF [4] models reconfigurability by adding Boolean conditions to FIFO channels. When a condition switches to false (resp. true) the channel is *disabled* (resp. *enabled*). Boundedness and liveness remain statically analyzable, and static or quasi-static schedules can be produced [13].

Reconfigurability using rewriting rules has also been studied for Petri nets (see [14] for a recent overview). In the general case, reconfigurable Petri nets do not preserve properties such as liveness, boundedness, or reversibility. In [15], 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 [16]. This approach allows the absence of deadlocks to be verified.

VII. Conclusion

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

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

References

[1] 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 MP-SoCs 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*, 2014.

[10] P. Fradet, A. Girault, R. Krishnaswamy, X. Nicollin, and A. Shafiei, "Rdf: Reconfigurable dataflow (extended version)," Inria - Grenoble - Rhône-Alpes, Research Report 9227, Nov. 2018.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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