RDF : A Reconfigurable Dataflow Model of Computation

Arash Shafiei

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Arash SHAFIEI

Thèse dirigée par
Pascal FRADET, Université Grenoble Alpes
et codirigée par
Alain GIRAULT, Université Grenoble Alpes
et co-encadrée par
Xavier NICOLLIN, Grenoble INP
préparée au sein du
Laboratoire d'Informatique de Grenoble
dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

RDF : Un Modèle de Calcul Flot de Données Reconfigurable

RDF : A Reconfigurable Dataflow Model of Computation

Thèse soutenue publiquement le **16 décembre 2021**,
devant le jury composé de :

Monsieur Pascal FRADET
Chargé de recherche HDR, Inria Grenoble Rhône-Alpes,
Directeur de thèse
Monsieur Alain GIRAULT
Directeur de recherche, Inria Grenoble Rhône-Alpes,
Co-directeur de thèse
Monsieur Jean-François NEZAN
Professeur des Universités, INSA Rennes, Rapporteur
Monsieur Tanguy RISSET
Professeur des Universités, INSA Lyon, Rapporteur et Président du jury
Monsieur Hamid-Reza ZARANDI
Professeur associé, Amirkabir University of Technology, Examinateur
Madame Ruby KRISHNASWAMY
Ingénieur docteur, Orange Labs, Examinatrice

Contents

Chapter 1

Introduction

1.1 Models of Computation

A Model of Computation (MoC) is a model describing how a set of arithmetic and logical operations are performed in order to produce an output given an input. Three main categories of MoCs are distinguished: sequential, functional, and concurrent [45].

The sequential MoCs are those described in automata theory [31]. In automata theory, a computational problem is modelled as a formal language. A formal language is a set of strings where each string is a sequence of characters of an alphabet. For example, the computational problem of verifying whether a graph is connected can be expressed using a language containing the string format of all connected graphs. For a given language, an automaton is designed in order to determine if a string belongs to that language, or in other word to accept an instance of the computational problem. For example, the language corresponding to the connected graphs problem is a set of the string format of all connected graphs and an automaton determines if a string belongs to this language, or in other words if a graph is connected.

Chomsky has proposed a classification of formal languages according to the formal grammar used to recognize them: regular languages (type 3), context-free languages (type 2), context-sensitive languages (type 1) and recursively enumerable languages (type 0) [17]. The model of computation to express the regular languages is Finite State Machine (FSM). This model consists of a number of control states and transitions between them. By observing a character in a string, the machine decides how to change the state. The amount of memory that this machine needs is fixed and very limited since the machine only remembers its current state. In order to recognize context-free languages, extra information about the context is needed. Push-Down Automata (PDA) adds a stack to express this class of languages. For recognizing context-sensitive and recursively enumerable languages more powerful memory models are needed. The Linear Bounded Automata (LBA)

is used for recognizing context-sensitive languages by adding a tape (array of memory cells) with a head that can read, write, move forward and backward. The length of its tape is bounded in the length of the input. A set of rules determines how the head moves. Finally, the Turing machine is used to recognize recursively enumerable languages. It is similar to an LBA except that the length of its tape is unbounded. Turing machines are capable of simulating any computer algorithm.

Sequential MoCs not only allow recognizing a language, but can also produce an output for a given input string. Mealy and Moore machines are examples of MoCs which produce output. Therefore, in the context of automata theory, an MoC is an automaton describing how a language is recognized and (optionally) how an output string is computed given an input string.

In the functional MoCs such as lambda calculus [18], computations are expressed based on functions. In order to solve a computational problem, a set of functions is defined and applied to an input. By binding and substituting variables, the corresponding output is produced. Lambda calculus is Turing complete, that is, it is capable of simulating any Turing machine.

In the concurrent MoCs, the computation is performed by a set of components that interact with each other using communication channels. In [35], an MoC is defined as a set of laws governing the interaction of components. Each component computes an output given an input using a set of mathematical and logical operations. The set of all components form a graph of computation in which the vertices of the graph represent the computation units and the edges of the graph represent the communication channels. Concurrent MoCs are suitable for modelling applications in the domain of signal, image, and data processing, because signals are streams of data transferred over communications channels and processed by a set of computation units.

This thesis focuses on a well-known concurrent MoC proposed by E. Lee and D. Messerschmitt : Synchronous Dataflow (SDF) [37]. It is deterministic in the sense that for a stream of inputs, it always produces the same stream of outputs regardless of the order of execution of its components. Besides determinism, two important properties of SDF are boundedness and liveness. Boundedness ensures that an SDF graph can be executed in bounded memory while liveness ensures that it always executes without deadlock. Other standard concurrent MoCs are Petri Nets [42, 43] and Process Networks [33].

A taxonomy of MoCs is shown in Figure 1.1 [45].

1.2 Motivation

Concurrent MoCs are suitable for modelling signal processing systems, such as multimedia processing system. In the following, we take a multimedia

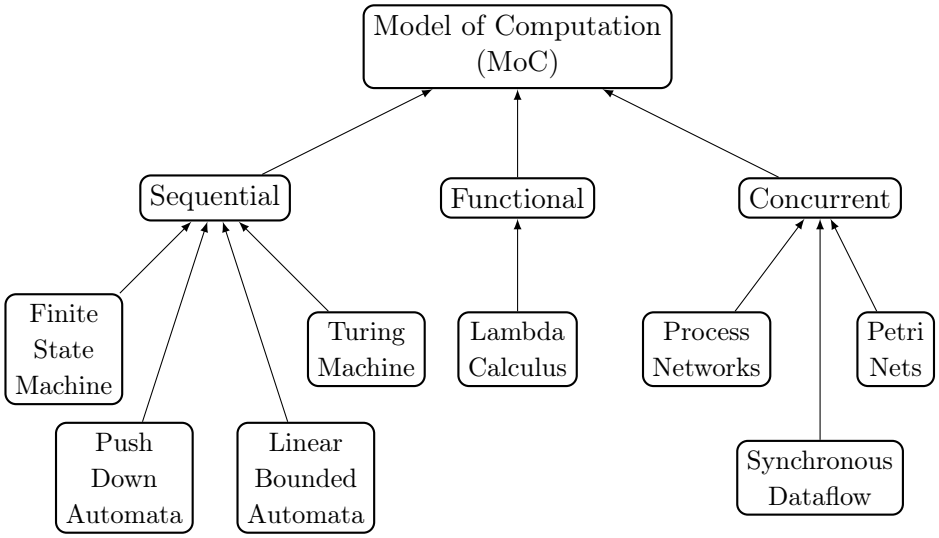


Figure 1.1: Taxonomy of Models of Computation [45]

processing example, characterize its main features, discuss about the existing MoCs for modelling this type of application, and finally identify the shortcoming of the existing MoCs. Here will reside the main motivation of this work.

A multimedia application usually consists of a number of predefined functions also called filters connected to each other in the form of a graph. Each filter receives data (which can be audio, image, video, or text) from its inputs, processes the data, and sends the data it has produced to the next filters. Some of these filters are independent and, on a multi-processor architecture, they can be executed in parallel.

Let us consider a simple application in which (1) a multimedia stream containing video and audio is received from a source (which can be a video file, or a camera), (2) this multimedia stream is decoded, images are sent to a filter and audio samples to another filter, (3) the image processing filter detects and extracts edges, (4) the audio processing filter reduces noises in the audio samples, (5) images containing only edges and audio samples with reduced noise are encoded into a new video stream by an encoder, (6) and finally the multimedia stream is sent to a sink (which can be a video file, a network channel, or a display). This application can be modelled as the computation graph shown in Figure 1.2.

Such kinds of applications can be modelled using concurrent MoCs such as SDF which models an application as a graph of filters. Moreover, on a multi-processor architecture, different bindings between filters and processors yield different performances. Since SDF provides scheduling facilities to optimize the performance metrics of graphs, using such MoC yields better performance. Two of the most common performance metrics for dataflow

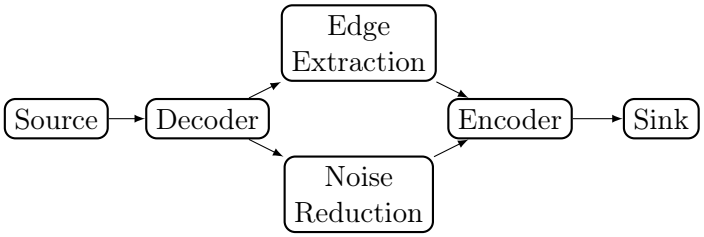


Figure 1.2: A simple multimedia application

applications are latency and throughput. In the previous example, binding edge extraction and noise reduction functions to different processors results in a higher throughput.

Furthermore, suppose that in our multimedia application example we have enough processors, that the quality of the input video increases and that edge extraction becomes a more time consuming function. In this case, we can add new edge extraction filters in order to process consecutive images in parallel to maintain the throughput. To do so, the configuration of the graph must change when the video quality changes, that is, the graph must be reconfigured while executing the application. Another kind of reconfiguration might be needed if the luminosity of the images lowers and a night filter needs to be added to the graph. The reconfiguration might also be needed when a filter is to be replaced by another one. For example, a different noise reduction filter might be needed because the noise in the input signal surpasses a threshold. For all these scenarios, the graph needs to be reconfigured.

The SDF MoC models an application using a *static* graph, meaning that the configuration of the graph cannot be changed dynamically. Extensions of SDF exist in which an application is modelled by a bounded number of graphs in different configurations. One of those MoCs, Scenario-Aware Dataflow (SADF) [25, 52], models an application using several SDF graphs and a finite state machine. By changing the state of the machine, the executing datafow graph can be changed. For the use cases we discussed above, a small number of configurations is enough and SADF can model such applications.

However, if in the previous example, the number of different video qualities is unknown, then adding many edge detection filters may be needed and the number of different configuration becomes unknown. Reconfiguration requirements have been studied in different extensions of SDF MoC, but none of them addresses the problem of modelling an *unbounded* number of configurations. In this thesis, the problem of reconfiguration of SDF MoC is studied and a new MoC called Reconfigurable Dataflow (RDF) is proposed to solve this problem.

1.3 Contribution

RDF is an MoC that extends SDF with *transformation rules*. Each transformation rule determines how the dataflow graph currently executing can be modified to another graph. A simple rule, such as a rule adding a new filter between two filters, can produce an unbounded number of configurations. The RDF MoC is designed to allow *dynamic reconfigurations* of SDF graphs while preserving the boundedness and liveness properties of the graphs.

The RDF MoC is an extension of SDF. An RDF graph is similar to an SDF graph except that each actor has a *type*. The types allow adding an unbounded number of actors as instances of a finite set of actor types. To allow the number of input or output edges of an actor to change dynamically, a generic actor type with *variable arity* is proposed. An RDF application also contains a *controller* to specify how and when a graph may be reconfigured. The controller specifies a number of transformation rules. Each rule consists of a left-hand side and a right-hand side. The left-hand side is a pattern that should match a subgraph in the current RDF graph. The subgraph which is matched is substituted by the subgraph specified by the right-hand side of the transformation rule. The rules are applied depending on conditions involving various criteria (throughput, latency, buffer occupancy, ...). From an initial RDF graph and a few number of transformation rules, an unbounded number of different RDF graphs can be generated, and this is a key feature of the RDF MoC.

SDF graphs have two important properties: They can be shown to be consistent and live. For the RDF MoC, we have to ensure that these properties are preserved after reconfigurations. *Static analyses* are proposed to ensure that a transformation rule is valid and that it preserves these properties.

Regarding the performance analysis of RDF, we have focused on *latency* and *throughput*. We have considered a widely used class of transformation rules and computed bounds for the impact of a rule on latency and throughput of the graph.

An implementation of RDF is proposed. When a transformation rule needs to be applied, the dataflow graph must be paused, the subgraph matched to the left-hand side must be substituted by the right-hand side of the rule, and the execution must be resumed. All data structures and algorithms to implement these mechanisms are elaborated. The most common conditions that may be used in practice to trigger transformation rules are presented. Graph matching is in general a costly operation. A practical approach to make the pattern matching efficient is proposed. Experimentations are performed in order (1) to show that transformation rules can be used to change the parallelization levels of a graph and therefore improve the performance of the graph, and (2) to show that the runtime cost of applying a reconfiguration is small and that the implementation can be used

for practical applications.

Finally, a small case study is implemented to illustrate what can be done using RDF. It is based on the application presented above that extracts edges from images. At runtime, the execution time of one of the actors is increased. It is first implemented using SDF and the throughput of the graph decreases when that actor becomes more time consuming. The same application is then implemented using RDF. This time, once the execution time of that actor increases, the loss of throughput is detected by a condition and a transformation rule is applied, which increases the parallelization level of the RDF graph. As a result, the throughput is maintained at the level it was before.

The structure of the document is as follows. In chapter 2, we review existing dataflow MoCs. We study SDF and some of its extensions. We discuss how those extensions address the problem of reconfiguration and what are their limitations. In chapter 3, we present the RDF MoC along with its static analyses. In chapter 4, we propose methods for analyzing the impact of transformation rules on latency and throughput of RDF graphs. In chapter 5, we describe our implementation of RDF in detail, conduct some experiments, and finally present a case study. We conclude in chapter 6 by summarizing our contributions and proposing some avenues for future research. An appendix provides further details on the RDF prototype.

Chapter 2

Dataflow Models of Computation

Lee et al. define a concurrent MoC as a set of laws governing the interaction of computation units, communication links, and memory [35]. These laws define how the the inputs can be computed concurrently given the inputs. This category of MoCs is called concurrent [45] which includes Process Networks, Petri Nets, Synchronous Dataflow, and their extensions. The concurrent models of computations (MoCs) describe the organization of communication links and computation units at a high level; they do not describe in details the whole application. In particular, the detailed behavior of computation units is not described by these MoCs.

There are different sub-classes of concurrent MoCs. A dataflow MoC is a concurrent MoC in which there is no notion of time and message. It models the computation using a directed graph in which vertices are the computation units, the edges are the communication links, and data flows on these edges. Each dataflow MoC defines its own laws for consuming, processing, and producing data. A dataflow MoC is usually characterized by its analyzability, expressiveness, and reconfigurability.

The analyzability concerns the complexity of analyzing applications specified in the MoC. For example, how complex it is to determine whether or not an application deadlocks (which is called *liveness analysis*), or whether it runs in bounded memory (which is called *boundedness*). Other types of analysis include performance analyses such as latency or throughput analysis.

The expressiveness concerns the ease of expression and compactness of applications of the MoC. Usually the more an MoC is expressive, the less it is analyzable. For example, a dataflow MoC may provide special control operators to modify the flow of data through the graph. These control operators make the MoC more expressive because a broader range of applications can be described by the MoC. However, analyzing such an MoC is more involved

and complex. One of the MoCs providing such control operators is BDF MoC presented in section 2.4.2. The control operators of BDF make is more expressive, but they also make boundedness and liveness undecidable [36].

Finally the reconfigurability concerns the ability of the MoC to reconfigure the applications at runtime. For example, if an MoC allows changing some parameters of an application at run-time, then the MoC is more reconfigurable. One example of such parameters is the number of data a computation unit produces at each execution. The focus of this thesis is on reconfigurable dataflow MoCs.

In sections 2.1 and 2.2, we discuss two early dataflow MoCs : Petri Nets (PNs) and Kahn Process Networks (KPNs). In sections 2.3 and 2.4, we present Synchronous Dataflow (SDF), some of its extensions and finally we summarize.

2.1 Petri Net

The Petri net model [42, 43] is an MoC describing concurrent processes. It models computations as a directed bipartite graph with two types of vertices: *places* and *transitions*. Transitions represent computation units and places represent memory. Arcs specify which places and transitions are connected. Since each arc connects a place to a transition, no two vertices of the same type can be connected to each other and hence the graph is bipartite. The places from which an arc is connected to a given transition are called the input places of this transition, while the places to which an arc is connected from a given transition are the output places of this transition.

Places may contain *tokens* that abstract away the available resources. At any given time, the state of a Petri net, called its *marking*, is the number of tokens contained in each of its places. Upon initialization, the state of a Petri net is called its *initial marking*.

Each arc is labeled with a positive integer called its *multiplicity*. Transitions may *fire*, to make the Petri net evolve. When a transition fires, (1) from each of its input places, it *consumes* as many tokens as the multiplicity of the associated arc, and (2) onto each of its output places, it *produces* as many tokens as the multiplicity of the associated arc. A transition can only fire when there are enough tokes in all of its input places, in which case it is said to be *enabled*. Petri net is a concurrent MoC because, from a given marking, several transitions may be enabled.

An example of a Petri net is shown in Figure 2.1. Transitions are represented by squares and places by circles. There are two tokens in the place p_1, which enables both transitions t_2 and t_3, so both can execute concurrently. Once they have been executed, each produces one token in the place p_2. The multiplicity of each arc is (implicitly) 1.

A marking is *reachable* if starting from the initial marking, a sequence of

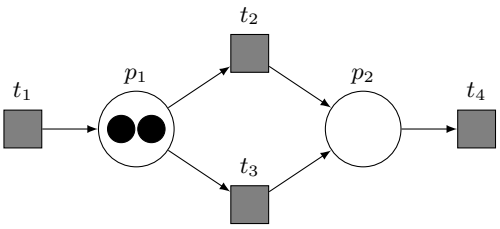


Figure 2.1: An example of a Petri net

execution of transitions results in that marking. Petri nets have a decidable reachability.

Boundedness and liveness of Petri nets are also decidable. A place in a Petri net is k-bounded if it does not contain more than k tokens in any reachable marking. A Petri net is called k-bounded if all its places are k-bounded. For liveness analysis, different degrees of liveness are defined; a transition can be either dead which means it never executes, or live at different levels of liveness depending on whether it sometimes, often, or always executes.

Petri nets are not deterministic, which means that the order in which transitions are fired (and tokens are produced) cannot be determined. Indeed, when any two transitions are enabled (regardless of whether they share an input place or not) the MoC does not specify in which order they are fired. For example, in Figure 2.1 we cannot determine in what order the two tokens produced by executions of t_1 are consumed by t_2 and t_3.

2.2 Process Network

Kahn Process Network (KPN) [33] is an MoC to express concurrent processes. KPN models the computation as a directed graph made of processes connected with channels of communication and running in parallel. Processes are functions transforming input streams of data into output streams. The channels are assumed to be unbounded first-in first-out (FIFO) queues. The functions are continuous, preventing the processes to send outputs after it has received an infinite amount of input, and they are monotone, meaning that the partial order of an input stream is preserved once it is processed. An example of a KPN graph with four processes p_1, p_2, p_3, and p_4 and four channels c_1, c_2, c_3, and c_4 is shown in Figure 2.2.

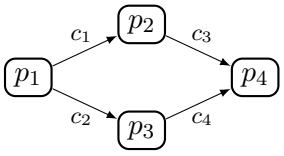


Figure 2.2: An example of a KPN graph

A process cannot test for emptiness of its input channels; whenever a process attempts to read from an empty input channel, it blocks and waits until some data has been produced and sent to that buffer by its predecessor process. Once the data is read from the input channels, the process computes the outputs and writes them on the output channels. Writing on output channels is never blocking. Therefore, a process can be in two states; either it is active, computing and writing, or it is waiting because it is blocked on some of its empty input channel. If we see the processes as functions and the communication channels as variables, then a process network can be seen as a system of equations on streams.

A KPN can be shown to be deterministic, which means that for a given sequence of inputs, the sequence of outputs is unique no matter how the processes are executed; their execution order does not impact the sequences produced on channels. A KPN could become non-deterministic if it was allowed to check for emptiness, to write to more than one channel, or if more than one process could read from the same channel or by sharing variables. If a process were allowed to check for emptiness, it could perform two different actions depending on the time the input arrives. This would imply that two different sequences of input data would produce two different sequences of outputs because of the variations in arrival of data. Similarly, if more than one process were allowed to read from one input channel, depending on the schedule of the processes in two different executions, different inputs would be read by the processes and as a result different outputs would be produced.

2.3 Synchronous Dataflow

The Synchronous Dataflow (SDF) MoC [38] models computation with a graph of computations units and communication channels. An SDF graph is a directed graph, where vertices, which are called *actors*, represent computation units. Actors are connected by *edges*, which represent communication channels. A channel is a first-in first-out (FIFO) buffer. Each atomic data written to or read from these buffers is called a *token*. A channel can contain a number of tokens before starting the execution. These tokens are called *initial tokens*.

The atomic execution of a given actor, which is also 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, that is, at least the number specified by the consumption rate of the corresponding edge. In SDF, all rates are constant integers known at compile time.

2.3.1 Definition

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 connecting two actors ($E \subseteq V \times V$);
- $\rho : E \to \mathbb{N}\backslash\{0\} \times \mathbb{N}\backslash\{0\}$ is a function that returns for each edge a pair (x, y), where x is the non-null production rate of its origin actor (producer) and y is the non-null 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).

A *port* is the connection point between an actor and an edge. The connection between an incoming edge (resp. outgoing edge) and an actor is the *input port* (resp. *output port*). An actor has a list of input ports and a list of output ports.

Figure 2.3 shows a simple SDF graph G_1 with 3 actors A, B, and C. The edge between A and B has a production (resp. consumption) rate of 2 (resp. 3). The edge (C, A) has 3 initial tokens.

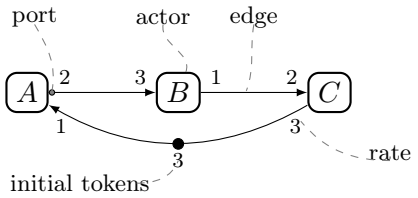


Figure 2.3: The SDF graph G_1.

Each edge has zero or more tokens at any moment. The *state* of an SDF 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 $[0; 0; 3]$ for the edge ordering $[(A, B); (B, C); (C, A)]$. Once A fires, it consumes 1 token from the 2 initial tokens available on the edge (C, A) and it produces 2 tokens on the edge (A, B). As a result the state of the dataflow graph changes to $[2; 0; 2]$ where the first element represent the 2 new tokens produced by A on the edge (A, B).

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

(A^3, B^2, C^1), where X^i means that actor X fires i times. We note $sol_G(X)$ the number of firings of X (also called the minimal solution of X) in the minimal iteration of the graph G, or $sol(X)$ when no ambiguity can arise. In the example, we have $sol(A) = 3$, $sol(B) = 2$, and $sol(C) = 1$. The *basic repetition vector* $\vec{Z}$ indicates the number of firings of actors (their solutions) per minimal iteration. For G_1, it is $\vec{Z}_{G_1} = [3, 2, 1]$ (for actors' ordering $[A, B, C]$). An iteration may be also not minimal. In an iteration which is not minimal, each actor is fired as many times as its solution multiplied by an arbitrary integer k. For example for $k = 2$, a non-minimal repetition vector corresponding to a non-minimal iteration for the graph G_1 is $[6, 4, 2]$. When we refer to solution or repetition vector, we mean always the minimal solution or minimal repetition vector.

We can represent an SDF graph using a *topology matrix*, similar to the way graphs are represented using an adjacency matrix in graph theory. The topology matrix of the SDF graph G, denoted by Γ_G, is a $|E| \times |V|$ matrix where $|E|$ and $|V|$ are the sizes of E and V. The value of the element $\gamma_{i,j}$ of Γ_G can be either a positive integer, a negative integer, or zero.

It is a positive integer if the j^{th} actor of V, that is v_j, is the origin actor of the i^{th} edge of E, that is e_i. In this case the value of $\gamma_{i,j}$ is the production rate of the edge e_i. The value of $\gamma_{i,j}$ is negative if v_j is the destination actor of e_i and its value is the consumption rate of the edge e_i. Finally the value of $\gamma_{i,j}$ is zero if v_j is neither the origin actor, nor the destination actor of e_i. For example, for the graph G_1 of Figure 2.3, we have:

$$\Gamma_{G_1} = \begin{array}{c} \begin{matrix} A & B & C \end{matrix} \\ \begin{pmatrix} 2 & -3 & 0 \\ 0 & 1 & -2 \\ -1 & 0 & 3 \end{pmatrix} \end{array} \begin{matrix} \\ (A, B) \\ (B, C) \\ (C, A) \end{matrix}$$

Later, we see how the topology matrix is used to check whether all actors have non-null solutions.

SDF has been used for many practical applications in numerous domains, including signal processing, automatic control, and embedded systems. Two important implementations of SDF are Ptolemy II [20] and SDF^3 [51].

2.3.2 Static analysis

Compared to KPN, SDF enforces constant production and consumption rates and it is, for this reason, amenable to more static analyses than KPN. We present three analyses provided by the SDF MoC: *consistency*, *boundedness* and *liveness*.

Consistency ensures that a given SDF graph has valid rates, which ensures that tokens do not accumulate on the edges during the execution. As a corollary, a consistent graph can run in bounded memory. Liveness ensures

that the graph can run infinitely without any deadlock. All these analyses are performed statically, that is before executing the graph.

Consistency and Boundedness

An SDF graph is said to be *consistent* if it admits a non null repetition vector. The repetition vector is obtained by solving the following *system of balance equations* with the solution of actors as variables. Each edge $X \xrightarrow{p \quad q} Y$ is associated with the balance equation $sol(X) \cdot p = sol(Y) \cdot q$, which states that all tokens produced by X during an iteration must be consumed by Y within the same iteration. The system has $|E|$ equations and $|V|$ variables. The SDF graph is consistent if and only if this system of equations admits a non-null solution, that is a repetition vector [38]. If the system does not have a non-null solution, the graph is *inconsistent*.

This system of balance equation can also be expressed using the topology matrix. For the topology matrix Γ_G of a given graph G, and its repetition vector $\vec{Z}_G$, we have:

$$\Gamma_G \cdot \vec{Z}_G = 0$$

For example for the graph G_1, the system of balance equations can be defined either by the three equations (on the left), or equivalently, by the matrix equation (on the right).

$$\begin{cases} 2 \cdot sol(A) & = 3 \cdot sol(B) \\ sol(B) & = 2 \cdot sol(C) \\ 3 \cdot sol(C) & = sol(A) \end{cases} \qquad \begin{pmatrix} 2 & -3 & 0 \\ 0 & 1 & -2 \\ -1 & 0 & 3 \end{pmatrix} \cdot \begin{pmatrix} sol(A) \\ sol(B) \\ sol(C) \end{pmatrix} = 0$$

Both methods give the same minimal non-null solutions: $sol(A) = 3$, $sol(B) = 2$, $sol(C) = 1$.

On the other hand, if we change the production rate of the edge (A, B) to 3, then the system does not have a non-null solution anymore and the graph becomes inconsistent. Indeed, after three firings of A, the actor B would be able to produce 3 tokens for C. The actor C would then consume only 2 of them, thereby producing 3 tokens on the edge (C, A). This behavior continues, and as a result tokens get accumulated on the edge (B, C).

In [7], an algorithm for computing the repetition vector with time complexity $O(|V| + |E|)$ is proposed. The algorithm initially sets the solution of a random actor to 1, and by traversing the graph (by breath first search or depth first search) finds the solutions of actors. The result is the solution of actors as rational numbers. By multiplying them by the least common multiple of all the fractions, minimal integer solutions are obtained.

An important consequence of consistency is that tokens do not get accumulated on edges. A consistent graph can therefore be executed infinitely with *bounded* memory (buffers).

Liveness

A *schedule* is a sequence of firings, so that each actor fires as many times as its solution in the repetition vector. Not all such sequences are valid, since an actor can be fired only when it has enough input tokens. A schedule is obtained by symbolic computation: When an actor cannot be fired, then that firing is *non-eligible*, and otherwise it is *eligible*. A schedule containing only eligible firings is called *admissible*. An admissible 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 has an admissible schedule [38]. A live SDF graph can execute infinitely without deadlock. In the following, we often use schedule for admissible schedule.

In order to compute a schedule, we use a symbolic computation algorithm that keeps track of the state of the graph and repeatedly adds the eligible firings to the schedule until either no actor can be fired, or until all actors have been fired as many times as their solutions. This class of scheduling algorithms are called class-S algorithms [38].

Among all admissible schedules, we distinguish *single appearance schedules* (SAS) (also called *flat SASs* in [1]) where, once factorized (*i.e.,* any sequence of n consecutive firings of actor X is replaced by X^n), each actor appears exactly once.

For example, the graph G_1 admits the schedule $\{A; A; B; A; B; C\}$, but the schedule $\{A; B; A; A; B; C\}$ is not admissible, because after the first firing of A there are only 2 tokens on the edge (A, B) and B cannot be fired. The graph has one flat SAS: $\{A^3; B^2; C^1\}$. In this thesis, whenever we talk about SAS, we mean a flat SAS. An example of a non-flat SAS is $\{(A; B; C)^2\}$ which is equivalent to $\{A; B; C; A; B; C\}$.

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. An acyclic SDF graph always admits an SAS. A cyclic SDF graph can be converted to an acyclic graph 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. The equivalent acyclic SDF graph is obtained by removing all the saturated edges from the cyclic SDF graph. Therefore, such acyclic graphs admit an SAS.

2.3.3 Performance analysis

Besides the static analyses of consistency, boundedness and liveness, SDF also allows performance analysis related to timing and memory consumption. Two important timing metrics are throughput and latency. The throughput measures the number of iterations completed per time unit, and the latency measures the number of time units it takes for an iteration to be completed. An important memory metric is the minimal size of buffers for the graph

to be live. These metrics can also be related. For example, an interesting property is the minimum buffer size that allows maximal throughput.

Each analysis is important for a category of applications. For example, in video streaming applications, the throughput is the most important one, because no matter how late the video is delivered, it is important that it is delivered at a desirable throughput in terms of images per second. In contrast, in a control-command application, latency analysis is the most important one, because no matter the throughput, it is important that commands are delivered as fast as possible.

In the previous section, we considered only the sequential execution of actors and we were only concerned with the total order of execution of actors. The schedules we discussed in the liveness analyses were sequential and unaware of any notion of time. In this part, we add the notion of time and consider parallel executions. A parallel schedule must choose the processor for each firing of actor as well as the time instants at which the actor must be fired. The performance analysis depends on the chosen scheduling policy. We may for instance look for a parallel schedule that maximizes the throughput.

On a multi-processor with multiple distributed memories, the communication costs between the processors are significant. For this kind of architecture, it is important to find an assignment of actors to processors (called placement) which minimizes the communication costs.

We first introduce some definitions about parallel scheduling and performance metrics, and then we discuss parallel schedules, latency and throughput analyses.

Definitions

For an SDF graph $G = (V, E, \rho, \iota)$, we define the following scheduling related functions, where time, duration, and the number of processors are modeled using integer numbers. Time starts from the first firing of a graph. It is also assumed that the processors are homogeneous, so the execution time of an actor on all cores is the same.

- $start : V \times \mathbb{N} \to Time$ is a function that returns for each actor v its start time $start(v, f)$ for a given firing f.
- $t : V \to Duration$ is a function that returns for each actor v its execution time $t(v)$.
- $end : V \times \mathbb{N} \to Time$ is a function that returns for each actor v its end time $end(v, f)$ for a given firing f.
  We have $\forall v \in V, 0 < f, end(v, f) = start(v, f) + t(v)$.

- $proc : V \times \mathbb{N} \rightarrow Processor$ is a function that returns for each actor v and a given firing f the processor $proc(v, f)$ on which the actor is fired.

Parallel schedule

A parallel schedule $S(G)$ defines for each firing of actors its start time and the processor on which the actor is executed.

$$S(G) = \{(start(v, f), proc(v, f)) \mid v \in V, 1 < f\}$$

We narrow down the scope of the problem by assuming that autoconcurrency is not allowed, which means that firings of the same actor cannot take place in parallel, that is, $start(v, f + 1) \geq end(v, f)$. We also assume that once an actor is assigned to a processor, it is executed only on that processor, that is, $\forall i, j, proc(v, i) = proc(v, j)$. So, we can simply write $proc(v)$ to indicate the processor on which the actor v executes.

In this thesis, we focus on as soon as possible (ASAP) scheduling policy. In this policy which is also referred to as self-timed execution, an actor is ready to fire as soon as it has enough tokens on its input edges. Therefore the scheduler must only define the processor on which each actor is executed. Assuming that there are enough processors, that is, the number of processors is greater than or equal to the number of actors, an actor executes as soon as it is ready. Otherwise, an actor must wait for other actors on the same processor to finish executing. We focus on the simplest case in which the number of processors is sufficient.

In a parallel dataflow execution, the execution starts with a prologue followed by a steady state. During the steady state, the start time of firing of a given actor in the schedule is *periodic*, that is, $\forall v, i, start(v, i + sol(v)) - start(v, i) = C$, where C is a constant. We see later that this is related to the throughput in the steady state.

In Figure 2.4, the ASAP schedule of the SDF graph $A \xrightarrow{3\ \ 2} B$ where $t(A) = 5$ and $t(B) = 3$ is shown. The first iteration in this schedule is the prologue phase (from time 0 to 16), and the steady state starts from the second iteration (from time 10 to 26). In the prologue phase, the first firing of the actor B starts just after the first firing of A in the iteration, while in the steady state, the first firing of B in an iteration must wait for the previous iteration to end. Later, we discuss about latency and throughput using this example.

The ASAP execution guarantees that the graph can be executed without deadlock, provided that each buffer has at least the minimal size required for liveness [41]. The liveness analysis supposes that buffer sizes are infinite, but in practice buffers are bounded and the graph may not be live if the buffer sizes are not sufficient.

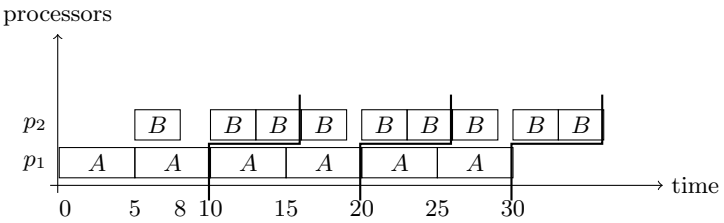


Figure 2.4: ASAP schedule of the SDF graph $A \xrightarrow{3\ \ 2} B$, where $t(A) = 5, t(B) = 3$.

An SDF graph can be converted to a directed acyclic graph (DAG) [37] and the scheduling techniques for DAGs can be also used for SDF graphs. In order to convert an SDF graph to a DAG, the graph is transformed to a simplified form called Homogeneous SDF (HSDF) graph in which a single firing of the original SDF graph is represented by a unique actor [37]. HSDF is a restriction of SDF in which all production and consumption rates are equal to 1. By transforming an SDF graph to an HSDF graph, the computation of latency and throughput becomes simpler. The equivalent HSDF graph of the SDF graph G_1 of the Figure 2.3 is shown in Figure 2.5.

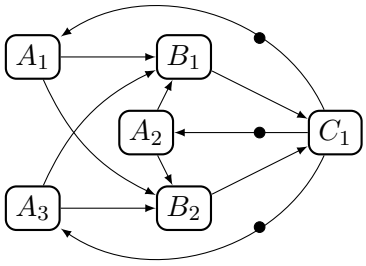


Figure 2.5: The HSDF graph equivalent to the SDF graph G_1.

Each of the 3 firings of the actor A in an iteration is represented by a single actor (namely A_1, A_2, and A_3), and similarly for the 2 firings of B (namely B_1 and B_2). The solution of actor C is 1, therefore there is a single actor C_1 in the HSDF graph. Each firing of actor C provides 3 tokens which are respectively consumed by 3 firings of the actor A, hence by A_1, A_2, and A_3 in the HSDF graph. Similarly, the initial tokens on the edge (C, A) in the original SDF graph are distributed among 3 firings of A. In the equivalent HSDF, these initial tokens are distributed on the 3 edges (C_1, A_1), (C_1, A_2), and (C_1, A_3).

Latency

The *end-to-end latency* of a given iteration of a graph is the largest end time of the last firings of all actors minus the smallest start time of the first firings of all actors in that iteration. Formally, the latency of the i^{th} iteration is

$$\mathcal{L}_G(i) = \max_{v \in V} end(v, i \cdot sol(v)) - \min_{v \in V} start(v, (i-1) \cdot sol(v) + 1)$$

For example, in Figure 2.4, the latency of each iteration in prologue or in steady state is 16, that is, $\forall i, \mathcal{L}_G(i) = 16$. For the second iteration, we have:

$$\begin{aligned}\mathcal{L}_G(2) &= \max_{v \in V} end(v, 2 \cdot sol(v)) - \min_{v \in V} start(v, sol(v) + 1) \\ &= \max(20, 26) - \min(10, 16) = 26 - 10 = 16\end{aligned}$$

For any edge $e = (x, y)$ with $\rho(e) = (p, q)$ and $\iota(e) = d$, we can establish the following relations:

$$\begin{aligned}start(y, k) &\geq end(x, \left\lceil \frac{k \cdot q - d}{p} \right\rceil) \\ start(y, k) &\geq end(y, k-1)\end{aligned}$$

The k^{th} firing of the actor y of the edge (x, y) can start only if all required tokens for that firing are produced, that is, $k \cdot q$ tokens. The edge has d initial tokens, therefore the actor x needs to fire $\lceil \frac{k \cdot q - d}{p} \rceil$ times, before the k^{th} firing of y can take place.

Therefore, if y has only (x, y) as incoming edge, in an ASAP schedule with enough processors, we have:

$$start(y, k) = \max(end(x, \left\lceil \frac{k \cdot q - d}{p} \right\rceil), end(y, k-1))$$

If the actor y has multiple incoming edges, then in an ASAP schedule, we need to compute the maximum end time over all predecessors x of y, so the starting time of the k^{th} firing of y is:

$$\begin{aligned}start(y, k) = \max(&\max_{\forall x, (x,y) \in E} (end(x, \left\lceil \frac{k \cdot cons((x,y)) - \iota((x,y))}{prod((x,y))} \right\rceil)), \\ &end(y, k-1))\end{aligned}$$

where $prod((x, y))$ and $cons((x, y))$ are the production and consumption rates of the edge (x, y) respectively.

For example, in Figure 2.4, the 5^{th} firing of actor B occurs only after the 4^{th} of actor A. Using the formula, we have:

$$\begin{aligned}start(B, 5) &= \max(end(A, \left\lceil \frac{5 \cdot 2}{3} \right\rceil = 4), end(B, 4)) \\ &= \max(20, 19) = 20\end{aligned}$$

The notion of *multi-iteration latency* is the *execution duration* $\mathcal{D}_G(n)$ of the first n iterations. It is equal to the end time of the last firing of the n^{th} iteration. In the sequel, we will use the multi-iteration latency $\mathcal{D}_G(n)$ to compute the period and the throughput of an SDF graph.

An algorithm to find the minimal latency for SDF graphs is proposed in [28] both for single-processors and multi-processors with sufficient resources to exploit all available parallelism. By adding one source and one sink to the graph, it produces a so-called latency graph used to compute the minimal end-to-end latency. In order to compute the minimum latency between the source and the sink actors, the latency of all paths between those two actors must computed. The number of all these paths is exponential in the number of actors of the graph, and therefore, the computation of the latency is also exponential in the number of actors.

In [48], it is proposed to use max-plus algebra to compute a tighter worst-case latency estimate than other existing approaches. The complexity of this approach is still exponential in the size of the dataflow graph, but it is shown to be practical for realistic case-studies.

Throughput

The *period* of a given iteration i of a graph G is the largest end time of the last firings of all actors in i^{th} iteration minus the largest end time of the last firings of all actors in the $(i-1)^{th}$ iteration.

$$\mathcal{P}_G(i) = \max_{v \in V} end(v, i \cdot sol(v)) - \max_{v \in V} end(v, (i-1) \cdot sol(v))$$

The *throughput* of a given iteration is defined as the inverse of the period of that iteration:

$$\mathcal{T}_G(i) = \frac{1}{\mathcal{P}_G(i)}$$

In Figure 2.4, the period of the first iteration (the prologue) is 16, but from the second iteration the period is 10. So the throughput during the steady state is $\frac{1}{10}$.

In a sequential execution where there is no parallelism, each iteration starts once its previous iteration has finished. In such case, the period of an iteration is equal to the end-to-end latency. When there is parallelism, the average period $\mathcal{P}_G$ can be computed as the average time an iteration takes. Therefore, we can compute the average period as follows:

$$\mathcal{P}_G = \lim_{n \to \infty} \frac{\mathcal{D}_G(n)}{n}$$

where we recall that $\mathcal{D}_G(n)$ is the multi-iteration latency of G. The average throughput is the inverse of the average period:

$$\mathcal{T}_G = \frac{1}{\mathcal{P}_G}$$

ASAP scheduling allows a graph to reach its maximum throughput [9, 11], provided that the buffer sizes are sufficient. This throughput for an acyclic graph depends only on the actor whose execution of all its firings is the longest in an iteration. It is equal to:

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

The reason is as follows. In an ASAP schedule with enough processors, each actor executes on a single processor. In the steady state, at each iteration, each actor must wait for its inputs to be ready except any actor v_m for which its $sol(v_m) \cdot t(v_m)$ is maximum. This is because all immediate predecessors of v_m take at most $sol(v_m) \cdot t(v_m)$ to complete their execution during an iteration. The actor v_m executes continuously in the steady state. Therefore, during the steady state, the period of an iteration is the duration the actor v_m takes to execute, and the throughput is the inverse of the period.

The throughput of an SDF graph can be computed by converting it into an HSDF graph and by finding the inverse of the maximum cycle mean (MCM) of the equivalent HSDF graph [50]. The cycle mean of a cycle is equal to the sum of execution times of the actors in the cycle divided by the number of initial tokens on the edges of this cycle [47]. This approach may be impractical for large graphs, because the complexity of the conversion from SDF to HSDF is exponential in the number of actors. Moreover, in order to compute the MCM, all cycles have to be found. The number of all cycles are also exponential in the number of actors.

In [27], an approach for computing the throughput is proposed in order to avoid transforming the SDF graph into an HSDF graph. This approach, which is based on a state space exploration, does not improve the complexity which is exponential in the size of the dataflow graph, but it is shown to perform better than other existing approaches in practice.

2.4 Extensions of Synchronous Dataflow

We can categorize dataflow MoCs into two classes : static dataflow MoCs [29] and dynamic dataflow MoCs [6]. The static MoCs cannot change at run-time whereas the ones in the second class provide mechanisms for reconfiguring the dataflow graph. We can further distinguish two types of reconfiguration: Parameterization and dynamic change of the topology of the graph.

By parameterization, we mean replacing a constant value by a parameter whose value can change dynamically. Features of a dataflow model that can be parameterized include values of production and consumption rates and numbers of initial tokens. The functions of actors can be also subject of change, which may result in change in the execution time of the actor.

There are also some MoCs such as Cycle-static dataflow (CSDF) [8] which allow the production and consumption rates to be changed without providing parameters, but rather by choosing the rate among a finite and statically known number of values.

If a dataflow MoC changes the set of actors and edges at run-time, we say that it changes the topology. For example, by allowing the state of a communication channel to be enabled or disabled using parameters, the topology can change during the execution. Another way to change the topology is to replace a dataflow graph by another one.

We present the class of MoCs allowing parameterization and the class of MoCs that allow dynamic change of the topology in turn. MoCs that allow enabling and disabling edges using parameters are also presented into the second class.

2.4.1 MoCs supporting parameterization

Parameterization refers to the change of constant values of the MoC by parameters. For example, rates are constant in SDF, but they may need to be changed for a specific application. An actor may need to consume a certain number of tokens during a time interval and another number of tokens during another interval. In this section, we present dataflow MoCs allowing the parameterization of some of their features.

Parametric Synchronous Dataflow (PSDF)

Parametric Synchronous Dataflow (PSDF) is the first parametric dataflow MoC that was proposed [5]. A PSDF actor is parameterized with a set of parameters. These parameters can be used to change the production and consumption rates, number of initial tokens and the functionality of an actor. The parametric rates can only take positive value (null rates are not allowed).

A PSDF actor is either a normal actor as in SDF with possibly parametric rates, or it is a *hierarchical* actor. An example of a hierarchical actor is shown in Figure 2.6. The input and output ports of a hierarchical actor, which are called *interface ports*, are connected to the actors in the higher level of hierarchy. Each hierarchical actor consists of a mandatory graph called the *body* graph with possibly parametric rates which models the functionality of the hierarchical actor, and two optional graphs called init and subinit graphs which are responsible for setting the parameters.

In the Figure 2.6, the *body graph* consists of 3 actors, actor A connected to an input interface port with a parametric rate p on its output edge, actor B with an internal parameter g to change its functionality, and actor C connected to an output interface port. The *subinit graph* contains a single actor that sets the parameter g and the *init graph* contains a single actor that

sets the parameter p. The schedule for this hierarchical actor is $\{A; B^p; C^p\}$, where B^p stands for the repetition of B's firings p times. If the solution of this hierarchical actor in its parent graph is x, then the schedule of an iteration becomes $\{(A; B^p; C^p)^x\}$.

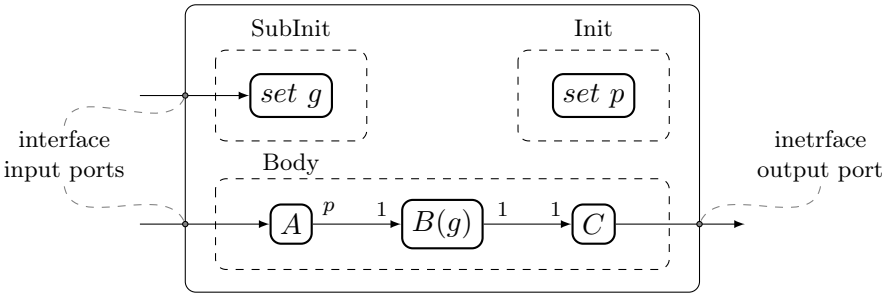


Figure 2.6: An example of a hierarchical actor in a PSDF graph

A hierarchical actor communicates with its parent actor, that is, the actor containing it, using the interface ports. The init graph is used to set the parameters and is called at the beginning of each firing of the parent actor, while the subinit graph is used to configure the parameters at the beginning of each iteration of the body graph, therefore it is invoked more frequently than the init graph. For example, in Figure 2.6, the hierarchical actor has two input interface ports and one output interface port. The init graph which sets the parameter p (output rate of A) is called each time the parent graph of the hierarchical actor starts its firing. The subinit graph that sets the parameter g (which may modify the functionality of the actor B) is called at the beginning of the iteration of the body graph.

Each hierarchical actor is analyzed separately. PSDF checks that all configurations of the body graph of every hierarchical actor are consistent and live.

Another MoC which is very similar to PSDF and offers hierarchical dataflow with parameters is Parameterized and Interfaced Synchronous Dataflow (PiSDF) [19]. It is simpler than PSDF as it does not separate init, subinit and body graphs. Each hierarchical actor is executed at each iteration. The idea of parameterization of PSDF has also been used to extend cyclo-static dataflow (CSDF), resulting in PCSDF [34].

In all these MoCs, it is not possible to modify the topology of the dataflow graph, although it is possible to change the functionality of actors, which can be seen as replacing actors.

Schedulable Parametric Dataflow (SPDF)

Schedulable Parametric Dataflow (SPDF) [23] is a parametric dataflow MoC where each rate is either a positive integer, or a parameter, or a product of positive integers and parameters.

Setting the parameters of an SPDF graph can occur within an iteration and it is not restricted to iteration boundaries as in PSDF. Each parameter has a unique actor, its *modifier*, which may modifying it. The actors that have parametric rates are the *users* of parameters. The modification is done according to a period. The annotation $p@\pi$ attached to an actor A indicates that the modifier actor A can set the value of the parameter p at every π firings of A. Periods can also be parametric. The number of modifications of a parameter p by A with the annotation $p@\pi$ within an iteration is computed *symbolically* as $sol(A)/\pi$. This is called the *frequency* of p, which must denote an integer (*i.e.,* π must be a divisor of $sol(A)$). The frequency can also be parametric, but it cannot be a symbolic fraction since it has to be an integer. The values of parameters are communicated by the modifiers to the users through special communication links that are added automatically by the compiler. These edges, which control the behavior of the SPDF graph, are called *control edges*.

Figure 2.7 shows a simple example of an SPDF graph. The actor A is the modifier of the parameter p and it can modify it every p firing (annotation $p@p$). The repetition vector, symbolically computed, is $[2p, 2, 3]$, therefore the actual frequency of the modification of parameter p is 2. There is also a parametric single appearance schedule $\{A^{2p}; B^2; C^3\}$. The expression A^{2p} means that A is executed $2p$ times consecutively. Each (implicit) control edge is added by the compiler. In Figure 2.7 we depict the control edge from A (the modifier of p) to B (the unique user of p) as dashed arrow, with the annotation $[p]$ to show that the control edge from A to B carries the parameter p.

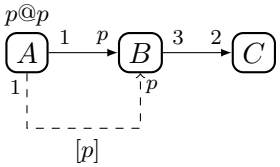


Figure 2.7: An example of a SPDF graph

Boundedness and liveness of SPDF graphs can be analyzed statically. These analyses are *symbolic*, that is, they are performed with regard to the parameters of the graph. For example, the solution of the actor A which is $2p$ is symbolic. Similarly, a symbolic schedule is expressed using the parameters of the graph. Because of the flexibility of rate modification, consistency is not sufficient for boundedness. In order to guarantee boundedness, every parameter p must be modified at the boundaries of the local iteration of the subgraph made of the users of p. This subgraph is called the *region* of parameter p. For liveness, it must be also verified that there is a directed path in the dataflow graph from each modifier to all the users. This prevents

the control edges to introduce non-live cycles.

Similar to PSDF, SPDF does not allow changing actors or edges. Therefore, in terms of dynamic change of topology it is the similar to PSDF, although it provides a higher degree of flexibility in terms of rate modification.

2.4.2 MoCs supporting dynamic topology

In the following, we present the dataflow MoCs that allow the topology of the dataflow graph to be changed dynamically.

Variable Rate Dataflow (VRDF)

Variable Rate Dataflow (VRDF) [54] is an MoC with parametric rates. The domain of a parameter can contain zero, but needs at least one strictly positive value. By accepting null rates, VRDF also allows changing the topology of the graph in the sense that a null rate disables its port.

Parameters can change within the iterations and they are set by a modifier actor which can modify their value at any of its firings. Because of this parameterization mechanism, consistency (evaluated as in SDF by checking the existence of a non-null repetition vector) does not imply necessarily boundedness and more restrictions are needed. Each parameter is used by at most two actors, its modifier, and one possible user. For any two actors using the same parameter, their symbolic solutions must be equal. For liveness, a symbolic schedule must be found.

Figure 2.8 shows an example of a VRDF graph. It respects all the constraints, since the solutions of both actors A and B are 2 and the parameter p is used in the output port of its modifier as well as the input port of its user. The iteration is (A^2, B^2, C). Similar to SPDF, the control channel from A to B (depicted as a dashed arrow) specifies explicitly the parameter it carries, which is p in this example.

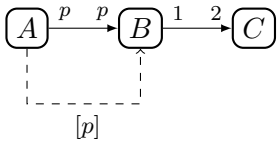


Figure 2.8: An example of a VRDF graph

The idea of parameterization of VRDF has also been applied to CSDF, resulting in Variable Rate Phased Dataflow (VPDF) [55] in which actors periodically change phase while their rates change according to their sequence of rates. Those rates are parametric as in VRDF.

Although changing the topology of the graph is possible in VRDF and VPDF (as ports can be deactivated using null rates), it is limited. For

example, suppose a graph needs to add actors to the graph at runtime. Firstly, all those actors must be foreseen in the initial graph. Secondly, VRDF allows only at most two actors to use the same parameter and as a result the number of required parameters for enabling and disabling n actors is linear in n.

In Figure 2.8, if $p = 0$, then the graph is reconfigured into the graph $B \xrightarrow{1\ \ 2} C$. The iteration changes to (B^2, C). Since A does not fire anymore, there must be a mechanism in place in order to change the value of p again, otherwise the graph cannot be reconfigured again. Therefore, the capability of VRDF for dynamic changes of topology is not only limited, but also raises issues [10].

Boolean Dataflow (BDF)

Boolean Dataflow (BDF) [16] is one of the early MoCs that allows changing the topology of the graph in a very limited way. It adds the ability of conditional branching to SDF by using two special actors, *switch* and *select*. The switch actor has a single input edge and two output edges, labeled true and false and an additional incoming edge to communicate a boolean token to the actor (see Figure 2.9). If the value of the boolean token is true, the actor sends its output on the edge labeled true, otherwise the output is sent to the edge labeled false. The select actor is the dual of this actor. It has two inputs labeled true and false and one single output, and the value of the boolean parameter determines which input is read by the actor.

Figure 2.9 shows a simple example of a BDF graph. The *control input* b is fed to both the switch and select actors. In this way, if the value of the control input b is true, the flow goes from A to B to D, otherwise, the flow passes through C instead of B.

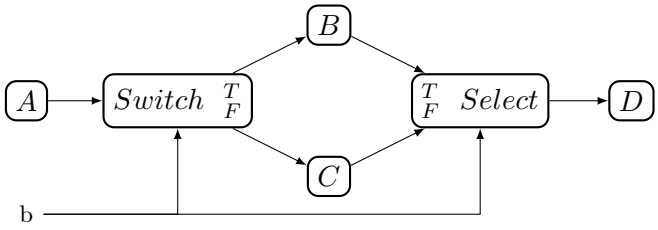


Figure 2.9: An example of a BDF graph

For consistency analysis, a probabilistic rate p_i is assigned to each control input b_i. This probabilistic rate corresponds to the probability of a control input to take true or false values. For example, a switch actor with the probability p for taking true values is supposed to produce np tokens on its output labeled true after n firings and $n(1-p)$ tokens on its output labeled false. The vector $\vec{p}$ contains the probabilities for all control inputs. The

system of balance equations of BDF contains such probabilistic values. If the system of balance equations for a given BDF graph has solutions for $\vec{p}$, the graph is *strongly consistent* and therefore it is consistent and bounded. The liveness analysis also takes these probabilities into consideration.

The Integer Dataflow (IDF) MoC [15] extends BDF with switch and select actors that select one edge among many edges according to the value of a control integer parameter. Dynamic change of topology is possible using BDF and IDF but in a limited way because actors have to be foreseen in the graph and as in VRDF only edges can be dynamically enabled and disabled.

Boolean Parametric Dataflow (BPDF)

Boolean Parametric Dataflow (BPDF) [3, 2, 4] is another parametric dataflow MoC that also allows dynamic change of the topology. These changes are done using parametric boolean conditions on edges. The conditions either activate or deactivate their associated edges and as a result change the topology of the graph. However, new actors cannot be added to the graph.

Similar to SPDF, the production and consumption rates are either positive integers or a product of integers and parameters, although changing the value of parameters is only possible at iteration boundaries. Unlike VRDF in which at most two actors can share a given rate parameter, in BPDF boolean and integer parameters can be shared among any number of actors.

BPDF provides in addition a mechanism to activate and deactivate edges using boolean conditions on the edges. These conditions are expressed using boolean parameters that can be changed within iterations. Each edge in a BPDF graph can have a boolean condition which is a combination of conjunctions and disjunctions and negations on a set of boolean parameters. If the condition evaluates to true, then the edge is activated and it behaves as an SDF edge. If the condition evaluates to false, then the edge is deactivated and its origin and destination actors do not write on nor read from it. When no boolean condition is specified, the edge behaves as an SDF edge.

A given boolean parameter can appear in several edge conditions in order to activate and deactivate them at the same time. For example, a single parameter b can be used to make a mutual exclusion between two edges labeled by two conditions b and $\neg b$, whereas this facility is not provided in VRDF.

Similar to the parametric rates in SPDF, each boolean parameters in BPDF has a unique modifier. All actors that are the origin or destination of an edge with a condition containing b are the users of the parameter b. Unlike the rate parameters, the boolean parameters can be modified within iterations. At each firing, if the value of a parameter b changes, its modifier propagates this new value of b to all the users of b. As in SPDF, each modifier of a boolean parameter specifies its period π, that is, the number of firings between two consecutive changes of a parameter. It is written as $b@\pi$. The

frequency of change of a boolean parameter b denoted by $freq(b)$ is defined as the symbolic solution of its modifier M, divided by the period of change π of the parameter, that is, $freq(b) = sol(M)/\pi$. Two restrictions are imposed on frequencies. First, the frequency of each boolean parameter must be an integer. Second, for every boolean parameter b with $freq(b)$ and for all users U of the parameter b, the value $freq(b)/sol(U)$ must be an integer. If these two conditions hold, the BPDF graph under study is called *period safe*.

An example of a BPDF graph is shown in Figure 2.10. The iteration is $(A^p, M^1, B^1, C^1, D^1, N^1, E^p)$. The graph contains two boolean parameters b and b'. The modifier of both parameters is the actor M. The period of both parameters b and b' is 1, which is equal to the solution of actor M. It means that they can be changed at each iteration and their frequency of change is 1. As a result, at each iteration the application may change from executing B to C and vice-versa. There is also one rate parameter p which can be changed at the beginning of each iteration.

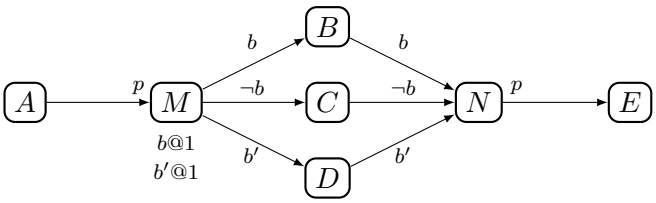


Figure 2.10: An example of a BPDF graph

If a BPDF graph is consistent and period safe, then it is bounded. This ensures that for any given edge with a boolean condition C, the number of tokens produced on that edge is equal the number of tokens consumed on that edge during any iteration, no matter what values the condition C takes during the iteration. For consistency analysis, because of the presence of parametric rates, the system of balance equation has symbolic solutions. This symbolic consistency analysis does not take the boolean conditions into account. For liveness analysis, the control edges for communicating the parameters (both integer and boolean) must be taken into account.

One of the applications of BPDF is to model an environment where the dataflow channels of SDF may be lossy and retransmission protocols must be considered. In such environment, retransmissions need to activate and deactivate ports of the graph. A translation of SDF graphs where some channels may be lossy into BPDF has been proposed in [21].

Dynamic change of topology is possible using BPDF as it was shown in the example. However, as all previous MoCs, all the topologies must be foreseen at compile-time and all the actors must be present in the BPDF graph.

Scenario-Aware Dataflow (SADF)

The Scenario-Aware Dataflow (SADF) MoC [25, 52] and its extensions such as Parameterized SADF (πSADF) [49] are MoCs which add a finite state machine (FSM) to the dataflow graph. By adding an FSM, these MoCs can specify several dataflow graphs and the FSM describe how to switch from one graph to another.

An SADF application consists of an FSM with a number of states, each associated with a dataflow graph. A state in SADF is called a *scenario*, which is one of the possible topologies of the application. At the end of each iteration, a transition can take place in the FSM to change the current dataflow graph.

Figure 2.11 shows an SADF application consisting of two scenarios and one FSM. In these graphs all rates are equal to 1. The finite state machine consists of two states S_1 and S_2 for scenario 1 and scenario 2 respectively. In scenario 2 the actor D is added. The execution starts with the state S_1. After each iteration, the state can change non-deterministically to the scenario S_2 or remain in S_1 (non-deterministically because none of these two transitions are labeled with a condition). There is also a transition from S_2 to S_1 which shows that the state can change non-deterministically from S_2 to S_1.

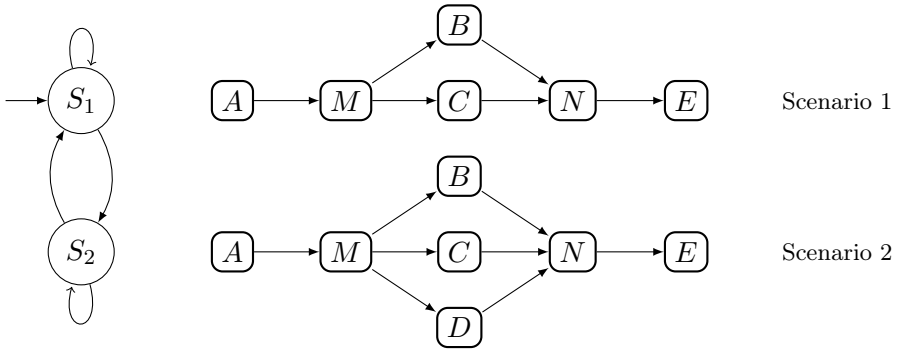


Figure 2.11: An instance of the SADF MoC

In SADF, the scenarios are SDF graphs with constant rates, whereas in its parameterized extension (πSADF), the rates can be either positive integers or products of positive integers and parameters. In πSADF, restrictions are imposed to guarantee boundedness and liveness. First, if a dataflow edge contains initial tokens, then the solution of its destination actor cannot be parametric and it must be a constant integer. Moreover, any edge with initial tokens must be saturated, that is, the number of initial tokens on the edge must be greater than the solution of its destination actor multiplied by the maximum value of its input rate. This requires that for each parameter, its maximum value must be specified. Second, each graph must have a

possibly parametric single appearance schedule. Third, the self-loops, that is, the edges with the same origin and destination, must have exactly one initial token and rates equal to 1. The graphs of all scenarios are analyzed statically and all must be consistent and live. πSADF also proposes methods to compute the worst-case throughput based on max-plus algebra assuming that the scheduling policy is self-timed [26, 48].

As it was shown in Figure 2.11, using SADF it is possible to change the topology dynamically. However, the number of topologies is bounded by the number of states of the FSM, and in πSADF, the number of possible configurations of each scenario is dependent on the maximum values of parameters. All possible topologies must be foreseen and be specified as a scenario.

Parameterized sets of modes Runtime Environment (PRUNE)

Parameterized sets of modes Runtime Environment (PRUNE) [13, 12] is a parametric dataflow MoC that captures the functionality of data dependent signal processing algorithms that require reconfiguration. Parameterized sets of modes (PSM) [39] refers to the ability of modeling a collection of configurations, where one or more parameters are used in order to choose a mode from the collection.

Each dataflow edge has a single rate which denotes the production and consumption rate of that edge. Each port of an actor is either a control port, a static port, or a dynamic port. Static ports are similar to the ports in SDF with a single constant positive rate. The rate of the dynamic ports can take two values: a fixed positive integer or zero. Therefore, a dynamic port can be activated and deactivated. The control ports control the activation of the dynamic ports. The consumption rate of the control ports is always one. An actor can be static or dynamic. Static actors have only static ports and they are similar to SDF actors with constant rates. Dynamic actors have control and dynamic ports and can contain also static ports. Before firing, a dynamic actor consumes one token from each of its control ports. Based on the value of the control tokens it has just read, it sets the value of the rates of its dynamic ports to either zero or their constant value. Then, it consumes data from the ports with non-null rates, processes the data, and finally produces data to the ports with non-null rates. Each data channel can contain zero or one initial token, while the control channels never contain any initial token.

In Figure 2.12, we can see an example of a PRUNE graph. Each edge carries a single rate corresponding to its consumption and production rates. Actor Q controls the output dynamic ports of the actor M connected to actor C. It can set it to either zero or one and, in this way, the edge (M, C) can be activated or deactivated.

PRUNE ensures consistency and liveness statically using some design constraints. For example, one constraint is that each pair of dynamic ports

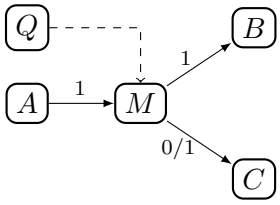


Figure 2.12: An example of a PRUNE graph

connected to each other must be controlled using the same control port. Another constraint is that a dynamic actor may have only dynamic input ports or dynamic output ports, but not both. The imposed constraints ensure that all possible topologies are consistent and live.

Dynamic reconfiguration is possible in PRUNE, but still it is limited. Although using a set of control edges allows creating several topologies, still all actors must be present in the initial graph. Similar to BPDF, PRUNE allows activating and deactivating edges, but because of the constraints, it appears less expressive. Compared to SADF, it is more difficult for PRUNE to define very different dataflow graphs in one application. SADF may changes the topology by changing completely the graph, whereas PRUNE as BPDF is limited to activate and deactivate edges.

2.5 Summary

The focus in this thesis is the class of concurrent models of computations and, more precisely, dataflow MoCs. These models can be characterized and classified using criteria, such as analyzability, expressiveness, and reconfigurability. The more a model is analyzable, the easier we can analyze it for certain properties such as liveness, boundedness, and non-functional properties such as latency and throughput. The more expressive an MoC is, the larger the class of applications it can express. Usually, more analyzable models are less expressive. Finally, the more reconfigurable an MoC is, the more its applications can adapt to the environment. The focus of this thesis is the reconfigurability of the paradigmatic dataflow MoC, that is, Synchronous Dataflow (SDF).

In this chapter, we studied SDF and discussed about its key static analyses, that is, boundedness and liveness. We also studied performance analyses of SDF. Based on the notions of start time of an actor and its execution time, we defined latency and throughput and discussed how they can be computed. Parallel scheduling of a graph amounts to allocating time and processing units to each of its actors. We defined the scheduling policy we focus on, which is the ASAP or self-time policy. In a self-timed execution, an actor is executed as soon as it is ready and has enough tokens on its incoming edges.

We studied different extensions of SDF, focusing on dynamic reconfigurations. We presented Parameterized Synchronous Dataflow (PSDF), Schedulable Parametric Dataflow (SPDF), Boolean Parametric Dataflow (BPDF), Variable Rate Dataflow (VRDF), Parameterized sets of modes Runtime Environment (PRUNE), Boolean Dataflow (BDF), and Scenario-Aware Dataflow (SADF).

PSDF and SPDF only allow changing the consumption and production rates of the actors. PSDF also allows changing the functionality of an actor, but it is a very limited reconfiguration facility.

VRDF, BPDF, and PRUNE allow changing the topology of the dataflow graph by different techniques. VRDF allows rates to be zero, so that an actor can be deactivated, BPDF allows edges to be deactivated using boolean parameters, and PRUNE allows ports of actors to be deactivated by providing dynamic ports. Although these MoCs are different in terms of expressiveness, in terms of reconfigurability they all have the same disadvantage: all possible configurations must be known before starting the execution. BDF is another MoC which allows changing the topology using two special actors called switch and select.

SADF allows changing the topology by changing the state of the application, each state being a different dataflow graph. The disadvantage is that the number of configurations we can have at compile is bounded and in practice usually small.

To conclude, all the MoCs we studied permit in practice a small number of configurations, which in addition must be foreseen before execution. They allow dynamic changes of the topology by enumerating all possible graphs at compile-time. In this thesis, we address this shortcoming and propose Reconfigurable Dataflow (RDF), a new MoC allowing to add and remove actors and edges at run-time. All possible configurations of an RDF application do not have to be explicitly stated at design time and their number can be arbitrary large or even unbounded.

Chapter 3

Reconfigurable Dataflow

Dynamic reconfiguration refers to the use of mechanisms to change dynamically the topology of the graph. Two graphs have the same topology if there is a bijection f from the set of actors of the first graph G to the set of actors of the second graph H such that the edge $A \xrightarrow{p \quad q} B$ is present in G if and only if the edge $f(A) \xrightarrow{p \quad q} f(B)$ is present in H. In such a case, the two graphs are called isomorphic. Two isomorphic graphs have the same number of actors and the same number of edges.

Different extensions of the SDF MoC provide mechanisms for reconfiguration differently. Some allow changing topology by providing a mechanism to activate and deactivate edges and ports, and some by replacing a graph by another graph. One drawback of all these existing dataflow MoCs is that the number of topologies they can generate at execution time is bounded and is known at design time.

In this chapter, we introduce the *Reconfigurable Dataflow (RDF)* MoC which allows dynamic reconfigurations in such a way that the number of topologies generated by an instance of the MoC can be unbounded. For instance, in a multimedia application, an unbounded number of image processing filters can be added. We also provide static analyses to verify that all possible topologies generated by a given RDF application are connected, bounded, and live. The mechanism used by RDF to achieve reconfigurability is the concept of graph transformation (or graph rewriting) which is elaborated in this chapter.

3.1 The Reconfigurable Dataflow MoC

The RDF MoC extends SDF with *actor types*, *explicit ports*, and a *reconfiguration controller*. The reconfiguration controller consists of a set of *transformation rules* and a set of *conditions*. A transformation rule is a graph rewrite rule describing how the current dataflow graph is modified. A graph is modified by replacing one of its sub-graphs by another sub-graph. A con-

dition is a boolean expression describing when the transformation rules are applied. By applying a transformation rule, actors and communication links can be moved, suppressed 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 introduce new actors in the graph as new type instances. For instance, a video application may require to introduce dynamically several noise filters at different places in the graph. This may be done by introducing new actors, instances of the noise filter type, in the graph. Transformation rules, types and instances allow the number of actors and the size of 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 to discriminate between two edges of the same actor bearing the same rates.

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

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

An RDF application starts by executing its initial graph, until a first transformation rule is 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.

3.1.1 Dataflow 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

- $T \subseteq Id_T \times \mathbb{N} \times \mathbb{N} \times (\mathbb{N}^* \to \mathbb{N}^*) \times (\mathbb{N}^* \to \mathbb{N}^*)$ 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 output ports respectively. A type $t = (i, k_1, k_2, f_1, f_2)$ has the identifier i, k_1 input ports, k_2 output ports, and the function $f_1(j)$ (resp. $f_2(j)$) returns the rate associated with the jth input port (resp. output port). The functions *idof*, *nbin*, *nbout*, *finr*, *foutr* return the identifier, number of input ports, number of output ports, input and output rate functions of their type argument respectively. For instance, $\mathit{finr}(t, \mathit{nbin}(t))$ 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 ($i \in \mathbb{N}^*$). 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. An edge $((a, i), (b, j))$ connects the ith output port of actor a to the jth input port of actor b.
- $\iota : E \to \mathbb{N}$ is a function that returns, for each edge, the number of its initial tokens (possibly 0).

In the following, we use capital letters for type identifiers from the set Id_T. Typically an actor is denoted by its type identifier and an index: A_2 denotes an actor of type A.

We consider only well-formed graphs, *i.e.,* properly connected and typed graphs. Formally,

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

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

$$\begin{array}{l} \forall a \in V, \forall\ 1 \leq i \leq nbin(typeof(a)), \forall\ 1 \leq o \leq nbout(typeof(a)), \\ \exists!(a', o', a'', i') \in V \times \mathbb{N}^* \times V \times \mathbb{N}^*, \\ \quad s.t.\ ((a', o'), (a, i)) \in E \wedge ((a, o), (a'', i')) \in E \end{array}$$

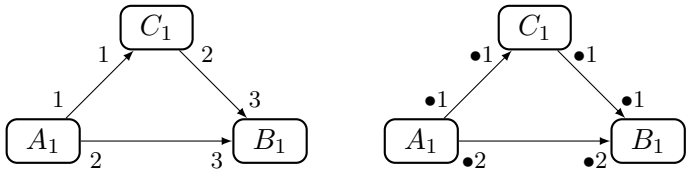
Edges are *valid* if they connect only actors of the graph and only through ports permitted by the actors' type. Formally,

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

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

To facilitate the reading, RDF graphs are often represented as in SDF with implicit ports and explicit rates. The graph G_0 of the Figure 3.1(a) is an RDF graph where A_1, B_1 and C_1 are actors of types A, B and C respectively. The graph has the same repetition vector and schedules as its SDF version. Its schedule is $\{A_1^3; C_1^3; B_1^2\}$.

The ports can also be explicit. The input ports are numbered from 1 to n, and output ports from 1 to m, where n and m are integers. The graph G_0 is shown in the Figure 3.1(b) with explicit ports and implicit rates. The notation $\bullet j$ indicates the port number j. In Figure 3.1(b), the actor A_1 has two output ports $\bullet 1$ and $\bullet 2$, the actor B_1 has two input ports $\bullet 1$ and $\bullet 2$, and the actor C_1 has one input port $\bullet 1$ and one output port $\bullet 1$.



(a) (b)

Figure 3.1: Graph G_0 with (a) explicit rates and implicit ports, and (b) explicit ports and implicit rates ($\mathit{foutr}(A,1) = 1, \mathit{foutr}(A,2) = 2, \mathit{finr}(B,1) = 3, \mathit{finr}(B,2) = 3, \mathit{finr}(C,1) = 1, \mathit{foutr}(C,1) = 2$).

We could also show both ports and rates on the figure, but for simplicity of the representation, we use either rates or ports.

3.1.2 Reconfiguration controller

The reconfiguration controller of RDF specifies when and how the current dataflow graph is modified. The basic operations are *transformation rules* which are specified as graph rewrite rules determining how a sub-graph of the current dataflow graph is replaced by another sub-graph. A *transformation program* combines the transformation rules in different ways. The controller has a *reconfiguration program* which specifies the conditions for applying the transformation programs.

Therefore, the reconfiguration program consists of a *sequence* of pairs of conditions and transformation programs:

$$[\mathit{cond}_1 : P_1; \ldots; \mathit{cond}_n : P_n]$$

Each condition cond_i is a boolean expression. For instance, a condition may check whether the throughput of the current dataflow graph is superior or inferior of a certain value. More details on the concrete examples of conditions are provided in chapter 5. The language for describing conditions is not part of the RDF MoC, because it involves elements (such as the throughput) that are external to the RDF application (see section 5.1.3) and because it does not interfere with static analyses.

The simplest option for transformation programs is to consider them made of a single transformation. This is the language we have used to

implement RDF (see chapter 5). Many more expressive options are possible. We describe in the following one possible and more expressive language.

A transformation program can be a combination of *transformation rules* (see section 3.1.3) 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 subgraph of G. 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 may 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.

If one condition $cond_i$ is satisfied, then the controller stops the execution of the RDF graph (see chapter 5 for more details), applies the transformation program specified by P_i, and finally resumes the execution. Only one pair $(cond_i : P_i)$ is selected. If more than one condition is true, then the first true condition in the sequence is chosen.

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

To ensure that a controller always preserves the 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 (see section 3.2). Individual transformation rules (and their analysis) is the technical heart of RDF which are presented next.

3.1.3 Transformation rules

Graph rewriting refers to the process of converting a graph to a new graph using rewrite rules which replace a sub-graph by a new sub-graph. An RDF transformation rule tr is a graph rewrite rule of the form

$$tr : lhs \Rrightarrow rhs$$

which selects a sub-graph matching the pattern *lhs* from the current dataflow graph, and replacing it by the graph *rhs*. We use the set-theoretic approach of [46] to graph rewriting: the terms *lhs* and *rhs* can be seen as non empty sets of edges. Actors names, types, and rates in *lhs* are either specified by constant values or possibly with pattern *variables* matching names, types and 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 a given dataflow graph G if there is a substitution σ mapping name (resp. types, rates) variables to actual names (resp. types, rates) in G such that the set of edges of $\sigma(lhs)$ belongs to G: *i.e.,* $\sigma(lhs) \subseteq G$. If *lhs* has no variables, the substitution σ is empty and the *lhs* is to be found as a sub-graph of G. The rule $tr : lhs \Rrightarrow rhs$ 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*, r_1, r_2, ... the pattern variables matching *rates*, and p_1, p_2, ... the pattern variables matching *ports*. For instance, A_x matches any actor of type A, and β_y matches any actor.

As an example, consider the transformation rule tr_1 depicted in Figure 3.2 and the dataflow graph in Figure 3.3. In the transformation rule tr_1, an actor of type B is replaced by an actor of type E.

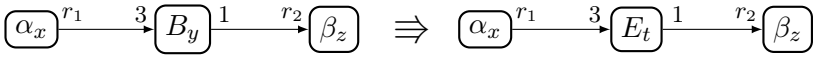


Figure 3.2: The transformation rule tr_1.

Figure 3.3: The dataflow graph G_1.

The terms α_x and β_x match any actor of any type, whereas the term B_y matches any actor of type B. When applied to the graph of Figure 3.3, the rule matches the subgraph

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

and yields the substitution

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

As a consequence, the rule tr_1 replaces the actor B_1 by a new actor E_1. It transforms the graph of Figure 3.3 into the graph of Figure 3.4.

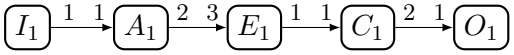


Figure 3.4: The resulting graph $tr_1(G_1)$

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 indexes $\bullet 1$, $\bullet 2$, ... or port variables $\bullet p_1$, $\bullet p_2$, ... in the transformation rules.

Static conditions

In a given transformation rule, the numbers of incoming and outgoing ports and the rates of all incoming and outgoing ports of each actor must be consistent with its type. Actors occurring in the *lhs* and *rhs* must have the same number of edges in both parts (condition (**C1**)). Actors occurring only in *lhs* or *rhs* must be fully linked: they must have explicit types and all their ports connected (conditions (**C2**) and (**C3**)). These 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, α_x and β_y 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, the type of removed actors must 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 incoming and one outgoing edge.

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

  In tr_1, E_t represents a new actor with an explicit type E. It must be verified that the type E have only one incoming and one outgoing edges.

Additional constraints are needed to be checked in order to preserve connectivity, consistency and liveness of the graphs (see section 3.2).

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

$$\underbrace{X \cup \sigma(lhs)}_{G} \quad \Rrightarrow \quad \underbrace{X \cup \sigma(rhs)}_{G' = tr(G)} \tag{3.1}$$

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 $\sigma(lhs)$ is replaced by the sub-graph $\sigma(rhs)$. The context X (the graph or set of edges surrounding the matched part) remains unchanged.

Initial tokens raise semantic issues. For instance, if a transformation has an *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 edges with initial tokens but impose that transformations do not manipulate them. In other words, an edge with initial tokens can neither be matched nor created.

An example of an RDF application is depicted in Figure 3.5. It consists of an initial dataflow graph with three actors A_1, B_1, and C_1 and two transformation programs each being a single transformations rule: tr_{add} and tr_{rem}. The first transformation rule adds a new instance of an actor of type B between two actors of types B and C and the second transformation rule removes an actor of type B connecting two actors of types B and C. When the condition $cond_1$ (resp. $cond_2$) holds, the transformation rule tr_{add} (resp. tr_{rem}) is applied. The conditions are left unspecified here. This example shows how an unbounded number of graphs can be created using RDF, because the rule tr_{add} can be applied an arbitrary number of times.

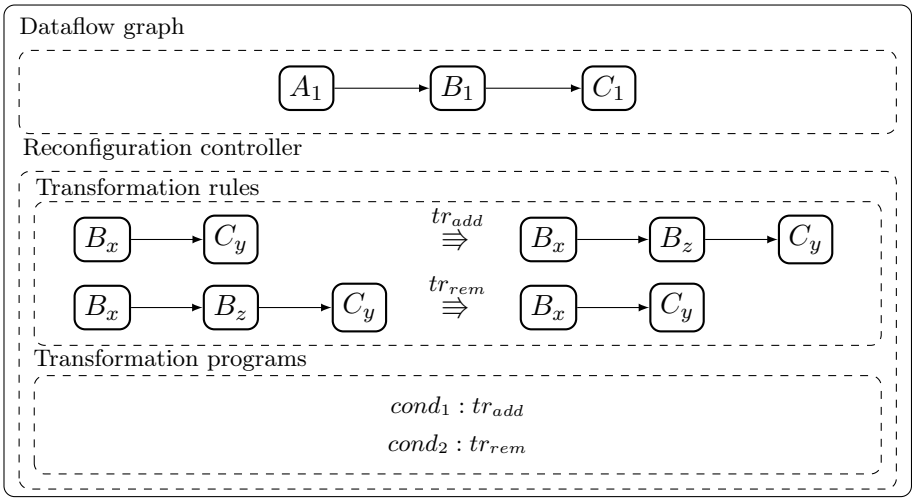


Figure 3.5: An example of an RDF application

3.1.4 Variable arity actors

RDF transformation rules may add and remove actors and sometimes the number of input/output of actors (i.e., their arity) may need to be changed dynamically. For instance, it might be convenient to have actors with 1, 2, or 3 input ports depending on the configuration in which it is used. Different types of variable arity actors can be used depending on the application. We first discuss two kinds of applications requiring two different types of actors

with variable arity and we present those actor types. Then, we study one application on parallel matrix multiplication requiring both kinds of variable arity actors.

Actor Z

Consider an actor Sum which (1) receives n integer tokens x_1, ..., x_n each one from its n input ports, (2) computes the sum of those integer values, and (3) sends the computed sum as an integer token on its single output port. If during the execution of the application, the number of integers to be summed changes, we need a variable arity actor.

The actor Sum which may have a parametric number of input ports, could be emulated by an actor Sum^p with parametric rate p (such parametric actors can be found in PSDF). The actor Sum^p has one single input port with parametric rate p and one single output port. At each execution, it reads p integers from its single input port, computes the sum of those p integers, and writes the result on its output port. The value of the parameter p of actor Sum^p corresponds to the input arity of the actor Sum.

Consider another example: An image processing actor F_1 receives an image, processes it and sends it to another actor F_2. If during the execution, we also need to display the output of the actor F_1, using an actor F_3, we apply a transformation rule in order to connect also F_1 to F_3. The actor F_1 now needs to duplicate its processed image on both of its output ports.

The number of inputs of actor Sum and outputs of actor F changes dynamically. Both of these examples can be implemented by variable arity actors. We can also imagine a Sum actor which needs to duplicate its output tokens on several ports. Therefore, an actor can have both inputs and outputs of variable arity.

Actor Z in Figure 3.6 is a generic variable arity actor with variable numbers of n inputs and m outputs. The rate of each input port is the constant value k and the rate of each output port is the constant value l. The semantic of the execution of actor Z is similar to standard SDF actors. At each firing, it consumes k tokens from each of its input ports, performs a computation, and finally produces l tokens on each of its output ports.

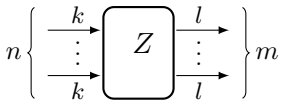


Figure 3.6: Generic variable arity actor Z

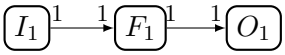
Each actor of type Z can perform a different computation. In general, its processing can be expressed as a function taking a list of size n and producing a list of size m at each of its firing. For example, the actor Sum is a function which consumes a list of n inputs and produces a list of size 1.

When a transformation adds or removes edges of a variable arity actor of type Z, the number of its ports are updated. Moreover, new ports are added at the end of the list of ports, while removing the i^{th} port makes the $(i+1)^{th}$ port become the i^{th} and so on.

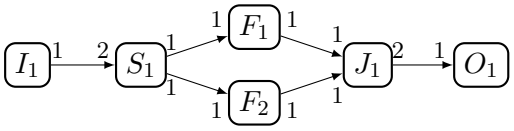
The actor Z is similar to normal SDF actors. The actor Sum with 1, 2, ..., N inputs can be seen as N different actors $Sum1$, $Sum2$, ..., $SumN$. Therefore, introducing actor Z does not change the static analyses of RDF.

Actor X

Consider the following dataflow graph where the filter F_1 is applied on images produced by the actor I_1.



If the resolution of produced images increases dynamically, it might be needed to add a filter F_2 in parallel to F_1 and to change the graph into



where the split actor S_1 reads two image blocks and distributes them to the two filters F_1 and F_2 and where the join actor J_1 reads two image blocks and passes them to O_1. Provided enough computing resources, the actors F_1 and F_2 can be fired in parallel and the throughput is improved. If a third parallel level of computation is needed, one would have to introduce a new filter between the split and join actors to distribute and gather the three images.

To allow such transformation rules an arbitrary number of times, RDF provides a special actor type X with *variable arity* (see Figure 3.7). The consumption rate on each of its incoming edges is q and the production rate on each of its outgoing edges is p where p and q are two unique *parameters*. Unlike in PSDF, these parameters cannot be modified by an actor. The value of the parameter p is always equal to the number of input ports of an actor of type X and the value of the parameter q is equal to the number of its output ports. At each firing, an actor of type X consumes $p \cdot q$ tokens and distributes them on its q outputs. It does not perform any computation.

Two special cases of actor X with variable arity are depicted in Figure 3.8. The split actor of type S has a single input and a variable number of outputs. The rate of its incoming edge is q. In other words, S is an actor X in which $p = 1$. It consumes q tokens from its single input and distributes them on its q outputs. The join actor of type J is symmetric, having a variable number

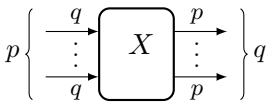


Figure 3.7: Actors X with variable arity

of inputs and a single output. The rate on its single output is p and the number of its input edges is equal to the value of p. So J is a special case of actor X in which $q = 1$. It consumes 1 token from each of its p inputs and gathers them on its single output of rate p. [1]

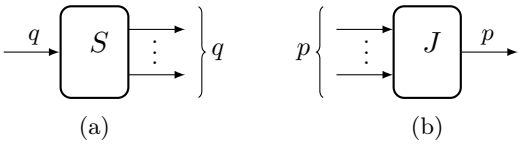


Figure 3.8: Actors with variable arity: (a) split and (b) join.

A variable arity X_k actor is associated with the unique parameters i_k (representing input rates or number of output ports) and o_k (representing output rates or number of input ports). For split actor S_k and join actor J_k, we note the corresponding parameters s_k and j_k respectively. Solving balance equations is performed according to those parameters as in parametric variants of SDF [10]. In this way, consistency is checked for all possible values of parameters.

When a transformation adds or removes edges of a variable arity actor X_k:

- the number of ports $nbin(X_k)$ and $nbout(X_k)$ are updated;
- the values of the parametric rates are changed so that $nbout(X_k) = i_k$ and $nbin(X_k) = o_k$;
- the functions *finr* and *foutr* are updated such that:
  - $\forall 1 \leq \ell \leq i_k, finr(X_k)(\ell) = o_k$,
  - $\forall 1 \leq \ell \leq o_k, foutr(X_k)(\ell) = i_k$.

  Note that 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 ℓ^{th} port makes the $(\ell+1)^{th}$ port become the ℓ^{th} and so on.

[1] The notion of split and join are also defined in StreamIt [53], which is a high-level language for representing streaming applications. Split and join actors are used there to specify independent parallel streams of data. StreamIt is based on message passing MoCs and not SDF.

To allow these updates, 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. According to condition C2, suppressing a variable arity actor would require to select all its edges whose number 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. Variable arity actors therefore should be anticipated in the initial graph. Moreover, transformations rules must ensure that variable arity actors remain fully linked, that is, they have at least one incoming and one outgoing edge. Hence, we have the following additional conditions:

- (**C4**) Variable arity actors cannot be suppressed nor be created by transformation rules.
- (**C5**) If a transformation rule removes an incoming (resp. outgoing) edge from a variable arity actor, then that actor must occur in the *rhs* with at least one incoming (resp. outgoing) edge.

The previous example can be dealt with by using the initial graph and transformation rule of Figure 3.9 which add a new parallel level of computation (here with a new instance F_y of the filter of type F) to process the incoming data. A controller may apply that transformation rule when, for instance, the throughput drops below a certain threshold. The reverse transformation could also be added to free computing resources when possible or needed.

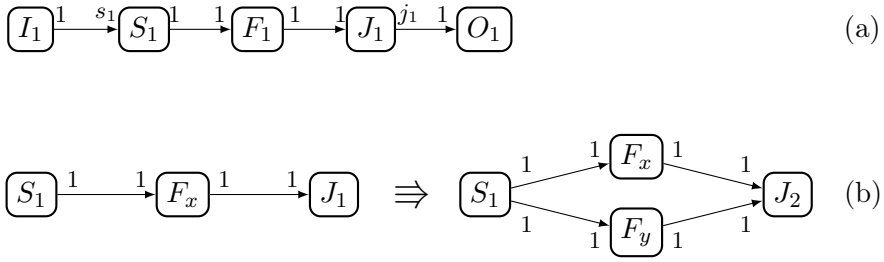


Figure 3.9: Adaptive image filtering: (a) initial dataflow graph and (b) transformation rule

Note that the iteration does not change between the initial graph and the graph after one application of the rule. It remains $(I_1^{s_1}, S_1, F_1, J_1, O_1^{j_1})$, but since the values of the parameters s_1 and j_1 change, the actual number of firings of untouched actors I_1 and O_1 changes.

The initial graph of the Figure 3.10 is inconsistent since it requires that the parameters denoting the number of outgoing (resp. incoming) edges of S_1 (resp. J_1) be equal, *i.e.,* $s_1 = j_1$. Even though that condition is satisfied in the initial graph, it might be invalidated by some transformation rule (*e.g.,* by adding an edge between S_1 and a new sink actor).

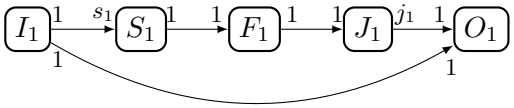


Figure 3.10: An inconsistent cyclic graph

A simple extension to alleviate this problem would be to allow consistency checking to produce such additional constraints and to statically check that all the transformations respect these constraints (see section 3.3.2).

Although it is possible for normal actors to specify their ports in a pattern, for actors with variable arity, we do not know their current number of ports. In any pattern involving variable arity actors, we can always use $\bullet 1$ to denote the first port because it always exists, and we introduce the notation $\bullet last$ to denote the last port of a variable arity actor. These notations allow, for instance, specific edges to be removed from variable arity actors.

Parallel matrix multiplication

We illustrate the use of both types of variable arity actors in a concrete example. Multiplication of matrices is an operation which can be performed in parallel. This is because when computing $C = A \times B$, the element $c_{i,j}$ of the matrix C is obtained by computing the dot product of i^{th} row of A and j^{th} column of B. Therefore, each element of the matrix C can be computed independently only by having access to one row of A and one column of B. We can implement this mechanism using a dataflow graph in different ways. One way is to dedicate one actor for computing one row of C. For computing one row of C, each actor needs one row of A and the whole matrix B.

The parallel matrix multiplication application continuously receives two matrices $A_{N\times K}$ and $B_{K\times M}$ and computes the resulting matrix $C_{N\times M}$. The application consists of N actors each computing one of the rows of C. Assume that during the execution of the application, the number of rows of A and columns of B may change. As a result, the number of actors for computing C needs to be changed and the two types of variable arity actors presented before are useful.

The dataflow graph of this application is G_{pm} (Figure 3.11) in which all rates which are not indicated are equal to 1. The parallel matrix multiplication application consists of five types of actors : A, B, Dup, S, $Multiply$, J, and C. At each firing, the A_1 actor sends tokens each containing a row

of the matrix A. The actor B_1 sends the matrix B in a single token at each iteration. The actor S_1 receives s_1 rows of the matrix A (here $s_1 = 4$) and sends each row on one of its outputs. The actor Dup_1 duplicates its input on all its output ports. Each actor $Multiply$ receives a token containing one row of the matrix A on its first input and a token containing the whole matrix B on its second input. Then, it computes one row of the matrix C and sends this row as a single token. The actor J receives j_1 rows of matrix C (here $j_1 = 4$) and serializes them on its output as j_1 tokens. The actor C_1 receives the rows of the matrix C, reconstructs the whole matrix C and displays the result. In this example, Dup_1 is a variable arity actor of kind Z, and the two actors S_1 and J_1 are both variations of actor X.

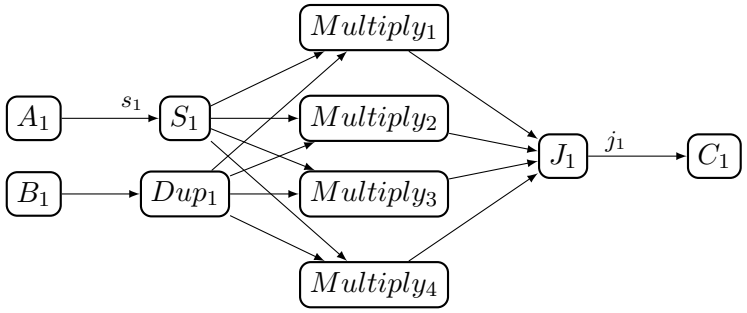


Figure 3.11: The parallel matrix multiplication dataflow graph G_{pm}

In the dataflow graph G_{pm} of Figure 3.11, the matrix A has 4 rows. But during the execution of the graph, the dimensions of the matrix A may change and therefore there is a need for reconfiguration. Suppose that the number of rows of A changes from 4 to 5. In this case one more $Multiply$ actor is needed. The transformation rule tr_{pm} in Figure 3.12 shows how one such actor can be added.

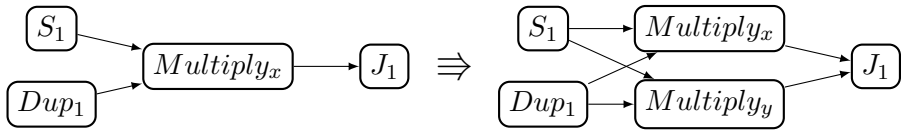


Figure 3.12: The transformation rule tr_{pm}

3.2 Static Analyses

Improving the expressivity and dynamicity of SDF should not come at the price of losing consistency and liveness analyses. We have to guarantee that the transformation rules preserve these properties on the transformed graphs. We also need to ensure that the transformation rules preserve the connectivity of the graphs so that the graphs remain well-formed. A simple approach

would be to verify dynamically whether a transformation rule yields a consistent, live, and connected graph. The disadvantage of this approach is that the user cannot know which transformation rules would fail. Moreover, there would be a cost of trying to apply transformation rules and verifying resulting graphs. We follow another approach which involves verifying *statically* whether a transformation rule always produces a consistent, live, and connected graph.

We study here how connectivity, consistency, and liveness can be analyzed and guaranteed statically for RDF applications. It is sufficient to ensure that:

- these three properties hold for the initial graph (SDF static analyses can be reused for that matter);
- assuming the considered properties hold for a given graph G, for each individual transformation rule tr, the properties still hold for the transformed graph $tr(G)$.

An RDF transformation program is said to be valid if all its transformation rules preserve the three properties. 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 properties.

3.2.1 Well-formedness

We establish a sufficient condition on a given transformation rule such that a well-formed graph remains so after applying that rule. We have to show that actors remain fully-linked, edges remain valid and the graph remains connected.

First, we state a condition to be respected by a given transformation rule, so that by applying it to a connected graph, the resulting graph remains connected. Afterwards, as a corollary, we show that a given transformation rule respecting the static conditions and the condition for connectivity always results in a well-formed graph if applied to a well-formed graph.

RDF graphs are always connected by definition, that is, there is an undirected path between every pair of vertices. A 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 a 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 lhs may match disconnected sub-graphs.

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

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

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

For the proof of the theorem, we recall the following facts and notations:

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

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

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

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

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

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

Actors x and y belong to X and therefore to G. Since G is a connected graph we have $x \xleftrightarrow[G]{+} y$. Recall that an actor belonging to lhs but not to rhs is removed from the graph. Therefore neither x nor y belong to $\sigma(lhs)$. The

undirected path between x and y in G must start and finish with an edge in X, meaning that it has one of the the following forms:

$$x \overset{+}{\underset{X}{\longleftrightarrow}} y$$
$$x \overset{+}{\underset{X}{\longleftrightarrow}} x_1 \overset{+}{\underset{\sigma(lhs)}{\longleftrightarrow}} x_2 \overset{+}{\underset{X}{\longleftrightarrow}} \ldots \overset{+}{\underset{\sigma(lhs)}{\longleftrightarrow}} x_n \overset{+}{\underset{X}{\longleftrightarrow}} y \qquad \text{with } n \geq 1$$

For the first path, since $x \overset{+}{\underset{X}{\longleftrightarrow}} y$, then by definition $x \overset{+}{\underset{tr(G)}{\longleftrightarrow}} y$.

For the second path, since $x_1, \ldots, x_n$ belong to X *and* belong to $\sigma(lhs)$, they also belong to $\sigma(rhs)$. Indeed, recall that, by definition of tr, actors belonging to the edges of X, cannot be suppressed by tr (Condition (C2)).

By condition (C^{rhs}), we have $x_1 \overset{+}{\underset{\sigma(rhs)}{\longleftrightarrow}} x_n$ and, edges in X being untouched by tr, we have $x \overset{+}{\underset{X}{\longleftrightarrow}} x_1 \overset{+}{\underset{\sigma(rhs)}{\longleftrightarrow}} x_n \overset{+}{\underset{X}{\longleftrightarrow}} y$. We therefore conclude that $x \overset{+}{\underset{tr(G)}{\longleftrightarrow}} y$. If $n = 1$, then $x \overset{+}{\underset{X}{\longleftrightarrow}} x_1 \overset{+}{\underset{X}{\longleftrightarrow}} y$, therefore $x \overset{+}{\underset{X}{\longleftrightarrow}} y$, and by definition $x \overset{+}{\underset{tr(G)}{\longleftrightarrow}} y$.

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

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

Sub-Case (C_1): $x \in \sigma(lhs)$.

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

$$x \overset{+}{\underset{X}{\longleftrightarrow}} y$$
$$x \overset{+}{\underset{X}{\longleftrightarrow}} x_1 \overset{+}{\underset{\sigma(lhs)}{\longleftrightarrow}} x_2 \overset{+}{\underset{X}{\longleftrightarrow}} \ldots \overset{+}{\underset{\sigma(lhs)}{\longleftrightarrow}} x_n \overset{+}{\underset{X}{\longleftrightarrow}} y \qquad \text{with } n \geq 1$$
$$x \overset{+}{\underset{\sigma(lhs)}{\longleftrightarrow}} x_1 \overset{+}{\underset{X}{\longleftrightarrow}} x_2 \overset{+}{\underset{\sigma(lhs)}{\longleftrightarrow}} \ldots \overset{+}{\underset{\sigma(lhs)}{\longleftrightarrow}} x_n \overset{+}{\underset{X}{\longleftrightarrow}} y \qquad \text{with } n \geq 1$$

For the first path, since $x \overset{+}{\underset{X}{\longleftrightarrow}} y$, then by definition $x \overset{+}{\underset{tr(G)}{\longleftrightarrow}} y$.

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

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

Sub-Case (C_2): $x \notin \sigma(lhs)$.

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

As a consequence, by condition (C^{rhs}) there is a path $x \overset{+}{\underset{\sigma(rhs)}{\longleftrightarrow}} x_i$. We can use the same reasoning as in Sub-Case $\mathbf{(C_1)}$ to show that there is a path $x_i \overset{+}{\underset{tr(G)}{\longleftrightarrow}} y$. By transitivity, we therefore conclude that $x \overset{+}{\underset{tr(G)}{\longleftrightarrow}} y$. □

A well-formed dataflow graph remains so after applying a transformation rule respecting the static conditions and the condition for connectivity, that is, all actors remain fully linked, all edges remain valid, and the graph remains connected. Formally :

Corollary 1. *Let G be a well-formed graph and tr : lhs ⇛ rhs be a transformation rule satisfying the conditions (C1), (C2), (C3), (C5) and (C^{rhs}), then $tr(G)$ is a well-formed graph.*

Proof. Condition (C1) ensures that the remaining actors (those in intersection of *lhs* and *rhs*) keep the same ports connected, so they remain fully linked. Condition (C3) ensures that newly created actors are fully linked. Variable arity actors are fully linked if they have at least one incoming and outgoing edge, which is guaranteed by condition (C5). The actors not present in the transformation rule remain untouched in the graph. Therefore, all actors in the transformed graph remain fully linked.

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

Given a connected graph and a rule respecting the condition (C^{rhs}), the resulting graph after applying the rule is connected.

Putting all facts together, when a transformation rule satisfying the conditions (C1), (C2), (C3), (C5) and (C^{rhs}) is applied to a well-formed graph, the resulting graph is well-formed. □

Clearly, the transformation tr_1 in Figure 3.2 preserves connectivity, but the following one

is invalid. Its right-hand term is not connected. Applying this transformation to G_1 (Figure 3.3) would produce two disconnected graphs.

3.2.2 Consistency

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

For each transformation rule $tr : lhs \Rrightarrow rhs$, we check that both graphs *lhs* and *rhs* are consistent and we compute the (possibly symbolic) smallest non-null solutions of their actors. Actors occurring both in *lhs* and *rhs* must have the same solution. New actors (*i.e.,* occurring only in *rhs*) only need to have a non-null solution, which is implied by the condition that the *rhs* is 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*

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

then $tr(G)$ is consistent.

Note that we write $sol_A(x)$ to denote the *minimal* solution of actor x in the system of equations corresponding to the graph (or pattern) A. If A is a pure SDF graph, this solution is an integer; if A has pattern variables matching rates, the solution can also be computed and is, in general, symbolic. If a graph (or pattern) has actors of variable arity, 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 [22].

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, that is, $G = G_1 \cup G_2$ 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 RDF graph partitioned into G_1 and G_2. We have:*

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

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

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

The condition $sol_{lhs}(x) = sol_{rhs}(x)$ means that the common minimal symbolic solutions of the system of balance equations 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 $sol_{\sigma(lhs)}(x)$ (resp. $sol_{\sigma(rhs)}(x)$) for 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 (m_1, m_2) be the minimal (coprime) pair of (k_1, k_2) where $m_1 = \frac{k_1}{\gcd(k_1,k_2)}$ and $m_2 = \frac{k_2}{\gcd(k_1,k_2)}$. We thus have:

$$\forall x \in X,\ sol_G(x) = m_1 sol_X(x) \quad \text{and} \quad \forall x \in \sigma(lhs),\ sol_G(x) = m_2 sol_{\sigma(lhs)}(x)$$

Condition (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 m_1 and m_2 can be used to make the solutions equal. As a result, for any shared actor between X and $\sigma(rhs)$, we have:

$$m_1 sol_X(x) = m_2 sol_{\sigma(rhs)}(x)$$

and, by Lemma 1, the graph $tr(G)$ is consistent. Furthermore, since m_1 and m_2 are coprime, they correspond to the minimal solutions of $tr(G)$:

$$\begin{aligned} &\forall x \in X,\ sol_{tr(G)}(x) = m_1 sol_X(x) \\ &\text{and}\ \ \forall x \in \sigma(rhs),\ sol_{tr(G)}(x) = m_2 sol_{\sigma(rhs)}(x) \end{aligned}$$

The proof holds for variable arity actors. The condition and the previous reasoning deal with symbolic solutions which can also accommodate parameters of X actors. An actor Z of n inputs and m outputs can be seen as $m \cdot n$ different configurations and in each configuration, the actor Z is a normal SDF actors, so the reasoning remains the same.

□

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

Example: The transformation rule tr_1 of Figure 3.2 preserves consistency. Both the *lhs* and *rhs* are consistent and their common actors have the same symbolic solutions. Indeed, the solutions of actors in the *lhs* are

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

and those of actors in *rhs* are:

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

The common actors x and z keep their integer solutions and the fresh actor t has a non-null integer solution.

This rule applied to the graph G_1 yields the consistent graph $tr_1(G_1)$ (Figure 3.4). The actors I_1, A_1, C_1, and O_1 keep their solutions (3, 3, 2, and 4, respectively) and the solutions of the new actor E_1 is 2.

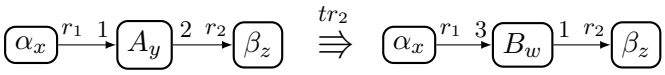


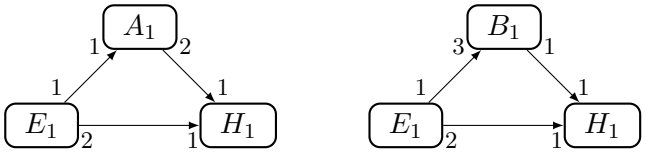
Figure 3.13: The transformation rule tr_2.

On the other hand, the transformation tr_2 in Figure 3.13 is invalid. The reason is that, even though *rhs* is consistent (triviality, because it is acyclic), the solution of actor z changes from $\frac{2.r_1.sol(x)}{r_2}$ to $\frac{r_1.sol(x)}{3.r_2}$. In the particular case of G_1, rule tr_2 produces a consistent graph but all solutions change ($sol_{tr_2(G_1)}(I_1) = 9$, $sol(B_1) = 1$, *etc.*).

In general, such rules can produce inconsistent graphs. For instance, when applied to the graph of Figure 3.14a, tr_2 would produce the inconsistent graph of Figure 3.14b. We have $sol_{G_2}(H_1) = 2$, and yet H_1 has no solution in $tr_2(G_2)$. The reason is that the edge (E_1, H_1) enforces a constraint on the solution of H_1 that cannot be seen in the transformation rule alone.

3.2.3 Liveness

A consistent graph is live if and only if it can be scheduled (see section 2.3.2). We present here sufficient conditions on transformation rules so that they preserve liveness for graphs with single appearance schedules (SAS). Recall also that, as stated in section 3.2.1, transformation rules do not match nor create edges with initial tokens.



(a) G_2 (b) $tr_2(G_2)$

Figure 3.14: Consistent (a) and inconsistent (b) graphs.

For each transformation rule $tr : lhs \Rrightarrow rhs$, it suffices to check that rhs is acyclic and that tr does not add a directed path between common actors of lhs and rhs that did not exist before. These conditions ensures that tr cannot introduce new cycles.

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

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

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

Proof. It is well known that any consistent acyclic SDF graph has a single appearance schedule [1]. We therefore focus on cycles and first prove the following lemma which states that a transformation respecting condition (C^{live}) cannot create new cycles.

Lemma 2. *Let $tr : lhs \Rrightarrow rhs$ a transformation rule satisfying condition (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 depending on whether or not x belongs to X:

1. $x \in X$

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

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

In the first case, all actors in the path are untouched by the transformation rule, therefore $x \xrightarrow[G]{+} x$.

In all other cases, actors $x, x_1, \ldots, x_n$ belong to X: $x \in X$ by hypothesis and each x_i is either the source or destination 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 new 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, condition (C^{live}) enforces that the path $x_i \xrightarrow[\sigma(lhs)]{+} x_j$ exists. Therefore, in each of the above cases, we have $x \xrightarrow[G]{+} x$ by transitivity.

2. $x \notin X$

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

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

   The first case is impossible because it is a self-cycle in *rhs*. Indeed, condition (C^{live}) enforces *rhs* to be live and since tr can only manipulate edges without initial tokens, $\sigma(rhs)$ must be acyclic.

   In the second 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 $\sigma(lhs)$, condition (C^{live}) ensures that $x_n \xrightarrow[\sigma(lhs)]{+} x_1$. Hence we have $x \xrightarrow[G]{+} x$.

□

We now return to the proof of Theorem 3. A consistent SDF graph admits an SAS (or a flat SAS following the terminology of [1]) iff all its 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 $x_0 \longrightarrow x_1 \longrightarrow \ldots x_n \longrightarrow x_0$ 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. □

The transformation rule tr_1 of Figure 3.2 preserves liveness. Indeed, the *rhs* does not introduce new directed paths between actors occurring both in *lhs* and *rhs* (*i.e.,* between A_x, β_y and C_z).

In contrast, the transformation tr_3 in Figure 3.15 is invalid. Actor Y_y is connected to Z_z in the *rhs* but not in the *lhs*. If the only schedule in the initial graph is one where Z_z needs to be fired before Y_y, then rule tr_3 produces a deadlocked (*i.e.,* non live) graph.

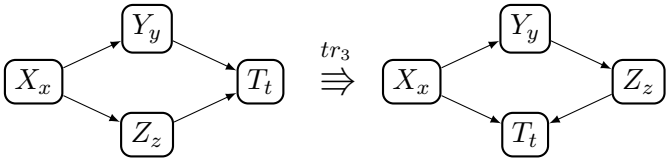
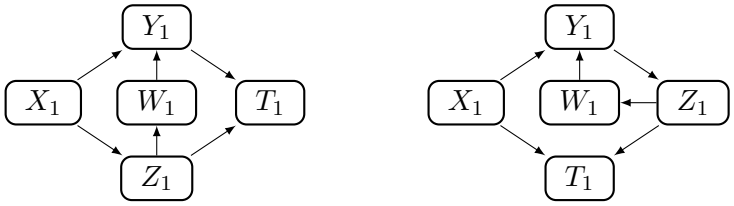


Figure 3.15: The transformation rule tr_3 (all rates are 1).

Such a case is shown in Figure 3.16. The rule tr_3 transforms the live graph G_3 of Figure 3.16a, which admits only the SAS $\{X_1; Z_1; W_1; Y_1; T_1\}$ into the deadlocked graph $tr_3(G_3)$ of Figure 3.16b.



(a) G_3 (b) $tr_3(G_3)$

Figure 3.16: Live (a) and deadlocked (b) graphs (all rates are 1).

3.3 Extensions

In the previous section, we imposed some constraints on the transformation rules in order to ensure that they preserve connectivity, consistency and liveness of dataflow graphs. Moreover, for variable arity actor X, we enforced that the parameters of each actor are unique. In this section, we propose an extension of liveness analysis and an extension of the actor X allowing some of the constrains to be relaxed.

3.3.1 Liveness analysis

In Theorem 3, the sufficient condition for liveness enforces that the initial dataflow graph has a single appearance schedule (SAS). We propose in this

section a sufficient condition to guarantee liveness for dataflow graphs that do not have an SAS, because this is the case of many live cyclic graphs.

In order to introduce that new condition, we define the following functions:

- $Schedules(G)$ which returns the set of all schedules of the graph G.
- $sched_{\mathcal{S}}$ which returns, for each actor $A \in V$, the array of number of disjoint firings of A in the schedule $\mathcal{S}$. For example, if $\mathcal{S}$ is of the form $\dots A^y \dots A^z \dots$ with no other occurrences of A than those two, then $sched_{\mathcal{S}}(A) = [y, z]$. Note that $sol(A) = \sum_{i \in sched_{\mathcal{S}}(A)} i$. Of course, when $\mathcal{S}$ is an SAS, $sched_{\mathcal{S}}$ returns an array with a single integer for all actors.

The RDF graph G_4 in Figure 3.17 is a consistent RDF graph with the iteration (A_1^5, B_1^2, C_1^5). It is also a live RDF graph with the schedule $\mathcal{S} = \{A_1^3; B_1^1; C_1^2; A_1^2; B_1^1; C_1^3\}$. The *sched* function for the schedule $\mathcal{S}$ is

$$sched_{\mathcal{S}} = \{(A_1, [3, 2]), (B_1, [1, 1]), (C_1, [2, 3])\}$$

.

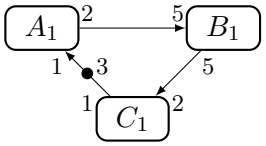


Figure 3.17: RDF graph G_4

Intuitively, the condition for liveness must ensure that all possible schedules in *lhs* have a corresponding schedule in *rhs*. The condition for liveness is formally defined as follows:

$$\begin{aligned} &\forall \mathcal{S}_{lhs} \in Schedules(lhs), \exists \mathcal{S}_{rhs} \in Schedules(rhs) \\ &\text{such that } \forall x \in lhs \cap rhs, sched_{\mathcal{S}_{lhs}}(x) = sched_{\mathcal{S}_{rhs}}(x) \\ &\text{and } \forall y \in rhs - lhs, sched_{\mathcal{S}_{rhs}}(y) \neq \emptyset \end{aligned} \tag{3.2}$$

In other word, for each schedule $\mathcal{S}_{lhs}$ in the *lhs*, there must exist a schedule $\mathcal{S}_{rhs}$ such that for each common actor x between *lhs* and *rhs*, the array of the number of disjoint firings of x in both *lhs* and *rhs* is equal. Moreover, each appearing actor y must be schedulable in $\mathcal{S}_{rhs}$.

Example: The transformation rule tr_4 (Figure 3.19) is applied to the RDF graph G_4. The transformation adds a new actor D_y between actors A_x and B_z. The resulting graph $tr_4(G_4)$ of the Figure 3.18 is obtained, which is consistent and live with the following schedule

$$\{A_1^3; D_1^3; B_1^1; C_1^2; A_1^2; D_1^2; B_1^1; C_1^3\}.$$

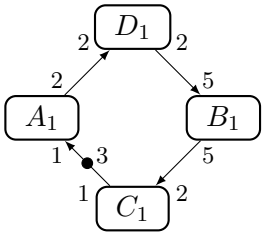


Figure 3.18: Resulting graph $tr_4(G_4)$

The transformation rule preserves consistency since the *rhs* is consistent and the solutions of actors A_x and B_z remain the same ($sol(A_x)$ and $\frac{2 \cdot sol(A_x)}{5}$). The solution for the new added actor D_x is identical to that of A_x, *i.e.,* $sol(A_x)$.

In order to check that tr_4 preserves liveness, for each schedule of *lhs*, we must find a corresponding schedule of *rhs*. The schedules of *lhs* are

$$Schedules(lhs) = \{\mathcal{S}_1 = \{A_x^5; B_z^2\}, \mathcal{S}_2 = \{A_x^3; B_z; A_x^2; B_z\},$$
$$\mathcal{S}_3 = \{A_x^4; B_z; A_x^1; B_z\}\}.$$

Similarly, we can compute all schedules of *rhs*.

Consider the schedule $\mathcal{S}_2$. If actor A_x fires 3 times according to the first element of its *sched* vector, it produces 6 tokens which can be consumed by 3 firings of the new actor D_y. Three firings of D_y will enable actor B_z to fire once according to the first element of its *sched* vector. One token will remain on the edge (D_y, B_z). For the second round, actor A_x fires 2 times according to the second element of its *sched* vector. This will enable D_y to fire 2 times. The 4 tokens produced and 1 token remaining from the previous firing allows B_z to fire another time. In this way, the *sched* vector for actor D_y is calculated as $[3, 2]$.

For all schedules $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ we can find a corresponding schedule in *rhs*, therefore the transformation rule tr_4 is valid.

Figure 3.19: Transformation rule tr_4

On the other hand, if the transformation tr_5 (Figure 3.20) is applied to the RDF graph G_4, the resulting graph $tr_5(G_4)$ (Figure 3.21) is consistent but not live.

Figure 3.20: Transformation rule tr_5

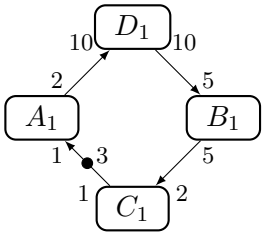


Figure 3.21: Consistent non-live resulting graph $tr_5(G_4)$

In tr_5, the *lhs* has three schedules $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$. Regarding actor A_x we have

$$sched_{\mathcal{S}_1}(A_x) = [5], sched_{\mathcal{S}_2}(A_x) = [3, 2], \text{ and } sched_{\mathcal{S}_3}(A_x) = [4, 1].$$

According to the condition (3.2), for all schedules in the *lhs*, a schedule in *rhs* must exist such that the *sched* function returns the same value for all common actors and such that the *sched* function of new appearing actors is not empty. For the schedule $\mathcal{S}_1$, we can find the schedule $\mathcal{S}_4 = \{A_x^5; D_y^1; B_z^2\}$, where $sched_{\mathcal{S}_4}(D_y) = [1]$. However, for $\mathcal{S}_2$ and $\mathcal{S}_3$, there is no schedule of *rhs*. The only schedule of *rhs* is $\mathcal{S}_4$. Using $\mathcal{S}_2$ and $\mathcal{S}_3$, the actor A_x fires 3 times and 4 times in one iteration respectively, none of which is enough for the actor D_y to start firing.

3.3.2 Variable arity actors

In section 3.1.4, we enforced each parameter associated with variable arity actor X to be unique. Some graphs may require to establish constraints between those parameters. For instance, the dataflow graph of the Figure 3.10 at page 44, requires the constraint $s_1 = j_1$.

We describe a different approach allowing constrains between parameters of variable arity actors.

The initial graph has a number of n actors of variable arity, each having one (in case of actor S and J) or two parameters (in case of actor X). Constraints between those parameters can be obtained from the system of balance equations or stated by the user. For instance $cstr_1 : s_1 = j_1$, indicates that the values of s_1 and j_1 must always be equal. These constraints must be respected by transformation rules. If a transformation rule tr manipulates the parameter p, it must preserve all the constraints associated with p.

For the graph G_5 (Figure 3.22), suppose we have the constraint $cstr_1 : s_1 = j_1$. The transformation rule tr_6 is valid, because it increments the values of both s_1 and j_1 by 1 and preserves the constraint.

On the other hand, if we have another constraint $cstr_2 : s_1 = s_2$, then the tr_6 is no more valid, because it changes the value of s_1, but not s_2.

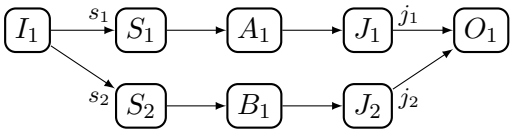


Figure 3.22: The dataflow graph G_5

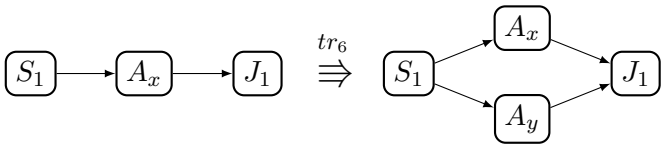


Figure 3.23: Transformation rule tr_6

3.4 Summary

Reconfigurability is one of the criteria on which we can compare models of computation. The reconfigurability refers to the ability of an MoC to deal with changes at execution time. In the previous chapter, we studied a number of existing extensions of SDF found in the literature and we saw how each can handle a limited form of reconfiguration. We distinguished two types of reconfigurations: parameterization and dynamic change of topology.

Although these mechanisms make it possible to change the configuration of the graph dynamically, they all share a disadvantage. The number of configurations they can produce using one instance of an MoC is bounded in theory and small in practice since modeling large graphs with large numbers of configurations requires using many parameters and enumerating explicitly all configurations.

In this chapter, a new MoC was proposed to overcome this disadvantage. The Reconfigurable Dataflow (RDF) MoC is an extension of SDF which proposes to use graph rewriting rules for handling reconfigurations. The RDF MoC uses dataflow graphs similar to SDF graphs except that each actor has a type. Using the notion of type allows creating an unbounded number of instances of a given type. Besides the initial dataflow graph, an RDF application has a controller that defines when and how the graph must be reconfigured. To define "when" the change happens, the controller offers a set of conditions which are boolean expressions (examples are presented in the next chapter). To define "how" the dataflow graph is changed, the controller offers a set of transformation programs. A transformation program is a combination of transformation rules. A transformation rule is a graph rewrite rule made of: a left-hand side pattern graph to determine which sub-graph of the dataflow graph must be replaced by the right-hand side graph. Once such sub-graph is found matching the pattern of the left-hand side, it is replaced by the graph specified in the right-hand side of the rule. In this

way, a large or even unbounded number of configurations can be obtained using even one single transformation rule applied multiple times.

Some transformation rules require the number of input and output edges of an actor to be changed. Variable arity actors are actors allowing this flexibility. RDF introduces two generic types of variable arity actors: actor Z and actor X. These actors have a variable number of inputs (each input edge having the same parametric rate) and a variable number of outputs (each output edge having the same parametric rate). The actor Z simply allows changing the number of inputs and outputs dynamically. The actor X however has this extra property that, at each of its firings, it produces the same number of tokens that it consumes. Two special cases of actor X are called split and join actors. The split actor receives q tokens from its single input and distributes the q tokens each on one of its q outputs. The join actor receives p tokens each one from one of its p inputs and gathers them on its single output. These actors allow distributing the computation along different parallel levels of computation.

We discussed how an RDF application can be analyzed statically in order to ensure that all possible configurations at run-time remain well-formed, bounded, and live. For this purpose, a number of sufficient conditions were proposed. For well-formedness, we must ensure all graphs produced at run-time are connected. The condition to ensure that given a connected graph, a transformation rule produces a connected graph is that the right-hand side of the transformation rule must be a connected graph. To preserve boundedness, the solutions of actors must not be changed by a transformation and new actors must have integer solutions. Moreover, the parametric solutions in the transformation rules must be natural numbers. This ensures that given a consistent graph, the transformation rule produces a consistent graph. Finally to ensure liveness, a transformation rule must not create any new directed cycle. Therefore, given a live graph and a transformation rule, the sufficient condition for the resulting graph to be live is that the right-hand side of the rule is live and for each path in the right-hand side of the rule, the path also exists in the left-hand side. The liveness condition is only for the graph with single appearance schedules.

In the next chapter, we present two performance analyses. We describe how the impact of transformation rules on the latency and throughput can be analyzed.

Chapter 4

Performance Analysis

Performance measurement of dataflow graphs refers to the process of analyzing time-related or memory-related aspects of the execution of dataflow graphs. Two important time-related performance metrics of dataflow graphs are latency and throughput. As defined before, the end-to-end latency of a given iteration of a graph is the largest end time minus the smallest start time among all firings in that iteration. The throughput is the inverse of the period where the period of a given iteration i of a graph is the largest end time of the firing of all actors in i^{th} iteration minus the largest end time of the firings of all actors in the $(i-1)^{th}$ iteration.

In RDF, knowing the impact of the transformation rules on the latency and the throughput of dataflow graphs would be useful. Consider a reconfiguration program applying a transformation rule only if the latency of the resulting graph is within a threshold. If we know the impact of a transformation rule on latency, and there is a deadline to respect, we would avoid applying transformation rules which may violate the deadlines. In this case, the impact of the transformation rule on latency have to be computed statically. Consider another application which applies specific transformation rules that increases the throughput, when it becomes too low. This application has to choose which transformation rule to apply based on their impact on the throughput.

The impact of a transformation rule on a performance metric can be evaluated either dynamically (at run-time) or statically (at compile-time). The dynamic approach is expensive in terms of computation and does not provide any information to the programmer at design time. In order to compute the impact of a transformation rule dynamically, the performance metric of the graph must be computed before and after applying the transformation rule.

In the static approach, the performance metric of the resulting dataflow graph obtained by applying a transformation rule is computed without any knowledge of the current graph and by considering only the rule. Of course, such impact can only be approximated. This is the topic of this chapter,

where we compute statically an upper bound of the impact of applying a transformation rule on the latency and the throughput of any RDF graph.

In this chapter, we obtain formulas for the impact of transformation rules on latency and throughput. These formulas depend on execution times and solutions of actors in the transformation rules. In some cases, the values of all execution times and solutions are known and the formulas give numerical values statically. In other cases, the formulas remain symbolic and can be instantiated dynamically to compute the impact of rules on the performance metrics.

4.1 Analysis of the impact on latency

We propose to compute statically an upper bound of the impact of a transformation rule on the latency of an arbitrary graph. Each rule must be analyzed once at compile time. This information can be used by the designer to modify or add rules. It can also be used during the execution by the controller.

We consider a class of widely used transformation rules. For a given transformation rule $tr : lhs \Rrightarrow rhs$ belonging to that class, and any arbitrary dataflow graph G with the latency $\mathcal{L}_G$, we compute an upper bound of the change of latency. That is, if the resulting graph is $tr(G)$ with the latency $\mathcal{L}_{tr(G)}$ and the rule increases the latency, an upper bound for $\Delta\mathcal{L} = \mathcal{L}_{tr(G)} - \mathcal{L}_G$ is computed. Otherwise, when the latency decreases, the upper bound is computed for $\Delta\mathcal{L} = \mathcal{L}_G - \mathcal{L}_{tr(G)}$.

We first study two examples in which we compute the schedules of a graph before and after applying a transformation rule inserting an actor between two other actors. Then, we give a formula to compute an upper bound on the impact of this insert transformation rule on the latency. The formula is then generalized for all classes of transformation rules which insert and remove actors between two actors with a few constraints. Finally, at the end of this section, we present the application of the analyses on some examples.

For all these calculations, we suppose that the scheduling policy is ASAP and that there are enough processors in order to assign one processor to each actor. The analysis depends on the execution time and solutions of actors. If all execution times and solutions are known, we can compute a numerical upper bound statically. Otherwise, the impact of transformation rule on latency is a symbolic formula whose parameters must be retrieved dynamically. In such cases, we cannot have a numerical information about the impact of the rule on the latency before starting the execution.

In the following examples, transformation rules of the form tr_{insert} (Figure 4.1), where $p, q, r \in \mathbb{N}$ and $(p \cdot q) \bmod r = 0$, insert one actor between two existing actors in a graph. On both sides of the transformation rule, we have $sol(x) = q$ and $sol(y) = p$ and the new actor z of type γ is such that

$sol(z) = (p \cdot q)/r$.

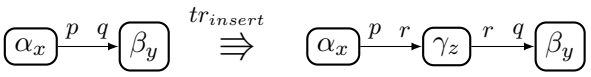


Figure 4.1: Generic rule tr_{insert} which inserts a new actor γ_z between α_x and β_y.

Example 1: Consider an instance of the transformation rule tr_{insert}, in which $p = 2$, $q = 3$, and $r = 6$ (Figure 4.2) and consider the graph G_1 of Figure 4.3. The execution times of the actors in the graph are $t(I) = 1$, $t(O) = 1$, $t(A) = 5$, and $t(B) = 3$ and their solutions are $sol(I_1) = sol(A_1) = 3$ and $sol(B_1) = sol(O_1) = 2$. The iteration of the graph G_1 is $(I_1^3, A_1^3, B_1^2, O_1^2)$ and its ASAP schedule is shown in Figure 4.4. The latency of the first iteration of G_1 is 20, $\mathcal{L}_{G_1} = 20$.

Figure 4.2: Transformation rule tr_1

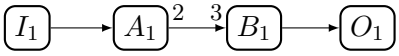


Figure 4.3: Graph G_1

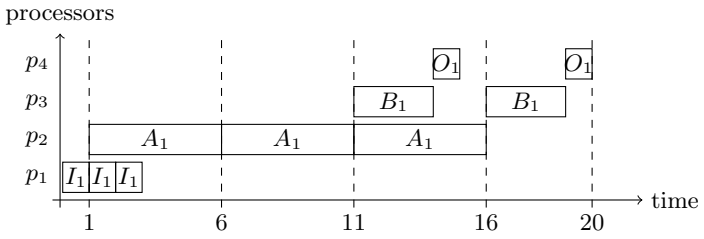


Figure 4.4: ASAP schedule for the graph G_1.

Applying the transformation rule tr_1 on the graph G_1 yields the graph of Figure 4.5. Suppose that the execution time of the actor C_1 is 4. The ASAP schedule of the graph $tr_1(G_1)$ is shown in the Figure 4.6, where the solution of the new actor C_1 is 1. The latency of the first iteration of $tr_1(G_1)$ is 27 $\mathcal{L}_{tr_1(G_1)} = 27$. Here, the latency is incremented by 7 ($\Delta\mathcal{L} = 7$).

Figure 4.5: The resulting graph $tr_1(G_1)$

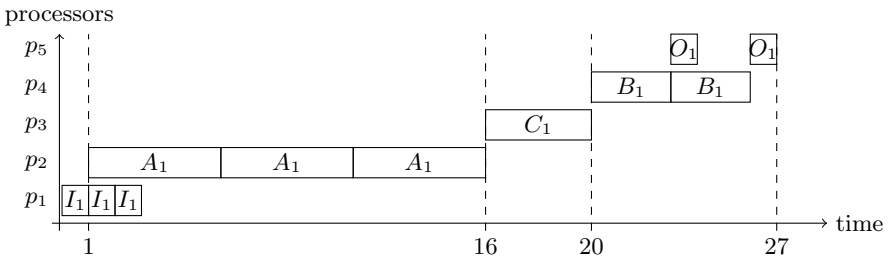


Figure 4.6: ASAP schedule for the graph $tr_1(G_1)$.

Example 2: Suppose now that the dataflow graph is not a chain, but rather the graph G_2 shown in Figure 4.7. The execution time of the actors in the graph are $t(S) = 1$, $t(J) = 1$, $t(A) = 3$, $t(B) = 2$, $t(D) = 4$, and $t(E) = 11$. The ASAP schedule for the graph G_2 is shown in the Figure 4.8. The latency of the first iteration is 24.

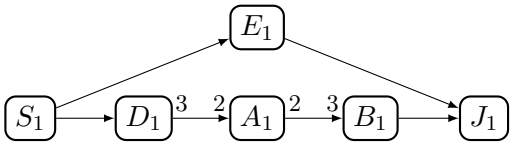


Figure 4.7: Graph G_2

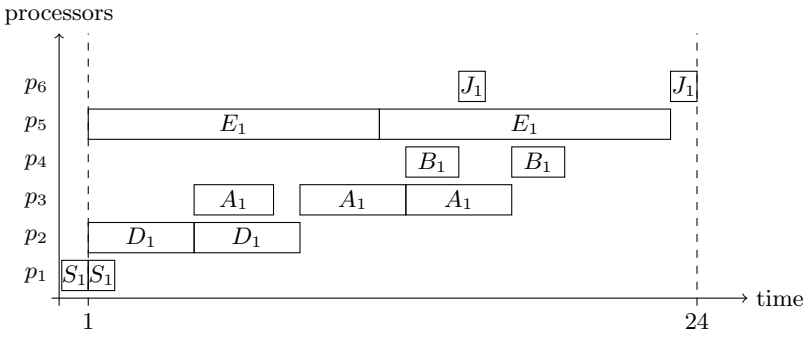


Figure 4.8: The schedule of the graph G_2.

The execution time of the actor C_1 is 4 as before. The ASAP schedule of the resulting graph $tr_1(G_2)$ is as shown in Figure 4.9.

We notice that in this example the latency is not affected by the transformation rule, because the critical path (S_1, E_1, J_1) is not impacted by the transformation rule.

In these two examples, we see that it is not possible to compute the *exact impact* of a transformation rule on the latency without knowing the dataflow graph to which it is dynamically applied. Since this information

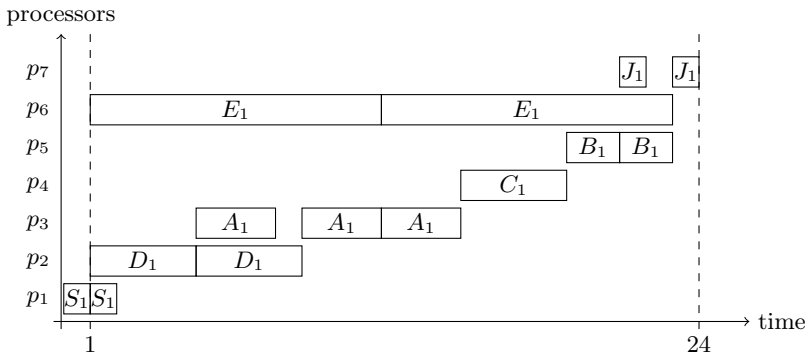


Figure 4.9: The schedule of the graph $tr_1(G_2)$.

is not available at compile-time, the best we can hope for is to compute an *upper bound of the impact* of the transformation rule on latency.

In the following, we study the impact of transformation rules on the latency of arbitrary graphs. In proposition 1, we consider a simple transformation rule that inserts a new actor between two connected actors. We compute an upper bound on the impact of the rule on the latency of any graph. In proposition 2, we generalize the previous result, by considering a large class of transformation rules where the *lhs* and *rhs* are acyclic graphs with a single source and a single sink actor.

In all examples and proofs, we only study the latency of the *first iteration*, which is the first iteration of the prologue and is used as an approximation of the average latency.

We use the following notations:

- $end_G(A)$ is the end time of last firing of a given actor A in an iteration of the graph G.
- $\sigma(x)$ is the actor in the graph matched by the variable x in the transformation rule considered.

Proposition 1 establishes an upper bound on the impact of an insert transformation rule on the latency of an arbitrary dataflow graph. The transformation rule tr_{insert} inserts one actor z between two actors x and y. Intuitively, the impact on latency is computed as follows. We first compute the minimum time at which the part of the dataflow graph matching the left-hand side of the rule would terminate. Then, we compute the maximum time at which the part of the dataflow graph matching the right-hand side of the transformation rule would terminate. The difference gives an upper bound on the impact of this rule on the latency of the graph.

Proposition 1. *Let G_1 be a graph with latency $\mathcal{L}_{G_1}$, $tr_{insert} : x \xrightarrow{p\ q} y \Rightarrow x \xrightarrow{p\ r} z \xrightarrow{r\ q} y$ be a valid transformation rule, and $\mathcal{L}_{G_2}$ be the latency of the*

resulting graph $G_2 = tr_{insert}(G_1)$*. The impact of the transformation rule on the latency is* $\Delta\mathcal{L} = \mathcal{L}_{G_2} - \mathcal{L}_{G_1}$ *with*

$$\Delta\mathcal{L} \leq (sol(y) - 1) \cdot t(y) + sol(z) \cdot t(z) \tag{4.1}$$

Proof. Let G_1 be any RDF graph and $G_2 = tr_{insert}(G_1)$. The subgraphs of the dataflow graphs G_1 and G_2 matched respectively by the left-hand side and right-hand side of the transformation rule tr_{insert} both terminate their execution with $\sigma(y)$ (the actor that y matches). In order to compute the upper bound on $\Delta\mathcal{L}$, we first calculate the smallest time instant at which the actor $\sigma(y)$ would terminate in G_1 before applying the transformation rule, that is, $\min_{\forall G_1}(end_{G_1}(\sigma(y)))$. We then calculate the largest time instant at which the actor $\sigma(y)$ would terminate after applying the transformation rule, that is, $\max_{\forall G_2}(end_{G_2}(\sigma(y)))$. Then, the maximum impact of the rule on the latency is

$$\Delta\mathcal{L} \leq \max_{\forall G_2}(end_{G_2}(\sigma(y))) - \min_{\forall G_1}(end_{G_1}(\sigma(y))).$$

First, suppose that the actor $\sigma(y)$ has no other immediate predecessor than $\sigma(x)$ and that the graph is acyclic. We study afterwards how to relax these hypotheses.

Before applying tr_{insert}, once $\sigma(x)$ has finished all its firings, it takes in the best case only one firing of $\sigma(y)$ to complete all the firings of $\sigma(y)$. For instance, when $sol(x) > sol(y)$ and $t(x) > t(y)$, the actor $\sigma(y)$ must be fired only once to reach $end_{G_1}(\sigma(y))$ (this case is illustrated in Figure 4.10-left). Therefore, we will have:

$$\min_{\forall G_1}(end_{G_1}(\sigma(y))) = end_{G_1}(\sigma(x)) + t(y) \tag{4.2}$$

After applying tr_{insert}, once $\sigma(x)$ has finished its last firing in the iteration, there remains at the worst case, $sol(y)$ firings of the actor $\sigma(y)$ and $sol(z)$ firings of the actor $\sigma(z)$ (this case is illustrated in Figure 4.10-right). An example of this case is when $r = p \cdot q$. In this case, the actor $\sigma(x)$ must be executed q times before the only firing of actor $\sigma(z)$. After the execution of actor $\sigma(z)$, the actor $\sigma(y)$ will be executed p times. Because each actor runs on a dedicated core, the execution of the actors $\sigma(y)$ and $\sigma(z)$ cannot be delayed by any other actor. Therefore, we have:

$$\max_{\forall G_2}(end_{G_2}(\sigma(y))) = end_{G_2}(\sigma(x)) + sol(y) \cdot t(y) + sol(z) \cdot t(z) \tag{4.3}$$

Since the graphs G_1 and G_2 are acyclic, the transformation rule does not impact the execution of the graph before $\sigma(x)$ and, $end_{G_1}(\sigma(x)) = end_{G_2}(\sigma(x))$. From equations (4.2) and (4.3), we conclude:

$$\Delta\mathcal{L} \leq (sol(y) - 1) \cdot t(y) + sol(z) \cdot t(z) \tag{4.4}$$

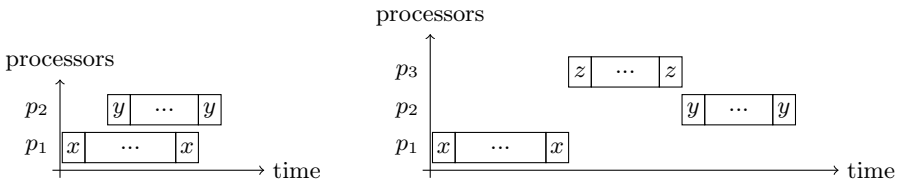


Figure 4.10: The best case ASAP schedule for the graphs G_1 and the worst case ASAP schedule for G_2.

The best case ASAP schedule for the graphs G_1 and G_2 are shown in Figure 4.10.

We now relax the two hypotheses mentioned earlier. Suppose that $\sigma(y)$ has other immediate predecessors in addition to $\sigma(x)$. Among all its predecessors, let m be the one that finishes its last firing later than all the predecessors of $\sigma(y)$. In other word, $m = \arg\max_{u \in pred(\sigma(y))} end_{G_1}(u)$. If $m = \sigma(x)$, the result is the same as what was computed in equation (4.4). Otherwise, the execution of m may only lower $\Delta\mathcal{L}$, and the upper bound of equation (4.4) remains correct (Figure 4.11 illustrates this case).

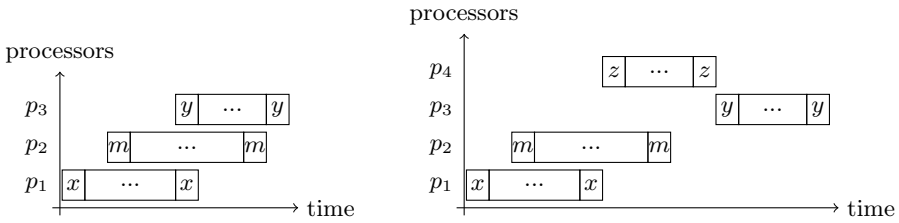


Figure 4.11: The best case form ASAP schedule for the graphs G_1 and the worst case form ASAP schedule for G_2 when there exists an actor $m \neq x$ which finishes its last firing later than all the predecessors of $\sigma(y)$.

Suppose now that the graph G_1 is cyclic. According to the liveness conditions, it must have a single appearance schedule (SAS). The existence of an SAS ensures that each cycle has at least a saturated edge. Recall that a saturated edge is an edge that provides its sink actor with enough tokens so that this actor can complete all its firings in one iteration. To compute the schedule, the saturated edges can be removed from G_1 and the resulting graph can be treated as an acyclic graph. Moreover, the liveness condition also ensures that no cycle is created and that transformation rules do not manipulate edges with initial tokens. Therefore, the graph G_2 has the same saturated edges and also can be treated as an acyclic graph. The bounds on the impact of a transformation rule on latency remain unchanged.

□

Consider the example 1 presented on page 63, where $p = 2$, $q = 3$, and

$r = 6$. In this case $sol(x) = 3$, $sol(y) = 2$, and $sol(z) = 1$. For the execution times, we have $t(x) = 5$, $t(y) = 3$, and $t(z) = 4$. According to proposition 1, we have

$$\Delta\mathcal{L} \leq (sol(y) - 1) \cdot t(y) + sol(z) \cdot t(z) = 4 + 3 = 7$$

If we apply this transformation rule tr_1 to the graph G_1, the upper bound we have found is the exact value of $\Delta\mathcal{L}$. If we choose a different value for r, of course the exact value of the $\Delta\mathcal{L}$ might be less than the upper bound.

Indeed, in example 2 presented on page 64, the actual value of $\Delta\mathcal{L}$ is 0, which is less than the upper bound 7.

We now generalize the previous result, by considering a larger class of transformation rules. In this class, the *lhs* and *rhs* of the rule are acyclic graphs with a single source and a single sink actor.

The use the following definition and notation:

- A *source-sink acyclic (pattern) graph* is a connected acyclic graph having a single source actor and a single sink actor.
- $P(G)$ is the set of simple paths in a source-sink acyclic graph G with source actor x and sink actor y (excluding the source x in each path). Each path in $P(G)$ is a sequence $(c_1, .., c_n, y)$ which contains a set of actors in G such that the edges (x, c_1), ... , (c_{n-1}, c_n), (c_n, y) belong to G and $\forall i, j, i \neq j \implies c_i \neq c_j$.

Proposition 2. *Let G_1 be a graph with latency $\mathcal{L}_{G_1}$, and $tr : H_1 \Rrightarrow H_2$ be a transformation rule, such that:*

- *H_1 and H_2 are two source-sink acyclic graphs, both having the source actor x and the sink actor y;*
- *the latency of H_2 is greater than H_1.*

. Let $\mathcal{L}_{G_2}$ be the latency of the resulting graph $G_2 = tr(G_1)$. The impact of the transformation rule tr on the latency of the graph is $\Delta\mathcal{L} = \mathcal{L}_{G_2} - \mathcal{L}_{G_1}$ with

$$\Delta\mathcal{L} \leq \max_{path \in P(H_2)} \sum_{c \in path} sol(c) \cdot t(c) - \max_{path \in P(H_1)} \sum_{c \in path} t(c) \tag{4.5}$$

Proof. The idea of the proof is similar to the previous proposition. First, we compute the minimum time instant at which the left-hand side the transformation rule would terminate. In order to determine this value, all the paths of the graph of the left-hand side must be compared to each other to find which path terminates last. Then, we compute the maximum time instant at which the right-hand side of the transformation rule would terminate. Similarly, we have to find which path terminates last by comparing all paths of

the graph of the right-hand side. Finally the difference computes the upper bound on the impact of the rule on the latency.

We follow the same reasoning as in proposition 1 to compute

$$\Delta\mathcal{L} \leq \max_{\forall G_2}(end_{G_2}(\sigma(y))) - \min_{\forall G_1}(end_{G_1}(\sigma(y))).$$

As before, we first suppose that all actors in H_1 and H_2 except x are not connected to any actor outside the rule, and that the graph is acyclic. We relax these constraints later.

Before applying the transformation rule, once the actor $\sigma(x)$ has finished firing, we must compute the minimum time it takes for the actor $\sigma(y)$ to complete its firings. In the best case, the time needed to execute each path $(c_1, c_2, ..., c_n, y)$ in H_1 is $t(c_1)+...+t(c_n)+t(y)$. This means that in the best case, once the actor $\sigma(x)$ has finished firing, only one firing remains for the actor c_1, then only one firing of actor c_2, and so on. Among all such paths in H_1, we choose the path for which the summation $\sum_{n\in path} t(n)$ is maximum, because that is the path which finishes executing last and $\sigma(y)$ cannot finish earlier. Therefore, we have:

$$\min_{\forall G_1}(end_{G_1}(\sigma(y))) = end_{G_1}(\sigma(x)) + \max_{path\in P(H_1)} \sum_{c\in path} t(c) \tag{4.6}$$

After applying the transformation rule, once the actor $\sigma(x)$ has finished firing, we must compute the maximum time it takes for the actor $\sigma(y)$ to complete its firings. In the worst case, the time needed to execute each path $(c_1, c_2, ..., c_n, y)$ in H_2 is $sol(c_1)\cdot t(c_1)+...+sol(c_n)\cdot t(c_n)+sol(y)\cdot t(y)$. This means that in the worst case, once the actor $\sigma(x)$ has finished firing, there remains $sol(c_1)$ firings of the actor c_1, and once the actor c_1 has finished, there remains $sol(c_2)$ firing of actor c_2, and so on. In the worst case, the production and consumption rates on the edge between c_i and c_{i+1} is such that c_{i+1} cannot start executing before c_i has completed all its firings. Among all the paths in H_2, we choose the path for which the summation $\sum_{c\in path} sol(c)\cdot t(c)$ is maximum, because that is the path which makes $\sigma(y)$ finishes the last. Therefore, we have:

$$\max_{\forall G_2}(end_{G_2}(\sigma(y))) = end_{G_2}(\sigma(x)) + \max_{path\in P(H_2)} \sum_{c\in path} sol(c)\cdot t(c) \tag{4.7}$$

Since the graphs G_1 and G_2 are acyclic and no actor before the actor x is changed by the transformation rule, therefore $end_{G_1}(\sigma(x)) = end_{G_2}(\sigma(x))$ From equations (4.6) and (4.7), we conclude:

$$\Delta\mathcal{L} \leq \max_{path\in P(H_2)} \sum_{c\in path} sol(c)\cdot t(c) - \max_{path\in P(H_1)} \sum_{c\in path} t(c) \tag{4.8}$$

As in proposition 1, the two previous constraints can be relaxes. After applying the transformation rule, if there is an actor m outside the transformation rule connected to any actor z in H_1 and H_2, which terminates its execution after all the predecessors of the actor z in the rule, then it would reduce $\Delta\mathcal{L}$, but the computed maximum value remains a valid upper bound. Moreover, as discussed in the proof of proposition 1, the upper bound also works for cyclic graphs with an SAS.

□

Example 3: Consider the graph G_3 shown in Figure 4.12. The solutions of actors are $sol(I_1) = sol(S_1) = sol(A_1) = sol(C_1) = 3$ and $sol(B_1) = sol(J_1) = sol(O_1) = 2$ and assume that the execution time of all actors is 1. The ASAP schedule for the graph G_3 is shown in Figure 4.13.

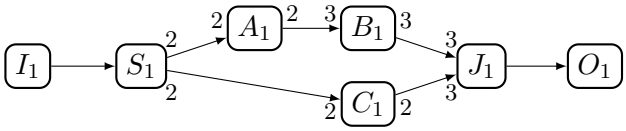


Figure 4.12: Graph G_3

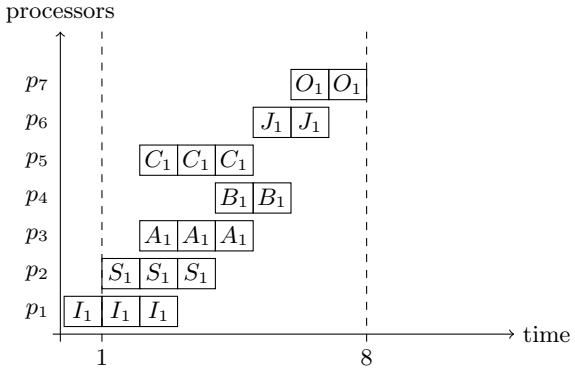


Figure 4.13: The schedule for the graph G_3

The latency of the first iteration is 8. For this schedule, we have:

$$end_{G_3}(J) = end_{G_3}(S_1) + \max_{path \in \{(A_1,B_1,J_1),(C_1,J_1)\}} \sum_{c \in path} t(c)$$
$$= end_{G_3}(S_1) + max(1+1+1, 1+1) = end_{G_3}(S_1) + 3$$

If we apply the transformation rule tr_2 shown in Figure 4.14 on G_3, we obtain the graph shown in Figure 4.15. The solutions of the new actors are $sol(D_1) = sol(E_1) = sol(F_1) = sol(K_1) = 1$.

The schedule of the graph $tr_2(G_3)$ is shown in Figure 4.16. The latency of the first iteration is 15, therefore $\mathcal{L}_{tr_2(G_3)} - \mathcal{L}_{G_3} = 15 - 8 = 7$. For this

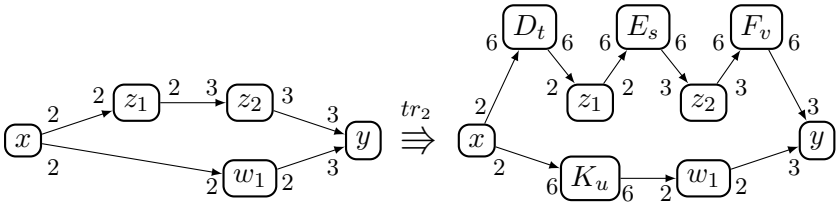


Figure 4.14: The transformation rule tr_2

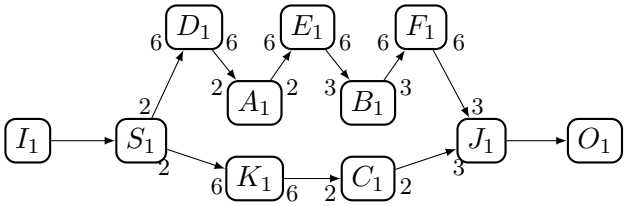


Figure 4.15: Resulting graph $tr_2(G_3)$

schedule, we have:

$$\begin{aligned} end_{tr_2(G_3)}(J_1) &= end_{tr_2(G_3)}(S_1) \\ &\quad + \max_{path \in \{(D_1,A_1,E_1,B_1,F_1,J_1),(K_1,C_1,J_1)\}} \sum_{c \in path} sol(c) \cdot t(c) \\ &= end_{tr_2(G_3)}(S_1) + max(1+3+1+2+1+2, 1+2+2) \\ &= end_{tr_2(G_3)}(S_1) + 10 \end{aligned}$$

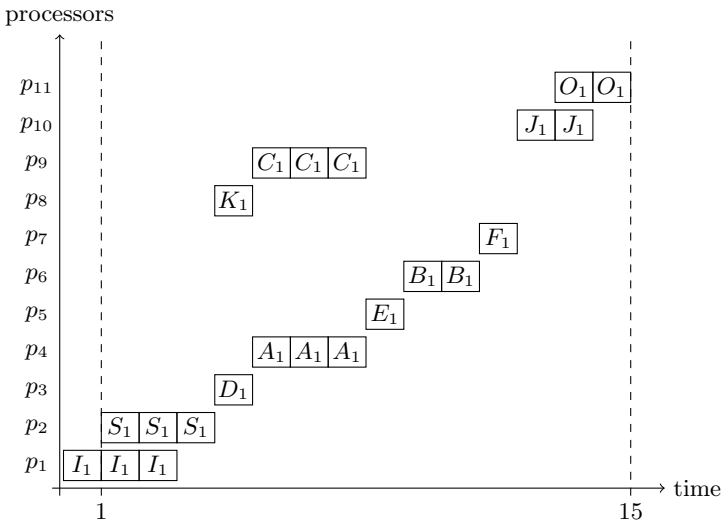


Figure 4.16: The schedule of the resulting graph $tr_2(G_3)$

According the equation 4.8, we have:

$$\Delta\mathcal{L} \leq \max_{path \in P(H_2)} \sum_{m \in path} sol(m) \cdot t(m) - \max_{path \in P(H_1)} \sum_{n \in path} t(n)$$
$$= 10 - 3 = 7$$

For the graph G_3, the impact of tr_2 on the latency reaches the upper bound. If in G_3, we add an actor L_1 between S_1 and J_1 with execution time $t(L) = 8$, then $\Delta\mathcal{L}$ would be 0.

We have a similar result for the transformation rules improving the latency.

Proposition 3. *Let G_1 be a graph with latency $\mathcal{L}_{G_1}$, and $tr : H_1 \Rrightarrow H_2$ be a transformation rule, such that:*

- *H_1 and H_2 are two source-sink acyclic graphs, both having the source actor x and the sink actor y;*
- *the latency of H_2 is less than H_1.*

Let $\mathcal{L}_{G_2}$ be the latency of the resulting graph $G_2 = tr(G_1)$. The impact of the transformation rule tr on the latency of the graph is $\Delta\mathcal{L} = \mathcal{L}_{G_1} - \mathcal{L}_{G_2}$ with

$$\Delta\mathcal{L} \leq \max_{path \in P(H_1)} \sum_{c \in path} sol(c) \cdot t(c) - \max_{path \in P(H_2)} \sum_{c \in path} t(c) \tag{4.9}$$

Proof. This proposition is the dual of the proposition 2 and its proof follows the same reasoning. □

Example 4: Consider the graph G_4 shown in Figure 4.17 where $t(A) = 1$ and $t(S) = t(B) = t(J) = 2$. The ASAP schedule for G_4 is shown in Figure 4.18. The latency of the graph is 12.

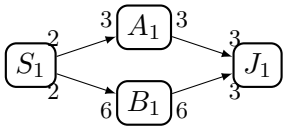


Figure 4.17: Graph G_4

If we apply the transformation rule tr_3 (Figure 4.19) on the graph G_4, we obtain the resulting graph $tr_3(G_4)$ which is scheduled as in Figure 4.20. The latency of the resulting graph is 9, therefore the improvement is $\Delta\mathcal{L} = 12 - 9 = 3$ which is the upper bound of the proposition 3, $\Delta\mathcal{L} \leq 6 - 3 = 3$.

If now we suppose that the execution time of all actors is 2, then $\mathcal{L}_{G_4} = 12$ and $\mathcal{L}_{tr_3(G_4)} = 10$ and $\Delta\mathcal{L} = 12 - 10 = 2$. According to the proposition 3, $\Delta\mathcal{L} \leq 4$. If the incoming rate and outgoing rate of the actor B is 2, $\mathcal{L}_{G_4} = \mathcal{L}_{tr_3(G_4)} = 10$, therefore $\Delta\mathcal{L} = 0$. According to proposition 3, $\Delta\mathcal{L} \leq 10 - 4 = 6$.

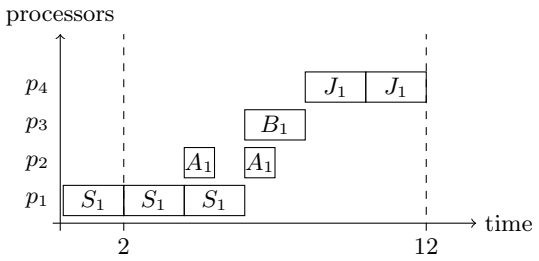


Figure 4.18: The ASAP schedule for the graph G_4

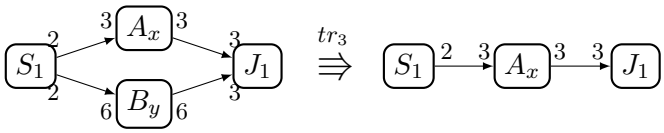


Figure 4.19: Transformation rule tr_3

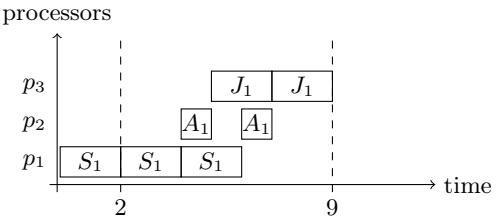


Figure 4.20: The ASAP schedule for the resulting graph $tr_3(G_4)$

4.2 Analysis of the impact on throughput

In this section, we compute bounds of the impact of a transformation rule on the throughput. We focus on acyclic graphs and we illustrate our results on several examples.

In the following proposition, we consider the class of all acyclic graphs for which the throughput is defined by the actor with the highest load, that is, the actor v having the maximum $sol(v) \cdot t(v)$. Either (1) the transformation rule adds an actor with a higher load to the graph and in that case the throughput is decreased, or (2) the rule removes the actor with the highest load and in that case the throughput is increased, or (3) the transformation does not change the throughput. Since the transformation rule is applied to an unknown graph, its actor with the highest load is not known, therefore we can only compute approximate bounds.

Proposition 4. *Let G_1 be an acyclic graph with throughput $\mathcal{T}_{G_1}$ with each actor being executed on a processor according to an ASAP schedule, let tr : $lhs \Rrightarrow rhs$ be a transformation rule, and $\mathcal{T}_{G_2}$ be the throughput of the resulting graph $G_2 = tr(G_1)$. The impact of the transformation rule on the throughput*

of the graph is $\Delta\mathcal{T} = \mid \mathcal{T}_{G_2} - \mathcal{T}_{G_1} \mid$ *such that :*

$$\Delta\mathcal{T} \leq \mid \frac{1}{\max_{v \in V_{rhs}} sol(v) \cdot t(v)} - \frac{1}{\max_{v \in V_{lhs}} sol(v) \cdot t(v)} \mid \tag{4.10}$$

Proof. Since G_1 is acyclic and each actor is being executed on a processor according to an ASAP schedule, its throughput is:

$$\mathcal{T}_{G_1} = \frac{1}{\max_{v \in V_{G_1}} sol(v) \cdot t(v)} \tag{4.11}$$

Let m be the actor such that $m = argmax_{v \in V_{G_1}} sol(v) \cdot t(v)$. We do not know if m is present in the transformation rule tr, but we know that there is an actor x such that $x = argmax_{v \in V_{lhs}} sol(v) \cdot t(v)$ and there is an actor y such that $y = argmax_{v \in V_{rhs}} sol(v) \cdot t(v)$. We distinguish the following cases:

- Case A : $sol(x) \cdot t(x) > sol(y) \cdot t(y)$. If $m = x$, then $sol(x) \cdot t(x) = sol(m) \cdot t(m)$, hence m cannot be in the *rhs* of tr because this would contradict the inequality $sol(x) \cdot t(x) > sol(y) \cdot t(y)$. In other words, m is suppressed by tr. In this case, the throughput is increased and the impact of tr on the throughput is $\frac{1}{sol(y) \cdot t(y)} - \frac{1}{sol(x) \cdot t(x)}$. If $m \neq x$, then $m \notin lhs$, then m is not suppressed by tr and therefore belongs to $tr(G_1)$. In this case the throughput remains unchanged.

- Case B : $sol(x) \cdot t(x) < sol(y) \cdot t(y)$. If $sol(y) \cdot t(y) > sol(m) \cdot t(m)$, then the throughput is decreased and the impact of the rule on the throughput is $\frac{1}{sol(y)t(y)} - \frac{1}{sol(x)t(x)}$. Otherwise, the throughput is unchanged.

- Case C : $sol(x) \cdot t(x) = sol(y) \cdot t(y)$. In this case the throughput does not change.

To summarize, the transformation rule either does not change the throughput or its impact on the throughput is $\mid \frac{1}{\max_{v \in V_{rhs}} sol(v) \cdot t(v)} - \frac{1}{\max_{v \in V_{lhs}} sol(v) \cdot t(v)} \mid$. □

Example 5: Consider the graph G_5 shown in the Figure 4.21. The execution times of actors are respectively $t(A) = 2$, $t(B) = 4$, and $t(C) = 3$. The throughput of the graph is $\mathcal{T}_{G_5} = \frac{1}{sol(C_1) \cdot t(C_1)} = \frac{1}{9}$.

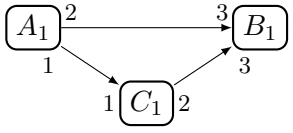


Figure 4.21: Graph G_5

In the transformation rule tr_4 shown in Figure 4.22, the execution time of the actor D is 2. If we apply the transformation rule tr_4, then the throughput of the resulting graph $G_6 = tr_4(G_5)$ (shown in Figure 4.23) becomes $\mathcal{T}_{G_6} = \frac{1}{6 \cdot 2} = \frac{1}{12}$. The actual impact of tr_4 on G_5 is $\Delta\mathcal{T} = \mid \frac{1}{12} - \frac{1}{9} \mid = \frac{1}{36}$.

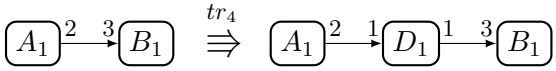


Figure 4.22: Transformation rule tr_4

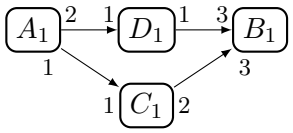


Figure 4.23: The resulting graph $G_6 = tr_4(G_5)$

According to the equation 4.10, we have :

$$\Delta\mathcal{T} \leq \mid \frac{1}{sol(D_1) \cdot t(D_1)} - \frac{1}{sol(B_1) \cdot t(B_1)} \mid = \mid \frac{1}{12} - \frac{1}{8} \mid = \mid -\frac{1}{24} \mid = \frac{1}{24}$$

As expected from proposition 4, the actual $\Delta\mathcal{T}$ is less than the upper bound : $\frac{1}{36} < \frac{1}{24}$.

If we suppose that $t(C) = 1$, we have $\mathcal{T}_{G_5} = \frac{1}{sol(B_1) \cdot t(B_1)} = \frac{1}{8}$ and $thr_{G_6} = \frac{1}{sol(D_1) \cdot t(D_1)}$, therefore $\Delta\mathcal{T} = \frac{1}{24}$ and the computed bound is reached.

Now suppose that $t(C) = 5$. In this case, the throughput remains unchanged, that is, $\mathcal{T}_{G_6} = \mathcal{T}_{G_5} = \frac{1}{15}$. Since we do not have any information about the dataflow graph at compile-time, we can only say that $\Delta\mathcal{T} \leq \frac{1}{24}$, although for this dataflow graph $\Delta\mathcal{T} = 0$.

Throughput in terms of tokens per second

Another unit for measuring the throughput of dataflow graphs is the number of tokens per second instead of the number of iterations per second. First, we see on an example why this unit is important for measuring the throughput, and then we discuss how we can compute the impact of a transformation rule on throughput measured in terms of this unit.

Consider the dataflow graph G_7 shown in Figure 4.24. The execution time of actor A_1 is 10 second and the execution time of the other actors is 1 second. The throughput of the graph is $\frac{1}{10}$ iterations per second.

If we apply the transformation rule tr_5 (Figure 4.25) on G_7, the graph $tr_5(G_7)$ is obtained (Figure 4.26). The throughput of the resulting graph $tr_5(G_7)$ remains unchanged, that is, $\frac{1}{10}$ iteration per second.

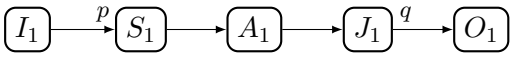


Figure 4.24: Graph G_7

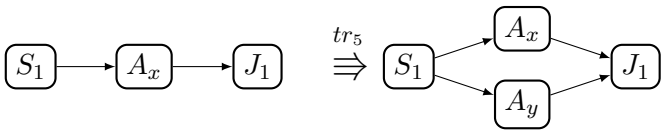


Figure 4.25: Transformation rule tr_5

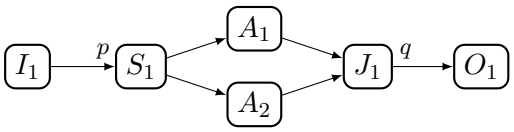


Figure 4.26: The resulting graph $tr_5(G_7)$

Although the throughput has not changed in terms of iteration per second, the number of tokens processed in one iteration and consumed by the sink actor O_1 is doubled. In G_7 the throughput is $\frac{1}{10}$ token per second, whereas in $tr_5(G_7)$, the throughput is $2 \cdot \frac{1}{10} = \frac{1}{5}$ token per second.

By increasing the parallel levels of computation, we can process more tokens per second and therefore we improve the throughput in terms of tokens per second. Therefore, 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 single sink O, this measure is equal to the number of tokens consumed by O per time unit.

For a graph with a single sink actor consuming n tokens per iteration we have

$$\mathcal{H}_G = n \cdot \mathcal{T}_G \tag{4.12}$$

with

$$\mathcal{T}_G = \frac{1}{\max_{v \in V} sol(v) \cdot t(v)} \tag{4.13}$$

This measure is more informative than the throughput since the graph and its iteration may change dynamically.

In proposition 4, we showed that $\Delta\mathcal{T} < \mid \mathcal{T}_{rhs} - \mathcal{T}_{lhs} \mid$. For the impact of the transformation rule on the throughput in terms of tokens per second, supposing that the sink actor of a graph never changes, we have:

$$\Delta\mathcal{H} < \mid \mathcal{H}_{rhs} - \mathcal{H}_{lhs} \mid \tag{4.14}$$

where $\mathcal{H}_{lhs}$ (resp. $\mathcal{H}_{rhs}$) is $\mathcal{T}_{lhs}$ (resp. $\mathcal{T}_{rhs}$) multiplied by the solution of the sink actor before (resp. after) applying the transformation rule.

In the graph G_7, the iteration is $(I_1^p, S_1, A_1, J_1, O_1^q)$. The solution of the actor O_1 is therefore q. After applying the transformation rule tr_5, the value of q is increased by 1. Therefore, if the solution of O_1 is $k \in \mathbb{N}$ before applying the transformation rule, it is $k+1$ after applying the rule. According to the equation 4.14, we have :

$$\Delta\mathcal{H} \leq \mid (k+1) \cdot \frac{1}{10} - k \cdot \frac{1}{10} \mid = \frac{1}{10}$$

We computed before $\mathcal{H}_{G_7} = \frac{1}{10}$ and $\mathcal{H}_{tr_5(G_7)} = \frac{1}{5}$, therefore, $\Delta\mathcal{H} = \frac{1}{10}$. The computed upper bound is reached in this example.

This number of tokens consumed per time unit is a measure that we use in the next chapter to characterize transformation rules improving throughput.

4.3 Summary

The impact of a transformation rule on a performance metric of the dataflow graphs can be dynamically computed by analyzing the performance metric before and after applying the transformation rule and computing the difference. However, this dynamic analysis approach does not give any information about the impact of each transformation rule on the performance metrics of a graph to the designer at compile-time.

Static performance analysis, on the other hand, is a way to provide some feedback about the impact of transformation rules on performance metrics of graphs statically. In this chapter, we focused on two performance metrics, that is, latency and throughput. Of course, since a transformation rule can be applied to different graphs unknown at compile time, a precise analysis is impossible. However, for some classes of transformation rules, we provided upper bounds on the impact of transformation rules on latency and throughput of dataflow graphs.

All formulas on the impact of transformation rules on latency and throughput obtained in this chapter depend on the execution times and solutions of actors in the transformation rule. However, if the types of all actors are known, their execution times are known and if at least one actor in the pattern is labeled by its name, then its solution along with the solutions of other actors may also be known. Labeling an actor by its name is a technique used in the next chapter for pattern matching. Therefore, the bounds are either numerical, or they are symbolic and can be instantiated dynamically.

Chapter 5

Implementation

In chapter 3, we presented RDF, but we did not specify how it was implemented. For instance, once a condition becomes true a transformation program is applied, but we have not discussed what kinds of conditions are used in practice and what steps are needed to apply the transformations. Before applying a transformation rule, the left-hand side of a rule must be matched into the graph. The efficiency of pattern matching remains to be addressed. Finally, once a transformation rule is applied, new appearing actors must be placed on available processing units, so a strategy is needed to perform such placement.

In section 5.1, we address all these questions by describing our prototype implementation of RDF. In section 5.2, we show some experiments performed using this prototype. We measure the costs incurred to perform a transformation rule for some examples. We also show an example in which by applying a transformation rule, the level of parallelization is increased and as a result, the throughput of the graph is increased. In section 5.3, we present a case study of RDF in the domain of video processing where the throughput may decrease due to some external changes. RDF allows to dynamically increase the level of parallelization and, as a result, to maintain the throughout.

5.1 System description

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

dynamically.

5.1.1 Standard execution

The initial dataflow graph is built 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 corresponds to $p \cdot sol(A)$ tokens. This is enough to achieve maximal throughput but smaller bounds are known for special classes of graphs. The types of tokens as well as the values of initial tokens, which are not part of the model, must be specified in the application (*e.g.,* integer, real, array of bit, *etc.*). Then, the communication links (edges) are created by providing to each input port a reference to the output port it connects to. Depending on the architecture (*e.g.,* single or multiple servers), actors can communicate through shared memory or message passing. Our prototype runs on a single multi-core machine and uses only buffers in shared memory to communicate. Finally, a thread is created for the controller and for each actor.

If multiple servers were used, then the address of the machine on which each actor runs should be specified. Then, each machine would be requested to create its own actors. For each input port, a message socket would be opened. Moreover, the actors should be provided with information related to the address of the machine and the port of their successors.

We first discuss how actors are executed between reconfigurations and what data structures are needed to implement them. Then, we discuss how actor communicate with each other through buffers and what data structure is needed to implement these buffers.

Actor's execution

Algorithm 1 presents the procedure ExcuteActor for the execution of a generic actor with `n` input ports and `m` output ports. The actor runs in an infinite loop which performs the following steps. At each iteration, it is fired as many time times as its solutions in one iteration. At each firing, it calls the Consume function on each of its input ports, it retrieves the tokens, processes the tokens (*i.e.,* applies the actor's functionality), and finally calls the Produce function on each of its output ports to write the produced tokens on the corresponding buffers.

Once an iteration is finished, the function WaitIfPaused is called. This functions checks if the controller asked to pause the graph for a reconfiguration. For this matter, the controller must notify the actor whether it is time to pause, and it must tell the actor until which iteration it has to continue its execution. Once the actor is paused, it has to wait for a signal from the controller to resume. This procedure is discussed more in section 5.1.2.

The data structure of Actor is presented in algorithm 1. The integer values `solution` and `iteration` are the solution and current iteration number of the actor. The array `ips` (resp. `ops`) of type InputPort (resp. OutputPort) is the list of input (resp. output) ports of the actor. The function `f` denotes the functionality of the actor. It takes as many argument as the number of input ports and returns as many values as the number of output ports. The integer values `n` and `m` are the number of input and output ports of the actor respectively. The boolean value `paused` indicates whether the controller has requested the actor to be paused. In order to pause the graph, the controller must pause all actors first (see algorithms 4 and 6). The integer value `iter_max` is the iteration number at which the actor must pause. It is the iteration number of the actor which is the fastest actor and has the largest iteration number among all actors of the dataflow graph. All actors needs to reach this iteration. Its value is set by the controller. The signal `sig` is used to notify the actor that the values of `paused` and/or `iter_max` have changed.

Communication through buffers

All buffers are created in the shared memory. Synchronization mechanisms of the circular buffers ensure that the execution rules of SDF are respected, that is, actors block when trying to read from empty buffers and when trying to write on full buffers.

When a consumer wants to read from a buffer through its input port, it first verifies if there are enough tokens to read according to its consumption rate (see functions Consume and Read in algorithm 2). If there are less available tokens than the rate, it waits for the signal `prod_sig` which is sent by the producer each time it produces (see the function Produce in algorithm 3). This signal `prod_sig` is part of the data structure Buffer (algorithm 2). It may be the case that after a production, there are still not enough tokens available, and in that case the consumer keeps waiting. Once the consumer successfully reads its buffer, it updates the number of tokens available on the buffer and sends the signal `cons_sig` to the producer.

A producer cannot write on a buffer when there is not enough space, that is, when the number of empty locations on the buffer is less that its production rate. In this case, it has to wait for signal `cons_sig` from the consumer until enough number of locations is available on its buffer. Once it has finished writing, it updates the number of tokens available on the buffer and sends the signal `prod_sig` to the consumer (see functions Produce and Write in algorithm 3).

The data structure InputPort is presented in algorithm 2. It has an integer value `cons_rate` which is the consumption rate of the input port. The input port reads token with this rate from its associated `buffer`. The dual data structure OutputPort is similar and described in algorithm 3.

Algorithm 1 Actor's execution

```
1: procedure ExecuteActor(Actor actor)
   ▷ The procedure executes a given actor and checks after each iteration
   whether the controller has requested the actor to pause.
2:     actor.iteration := 1;
3:     n := actor.n;
4:     m := actor.m;
5:     while true do
6:         firing := 1;
7:         while firing <= actor.solution do
8:             it_1 := Consume(actor.ips[1]);
9:             ...
10:            it_n := Consume(actor.ips[n]);
11:            ot_1, ..., ot_m := actor.f(it_1, ..., it_n);
12:            Produce(actor.ops[1], ot_1);
13:            ...
14:            Produce(actor.ops[m], ot_m);
15:            firing := firing + 1;
16:        end while
17:        WaitIfPaused(actor)
18:        actor.iteration := actor.iteration + 1;
19:    end while
20: end procedure
21:
22: procedure WaitIfPaused(Actor actor)
   ▷ The procedure pauses the actor if the controller has requested to
   pause.
23:    while actor.paused ∧ actor.iteration == actor.iter_max do
24:        wait_for_signal(actor.sig)
25:    end while
26: end procedure
27:
28: structure Actor
29: int solution; int iteration;
30: function f; int n; int m;
31: InputPort array ips; OutputPort array ops;
32: bool paused; int iter_max; signal sig;
33: end structure
```

The data structure Buffer (algorithm 2) has the integer values `size` and `nb_tokens` which denote the size of the buffer and the number of tokens in the buffer. The signal `prod_sig` (resp. `cons_sig`) is the signal for the producer (resp. consumer) on the buffer to notify the consumer (resp. producer) that

it has produced on (resp. consumed from) the buffer. The `array` is the array containing the data (tokens) on the buffer. The integer values `r_idx` and `w_idx` are the indexes to the last accessed location in the array for reading and writing.

Algorithm 2 Consumption from an input port

```
1:  procedure CONSUME(InputPort ip)
    ▷ The procedure returns a list of tokens read by a given input port ip.
2:      while ip.buffer.nb_tokens < ip.cons_rate do
3:          wait_for_signal(ip.buffer.prod_sig);
4:      end while
5:      it := READ(ip.buffer, ip.cons_rate);
6:      send_signal(ip.buffer.cons_sig);
7:      return it;
8:  end procedure
9:
10: procedure READ(Buffer buffer, Integer r)
    ▷ The procedure returns r tokens from the buffer.
11:     buffer.nb_tokens := buffer.nb_tokens - r;
12:     last := (buffer.r_idx + r - 1) % buffer.size;
13:     it[1 .. r] := buffer.array[buffer.r_idx .. last];
14:     buffer.r_idx := last;
15:     return it;
16: end procedure
17:
18: structure InputPort
19: int cons_rate;
20: Buffer buffer;
21: end structure
22:
23: structure Buffer
24: int size; int nb_tokens;
25: signal prod_sig; signal cons_sig;
26: array array; int r_idx; int w_idx;
27: end structure
```

Since each actor runs on a separate thread, actors can be executed in parallel independently of each other according to an ASAP policy; an actor fires as soon as it has enough tokens on its input ports. Note that we do not consider auto-concurrency, because each actor is running in a single thread and each firing can happen only after the previous firing has ended.

Each port specifies the type of tokens for its buffer, so when a circular buffer is created, enough memory is allocated for each location on the buffer. The circular buffer is implemented using an array and an index to the last

Algorithm 3 Production on an output port

```
1: procedure PRODUCE(OutputPort op, output tokens ot)
   ▷ The procedure writes the list of tokens ot to the output port op.
2:     while op.buffer.size - op.buffer.nb_tokens < op.prod_rate do
3:         wait_for_signal(op.buffer.cons_sig)
4:     end while
5:     WRITE(op.buffer, ot, op.prod_rate);
6:     send_signal(op.buffer.prod_sig)
7: end procedure
8:
9: procedure WRITE(Buffer buffer, output tokens ot, Integer r)
   ▷ The procedure writes r output tokens ot on the buffer.
10:    buffer.nb_tokens := buffer.nb_tokens + r;
11:    last := (buffer.w_idx + r - 1) % buffer.size;
12:    buffer.array[buffer.w_idx ..  last] := ot[1 ..  r];
13:    buffer.w_idx := last;
14: end procedure
15:
16: structure OutputPort
17: int prod_rate;
18: Buffer buffer;
19: end structure
```

filled cell, where the index returns to the first cell once reached the last cell (see algorithms 2 and 3). By default, the size of the buffer of a port with rate p of an actor A is set to accommodate $p \cdot sol(A)$ tokens. This size can also be specified by the programmer of the application and must assure that no deadlock is created. For each edge (A, B) with production rate p and consumption rate q, the minimum buffer size to ensure that the graph executes without deadlock is $p+q-\gcd(p,q)$ [7]. This is sufficient to prevent deadlocks but not to achieve maximal throughput. According to [11] for a graph without any undirected cycle, if the buffer of every edge (A, B) with production rate p and consumption rate q is at least $2(p + q - \gcd(p, q))$, then the ASAP execution of the graph achieves the maximal throughput.

5.1.2 Reconfigurations

In chapter 3, we presented an expressive language for the reconfiguration program of the controller. In our prototype, we considered a simplified version of this language in which the reconfiguration program is specified as a list of single transformation rules triggered by conditions: $[cond_1 : lhs_1 \Rrightarrow rhs_2; \ldots; cond_n : lhs_n \Rrightarrow rhs_n]$. The rules are ordered and only the first

applicable one is considered at each reconfiguration. The conditions will be described in 5.1.3.

The process of a reconfiguration is described in algorithm 4. The procedure of the controller takes a dataflow graph `G` and a reconfiguration program `P`. The structure of a graph (type Graph) is presented in algorithm 4. A graph has an array of actors and an array of edges. An edge is a pair of actors. A rule (type Rule) has a left-hand side `lhs` and a right-hand side `rhs` both of type Graph. A reconfiguration program (type Program) is made of an array of program elements `progs` (type ProgramElement) and an integer value `size` denoting the size of the array. A program element itself is made of a string value `cond` which is the condition triggering the `rule` and a `subgraph` which is matched in the left-hand side of the rule. The value of the `subgraph` is null either if the controller has not yet processed its associated program element or if it has processed but has not found any subgraph in the dataflow graph matching the rule.

The controller, which runs on its own thread, finds the transformation rules whose left-hand side matches the current graph (FindMatchingRules). Since the graph does not change between two consecutive transformations, finding matching rules is done once after each reconfiguration. It is performed in parallel with execution of the dataflow graph and its cost is reduced.

The controller checks every `t` units of time whether a transformation rule is applicable. A transformation rule is applicable when its condition is true *and* its left-hand side matches the current dataflow graph. The value of `t`, which can be specified by the programmer, must be large enough, so that the graph has enough time to return to its steady state and so that conditions (*e.g.,* throughput) can be measured correctly. The procedure ApplicableRule in algorithm 5 finds the transformation for which the condition holds *and* for which the procedure FindMatchingRules has found a subgraph. If a rule is applicable, then the reconfiguration is performed.

Reconfigurations cannot be performed any time during the execution. Transforming the dataflow graph in the middle of an iteration or when actors are not in the same iteration would cause data loss. A reconfiguration should only occur in a consistent state, that is, after an iteration has completed and the graph has returned to its initial state. By the initial state, we mean the initial state of the graph after the previous reconfiguration. Moreover, all actors must have completed the same iteration.

Before reconfiguring the graph, the execution must be paused. The function Pause, called by the controller to stop the execution, is presented in algorithm 6. The controller sets the value of `actor.paused` for all actors to true. It also requests the iteration number of all actors and computes the largest iteration number. It then sets the value of `actor.iter_max` for all actors, so that they continue until the end of that iteration and pause.

Once all actors reach the iteration `iter_max`, the controller knows that the graph is back to its initial state, that is, the same state as it was just after

Algorithm 4 Reconfiguration steps

```
1: procedure Controller(Graph G, Program P)
   ▷ The controller takes a reference of the current dataflow graph G and a
   controller program P. If an applicable rule is found, the controller pauses,
   reconfigures, and finally resumes the graph.
2:     FindMatchingRules(G, P);
3:     while true do
4:         wait(t units of time);
5:         idx := ApplicableRule(P);
6:         if idx != null then
7:             Pause(G);
8:             Reconfigure(G, P, idx);
9:             Resume(G);
10:            FindMatchingRules(G, P);
11:        end if
12:    end while
13: end procedure
14:
15: structure Graph
16: Actor array actors; Actor pair array edges;
17: end structure
18:
19: structure Rule
20: Graph lhs; Graph rhs;
21: end structure
22:
23: structure ProgramElement
24: string cond; Rule rule; Graph subgraph;
25: end structure
26:
27: structure Program
28: ProgramElement array progs; int size;
29: end structure
```

the previous reconfiguration. The controller performs the transformation rule which may involve removing actors and edges (*i.e.,* buffers). It may also create actors with fresh new names on new threads and edges with newly allocated buffer. A valid transformation rule always removes an actor along with all its edges, so there never remain disconnected buffers. When a transformation rule replaces an edge, the references from input ports to corresponding output ports are updated (see the function Reconfigure in algorithm 6).

After reconfiguring the graph, the controller asks all actors to resume

Algorithm 5 Finding applicable rule and matching rules

```
1: procedure ApplicableRule(Program P)
   ▷ The procedure takes a reconfiguration program P and returns the first
   matching rule whose associated condition holds.
2:     idx := 0;
3:     while idx < P.size do
4:         if P.progs[idx].cond == true
             and P.progs[idx].subgraph != null then
5:           return idx;
6:         end if
7:         idx := idx + 1;
8:     end while
9:     return null
10: end procedure
11:
12: procedure FindMatchingRules(Graph G, Program P)
   ▷ The procedure takes the reference of the current dataflow graph G and
   a reconfiguration program P and finds the matching rules.
13:    idx := 0;
14:    while idx < P.size do
15:        P.progs[idx].subgraph := null;
16:        FindMatching(G, P.progs[idx]);
17:        idx := idx + 1;
18:    end while
19: end procedure
20:
21: procedure FindMatching(Graph G, ProgramElement PE)
   ▷ This procedure implements pattern matching and finds for the pattern
   PE.rule.lhs in the graph G. Once a pattern is matched, it fills the
   PE.subgraph with a reference to the matched subgraph. The pattern
   matching is described in section 5.1.4.
22: end procedure
```

their executions by setting the value of the `actor.paused` to false. It then sends the signal `actor.sig` so that the threads of actors sleeping on those signal wake up and continue their execution (see the function Resume in algorithm 6). The execution proceeds as before, each actor on its own thread firing as soon as possible. Finally, after the reconfiguration, the controller looks for the rules matching the new graph by calling FindMatchingRules.

Algorithm 6 Pausing, reconfiguring, and resuming the graph

```
1:  procedure PAUSE(Graph G)
    ▷ The procedure pauses all actors of the current dataflow graph G at an
    iteration number.
2:      max := 0;
3:      for each actor in G.actors do
4:          iter := actor.iteration;
5:          actor.paused := true;
6:          actor.iter_max := iter;
7:          max := maximum(max, iter);
8:      end for
9:      for each actor in G.actors do
10:         actor.iter_max = max;
11:         send_signal(actor.sig);
12:     end for
13: end procedure
14:
15: procedure RECONFIGURE(Graph G, Program P, idx)
    ▷ This procedure (1) waits for all actors to reach the iteration iter_max,
    (2) creates the appearing actors of the P.progs[idx].rule with fresh
    new names, (3) deallocates the disappearing actors and edges (buffers)
    of the P.progs[idx].rule from the graph G, and (4) establishes new
    connections.
16: end procedure
17:
18: procedure RESUME(Graph G)
    ▷ The procedure resumes all actors of the current dataflow graph G.
19:     for each actor in G.actors do
20:         actor.paused := false;
21:         send_signal(actor.sig);
22:     end for
23: end procedure
```

5.1.3 Program conditions

In this section, we present the conditions that can be used in the reconfiguration program. A condition is a boolean expression of variables which may belong to four categories: performance metrics, model parameters, actor-specific variables, and system variables. Other categories might also be considered.

Performance metrics: There are three important performance metrics for which we can decide to change the configuration of the graph: throughput, latency, and buffer occupancy.

For instance, when the throughput of the dataflow graph is above or below some threshold, or the latency of an iteration is above a threshold (which indicates that a deadline is at risk), or the occupancy of a buffer is too high (which means that the tokens accumulate and may not be treated in time), a transformation rule may be triggered. Therefore, a reconfiguration may be performed to prevent that the throughput of a system drops below a threshold, to prevent a deadline miss or to prevent token accumulation. One way of coping with these situations is by increasing the level of parallelism in some part of the graph.

Similarly, if the throughput is too high, or the latency or the buffer occupancy is too low (which indicates that resources are over-utilized), some resources can be released by decreasing the level of parallelism.

Examples of conditions in this category are `throughput(A) > m`, `latency(A) > n`, or `occupancy(AB) < o`, where `A` is an actor, `AB` is an edge, and `m,n,o` are constant integers.

Model parameters: A reconfiguration might be allowed only if some parameters of the model (*e.g.,* the parameter of a variable arity actor) remain in a range of values. For example, consider a graph with two split and join actors which control the parallelization level. If this arity increases to a certain value, we may have to prevent the parallelism level to increase.

Example of a condition in this category is `output_arity(S) < k` where `S` is a split actor and `k` is an integer. The condition ensures that the reconfiguration is performed only if the arity of the split actor is less than a threshold.

Actor-specific variables: Some actors may provide a number of functions for returning the values of their internal variables. Based on the values of these variables, the controller can decide to change the configuration of the graph. For example, based on the number of macroblocks in an image, we may want to change the configuration of a multimedia RDF application. A decoder actor possesses the information about the macroblocks. This information can be accessible to the controller if the actor provides a function to return it.

One example of this style of condition is `macroblocks(A) = k` where `A` is an actor for which the function `macroblocks` is defined and exposed to be used by the controller and `k` is an integer.

Environmental variables: A condition can also refer to the number of units of time that has passed since some event (*e.g.,* the first firing of an actor). For example, because of energy concern, we would like to change the type of a decoder after some period of time. Another environmental variable is the absolute time. For example, when the night falls, we may need to add a night filter to an image processing application. We can imagine other environmental variables such as temperature, humidity, etc.

Examples of conditions of this kind are `time() = m` meaning that the absolute time is `m` or `elapsed_time(A,k) = n` meaning that `n` units of time

is passed since the $\mathtt{k}^{th}$ firing of actor `A`.

Of course, this list of conditions is not exhaustive and other kinds of conditions may be considered without changing the rest of implementation.

5.1.4 Pattern matching

After each reconfiguration, the controller checks which transformation rules match the current graph, that is, the left-hand side of which rules match a subgraph of the dataflow graph. The problem of matching a graph to a subgraph of another graph is called the subgraph isomorphism problem [24].

Given two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$, the subgraph isomorphism problem asks whether G contains a subgraph (V, E) isomorphic to H, that is, a subset $V \subseteq V_G$ and a subset $E \subseteq E_G$ such that $|V| = |V_H|$, $|E| = |E_H|$, and there exists a bijection $f : V_H \to V$ satisfying

$$\forall (u, v) \in E_H \Leftrightarrow (f(u), f(v)) \in E.$$

The subgraph isomorphism problem is NP-complete [24]. It can be solved in polynomial time if G is a forest and H is a tree. For directed graphs, it remains NP-complete even if G is acyclic and H is a directed tree.

If the pattern (left-hand side of the rule) does not contain any variable, then the pattern matching is equivalent to the subgraph isomorphism problem. But if the pattern has variables, the bijection f maps the set of actors of V_H to the set of actors of V in such a way that the substitution function σ finds unique values for all variables of the pattern. Supposing that the pattern contains only variables for the names, the pattern matching problem may be formalized as follows.

Given a pattern $lhs = (V_{lhs}, E_{lhs})$ and a dataflow graph $G = (V_G, E_G)$, the pattern matching problem tries to find a subgraph G isomorphic to *lhs*, that is, a subset $V \subseteq V_G$ and a subset $E \subseteq E_G$ such that $|V| = |V_{lhs}|$, $|E| = |E_{lhs}|$, and there exists a substitution function $\sigma : V_{lhs} \to V$ satisfying

$$\forall (u, v) \in E_{lhs} \Leftrightarrow (\sigma(u), \sigma(v)) \in E.$$

The substitution function $\sigma(x)$ returns the actor in a graph matched to the actor x in the left-hand side of a transformation rule.

Graph pattern matching therefore is a potentially costly operation that may involve graph exploration and backtracking. In our implementation, we impose constraints such that pattern matching is linear in the size of the *lhs*. We enforce that the *lhs* has at least one named actor serving as a root and that all the edges of the pattern can be traversed starting from roots and following explicit ports.

Matching such an *lhs* consists in selecting the named actors and proceeding by following the named ports. These constraints ensure that the whole *lhs* can be traversed and that the whole subgraph matching the *lhs* can be selected without backtracking.

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

This pattern has two explicitly named actors that can be directly selected in the graph: A_5 and B_3. Note that these actors must appear the initial graph since transformation rules do not know the names of dynamically created actors. From these roots, α_x, β_y and γ_z can be selected unambiguously. Indeed, the edges (α_x, A_5) and (A_5, β_y) can be followed by the input port 1 and output port 2 of A_5 respectively. The edge (γ_z, B_3) can be followed using the input port 3 of B_3 and finally (β_y, γ_z) from the input port 1 of γ_z.

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

Note that these constraints can be relaxed. For instance, if the type A has a single input port, the pattern does not need to make it explicit. If the second output port of A is the only one to have the rate 4 then the pattern can use that rate instead. The point is that the matching should proceed without performing any choice and therefore in linear time. It is always possible to make pattern precise enough by introducing dummy named actors to act as pointers on the graph and as roots in patterns. Another stricter constraint would be to force only one actor to be named where all ports are explicit and named.

Of course, if the patterns are small, then in practice the exponential time could be acceptable depending of the type of the application. In that case, we can have a looser constraint. For instance, we can only force one single actor to be named. The time complexity would remain exponential in the size of the left-hand side of the rule, but if the left-hand side is small, that time complexity would be acceptable in practice.

We now consider another example in which variable arity actors are present. Consider the dataflow graph G_1 in the Figure 5.1 and two transformation rules tr_1 and tr_2 in the Figure 5.2.

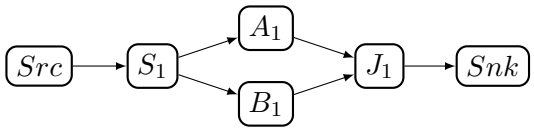


Figure 5.1: Dataflow graph G_1

We can match the *lhs* of tr_1 in linear time because the pattern matching algorithm chooses the named actor S_1, then it retrieves the last (second) port $\bullet last$ and directly matches the actor B_1, then it follows the output port

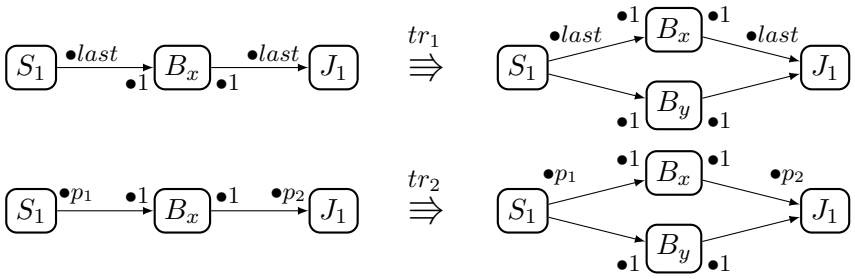


Figure 5.2: The transformation rules tr_1 can be matched in linear time in the size of its left-hand side, but the tr_2 cannot.

$\bullet 1$ of the actor B_1 and reaches J_1 without any backtrack. On the other hand, the pattern matching of the transformation rule tr_2 follows one of the ports of S_1, possibly the first, and does not find an actor of type B and must backtrack. This rule is therefore rejected.

5.1.5 Placement strategy

When placing RDF actors on a multi-processor or multi-server systems, different strategies can be adopted depending on the performance metric needed to be optimized. In particular, we may choose to maximize throughput or to minimize the latency.

In this section, we focus on a placement strategy that aims to improve performance by optimizing two criteria: load balancing and communication costs. The goal is to distribute RDF actors across multiple processors while minimizing inter-actor communication costs. It ensures that processors are equally overloaded and dependent actors are collocated on the same processor. Intuitively this exploits the intrinsic parallelism while reducing communication overheads due to parallelism.

Assuming that the processors and communication channels are homogeneous, the stated multi-criteria optimization problem can be expressed as a graph partitioning problem [24]. Given a graph $G = (V, E)$, with a weight $w(v)$ for each vertex $v \in V$ and a distance $d(e)$ for each edge $e \in E$, the graph partitioning decision problem tries to find whether there exists a partition of V into disjoint sets $V_1, ..., V_m$ such that the sum of weights in each partition is less than some value $K \in \mathbb{N}$, that is,

$$\forall i, \quad 1 \leqslant i \leqslant m, \quad \sum_{v \in V_i} w(v) \leqslant K$$

and such that if $E_{i,j} \subseteq E$ is the set of edges that have their two endpoints in V_i and V_j, where $i \neq j$, then the sum of distances of these edges is less

than some value $J \in \mathbb{N}$, that is,

$$\forall i, j, \quad 1 \leqslant i, j \leqslant m, \quad i \neq j, \sum_{e \in E_{i,j}} d(e) \leqslant J$$

The load of an actor A with execution time $t(A)$ is $sol(A) \cdot t(A)$ during one iteration. The flow of an edge (A, B) with production rate p and consumption rate q is $sol(A) \cdot p$ $(= sol(B) \cdot q)$ tokens during an iteration.

For a dataflow graph, we take the load of an actor as its weight, and the flow of an edge as its distance. In the graph partitioning problem, the sum of weights and distances are unitless. In the dataflow placement problem, the sum of loads of actor is a duration and the sum of flows is a number of tokens. Since each communication channel has a capacity in terms of tokens per second and channels are assumed to be homogeneous, we express the sum of flows in terms of time units. If the size of tokens on different edges is different, we can multiply the flow of each edge by the size of its tokens (expressed in terms of bytes for instance). Therefore, we optimize loads and flows in terms of time units.

However, the graph partitioning problem being NP-complete [24], we have to rely on approximations. We formalize a simple heuristic to deal with placement after reconfiguration.

Heuristic

We assume that the initial dataflow graph has been placed either through an approximation or exact algorithm and focus on a heuristic to place new actors and existing actors whose edges are changed by a reconfiguration. When a transformation rule is applied, some actors remain in the graph, and their neighbors remain unchanged. These actors are not needed to be re-assigned. But if the neighbors of an actor change, then that actor may need to be re-assigned to another processor as well.

At reconfiguration time, we know the load $L(pr)$ on each processor pr, that is, the sum of execution time of actors placed on the processor pr multiplied by their solutions:

$$L(pr) = \sum_{a \in P(pr)} sol(a) \cdot t(a)$$

where $P(pr)$ is the set of all actors placed on the processor pr.

We now consider how to place actor a with load $sol(a) \cdot t(a)$ and the set of incoming edges $in(a)$ and outgoing edges $out(a)$.

Note that the buffer of an edge (a, b) can be put either on the producer side or the consumer side. If the buffer is empty most of the time, then it would be better to place the buffer on the consumer side. Indeed, the cost for the consumer to check if the buffer has enough token will be lower. If

the buffer is full most of the time, then it would be better to place it on the producer side, which lower the cost of checking a full buffer for the producer. We assume that in the average case, buffers are neither empty nor full most of the time. In this case, it does not matter where to place the buffers and we can keep buffers at the producer side as explained before.

In a single-criteria optimization problem in which the communication costs are ignored, assuming that the processors are homogeneous, an obvious policy is to place the new actor on a processor with the minimum load:

$$\arg\min_{pr \in Pr} L(pr)$$

where Pr is the set of all processors.

In our multi-criteria problem, we have to consider also the flow of each communication link. The extra communication cost which is produced by placing actor a on processor pr is as follows:

$$C_a(pr) = \sum_{\substack{e \in \{e \mid (e=(a,b) \in out(a) \\ \vee\ e=(b,a) \in in(a)) \\ \wedge\ b \notin P(pr)\}}} d(e)$$

In the above formula, if actor a is placed on the processor pr, then $C_a(pr)$ computes the sum of the flows of the communication links between a and its neighbors not placed on pr. In other word, it computes the sum of the flows transmitted through distant links.

We can use approximation algorithms to solve the stated problem. This placement strategy may be used on a multi-server architecture, where the communication costs are significant. We have not experimented it, because our implementation of RDF is for a single multi-core server with shared memory. For this architecture, the performance of Linux scheduler is satisfactory.

Linux scheduler

In the experiments, we used the Linux scheduler. We describe here how it works and how it deals with load balancing and communication costs. Linux's Completely Fair Scheduler (CFS) divides the processor's cycles among threads [40]. The scheduler slices the time of the processor and assigns threads to these slices. In order to define the size the slices, the scheduler sets a fixed interval during which each thread must run at least once. Each interval is divided among threads in proportion to their priorities. In our case, the priority of all threads are equal, so the interval is divided equally. When a thread runs, it accumulates its run time. Once the thread run time exceeds its assigned time slice, it is pre-empted from the processor if there is another thread ready to execute.

Threads are organized in a run queue, implemented as a red-black tree. In this tree, the threads are sorted in the increasing order of their run time. The scheduler always picks the thread with smallest run time.

In case of a multi-core architecture with a shared memory, each core has its own run queue. In this case, the run queues must be kept balanced. The scheduler periodically runs a load-balancing algorithm to keep the queues roughly balanced. The scheduler also takes the cache and memory hierarchy into account and favors keeping threads which communicate more on the same core to improve cache reuse, and thus reducing latency to access memory.

For RDF, each actor runs on a separate thread and the Linux scheduler places them in such a way that the load is balanced on all cores. Moreover, if two actors communicate through a buffer, since the producer wakes up the consumer thread, according to the Linux placement strategy, they are placed such that the cache coherency overhead reduces which results in lower communication cost. For a multi-server architecture, the Linux scheduler is likely to be inefficient since it does not have information about the communication costs.

5.2 Experiments

We now describe the experiments we have performed using our RDF implementation. Our objectives were to evaluate reconfiguration costs, and show how RDF can be used to change levels of parallelization and dynamically improve performance.

For the experiments, we used an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 12 cores running Linux. A single shared memory is available to which all cores have access. A single L3 data cache is shared between all cores. Each core has also two dedicated L1 and L2 caches.

For the performance metric of a given graph G, we use the throughput in terms of tokens per second $\mathcal{H}_G$ introduced on page 75 in chapter 4. For a graph with a single sink actor consuming n tokens per iteration we have

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

with $\mathcal{T}_G$ denoting the standard notion of throughput. In the following, we often use the term throughput to refer to the number of tokens consumed per second.

5.2.1 Reconfiguration costs

An important metric to evaluate is the cost of reconfigurations in terms of time units. Indeed, RDF would lose part of its interest for streaming applications if reconfiguration takes too long. This cost can be decomposed in two parts:

Rule	Matching cost (μs)	Transformation cost (ms)
$insert_{10}$	40	1.44
$insert_{20}$	35	2.56
$insert_{30}$	45	3.88
$insert_{40}$	50	4.88
$remove_{10}$	225	0.75
$remove_{20}$	530	1.49
$remove_{30}$	881	2.43
$remove_{40}$	1420	3.66

Table 5.1: Transformation costs

- the cost of the transformation itself, *i.e.,* matching the *lhs* and replacing it by the *rhs*, possibly creating/removing actors and communication links.
- the total reconfiguration cost including the time taken to pause the execution of the graph, to transform the graph, and to resume the execution until the graph reaches again its steady state and maximal throughput.

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

- $insert_N : I \to O \Rrightarrow I \to A_{x_1} \to ... \to A_{x_N} \to O$
- $remove_N : I \to A_{x_1} \to ... \to A_{x_N} \to O \Rrightarrow I \to O$

  for $N \in \{10, 20, 30, 40\}$ and $t(I) = t(O) = t(A) = 10ms$.

Therefore, we experiment using in total 8 transformation rules which create and remove N actors respectively. The initial graph for all 4 transformation rules $insert_N$ is $I \to O$. For each transformation rule $remove_N$, the initial graph is $I \to A_1 \to ... \to A_N \to O$. For each transformation rule `tr`, the controller consists of a single condition

```
time()=1s :  tr
```

In Table 5.1, the matching and transformation costs are shown. To obtain the values, experiments are performed three times each and the average value is computed.

The matching costs are linear in the size of the rule, due to the fact that there is no backtracking while matching. The transformation costs are linear in the size of the rule, since the number of actors and buffers to construct or

Graph	Reconfiguration cost (ms)
H_5	95
H_{10}	116
H_{15}	132
H_{20}	156

Table 5.2: Reconfiguration costs

destruct is main part of the cost. For standard transformation rules, which involve less than 10 actors, the expected cost is around $1ms$.

The global reconfiguration costs are evaluated by comparing an execution with a dummy transformation ($lhs \Rrightarrow lhs$) with the same execution without any transformation.

To measure the reconfiguration costs, we use the initial graphs H_N as specified below:

- $H_N : I \rightarrow A_1 \rightarrow ... \rightarrow A_N \rightarrow O$ where $N \in \{5, 10, 15, 20\}$, $t(I) = t(O) = t(A) = 10ms$.

Graphs with different number of actors are used to show that the reconfiguration cost is linear in the number of actors.

We use a dummy transformation rule $I \rightarrow A_1 \Rrightarrow I \rightarrow A_1$ and measure the total execution time for the graphs H_N with and without applying that rule once. The difference between these two execution times determines the total reconfiguration costs. Table 5.2 shows that reconfiguration costs are around $100ms$. They are much higher than the cost of the transformation alone.

To summarize, we noticed that the transformation cost is linear in the size of transformation rules. They are in the order of $1ms$ for rules with up to 40 actors. Rules inserting actors are more time consuming than rules removing actors, therefore constructing buffers and actors takes longer than deallocating them. For global reconfiguration costs, they are dependent on the number of edges since tokens need to be consumed before pausing the graph and buffers need to be emptied. We showed that reconfiguration costs are in the order of $100ms$ for graphs with up to 20 actors.

5.2.2 Parallelization impact

We show here how RDF can be used in order to increase the throughput in terms of tokens per second ($\mathcal{H}$) of an application by increasing parallelism.

The dataflow graph G_2 used in this example is shown in Figure 5.3, where as usual, rates are 1 when they are not explicitly mentioned. It consists of the following actors. Src is a source actor that produces an integer number, S is a split actor, A simply copies its input to its output, J is a join actor, and

Snk a sink actor consuming integers. Using busy loops, we set the execution times of Src and Snk to $10ms$ each, S and J to $2ms$ each, and A to 50ms.

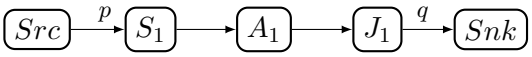


Figure 5.3: Dataflow graph G_2

Two transformation rules tr_3 and tr_4 adding and removing an actor A respectively, are used (Figure 5.4). At time instants $5s$ and $10s$, the transformation rule tr_3 is applied and at time $20s$ and $25s$, the transformation rule tr_4 is applied.

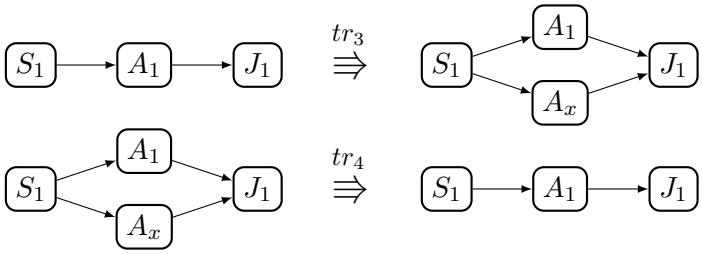


Figure 5.4: Transformation rules tr_3 and tr_4

The experiments were performed using 12 cores. The throughput of the graph is shown in Figure 5.5. After the first and second time applying the transformation rule tr_3, the throughput increases from 20 to 40 and 60 tokens/second. By applying the transformation rule tr_4 twice, the throughput returns to its initial value.

As it was expected, after the first and second transformation, the throughput increases from 20 to nearly 38.7 and 57.4 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, those two rules are sufficient to adapt the application to any required throughput as they can be applied as many times as needed. With other dynamic MoCs such as SADF, we would have to plan for a fixed number of configurations statically.

5.3 Case study

SDF was designed for signal processing applications. Video streaming applications fit well into this model of computation. We show how features of RDF can be used in this domain.

Our case study is an application that captures a video stream from a source, decodes the video stream into images, detects the edges in the de-

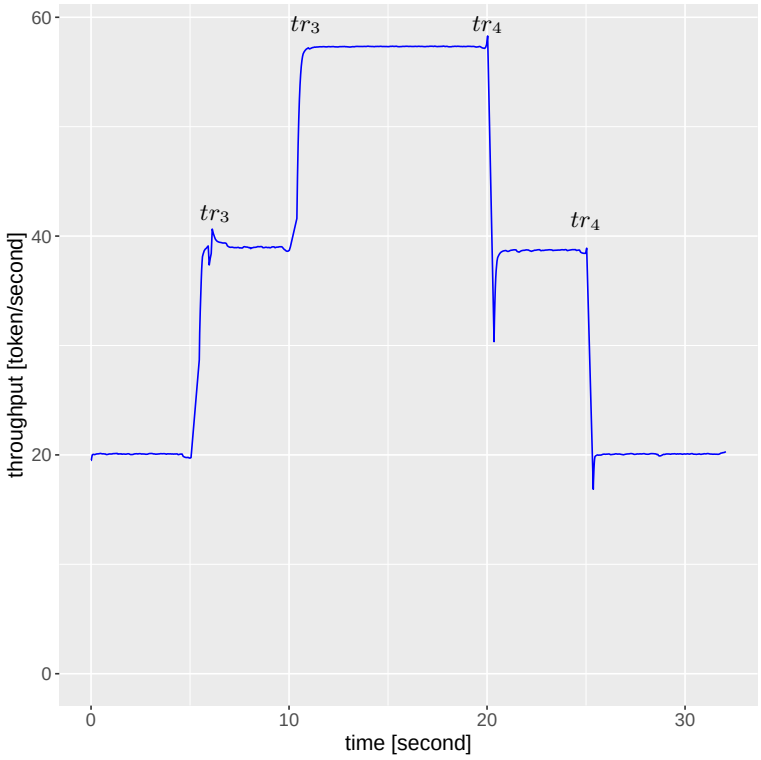


Figure 5.5: Throughput of the graph on 12 cores

coded images, and finally displays the images with a constant frame rate. The frame rate is measured in terms of frames per second.

Dynamic adaptive streaming protocols allow changing the video quality based on network bandwidth. So, if bandwidth allows, the video can be streamed with a higher quality. Suppose that during the streaming, the video quality increases. As a result, the processing of images becomes more time consuming and if the application is not aware of this change, it will not be able to display the images with the desired frame rate. Even when sufficient processing power is available, if the application is unaware of changes, it will not use it.

This application is modeled with RDF and we show how it can avoid resource under-utilization and maintain the throughput. We model the initial application using the graph $V_1 \rightarrow S_1 \rightarrow C_1 \rightarrow J_1 \rightarrow P_1$. Actor V_1 captures the video stream, decodes the stream, and sends the decoded images, actor S_1 distributes the image, actor C_1 receives an image and extracts the edges using an edge detection algorithm, actor J_1 gather the image, and finally actor P_1 displays the received image.

Suppose that V_1 captures the video at 25 frame per second (fps) and P_1 displays the images at 25fps. When the quality of the input video stream

increases, the edge detection takes longer. The images cannot be displayed at 25fps anymore, and the throughput drops to 15fps. By modeling this application using RDF, we can add other edge detection actors C_2, ..., C_n between S_1 and J_1 in parallel. The actor S_1 distributes images between those edge detection actors to improve the throughput of the video display. If the video quality decreases, the edge detection becomes faster and the output buffer of the actor S_1 becomes empty most of the time. In that case, some of those edge detection actors are not necessary and we can remove them.

This application is shown in Figure 5.6. The transformation rule tr_{inc} is applied when the quality of the input video increases and the throughput at the actor P_1 decreases and therefore goes below 20fps. On the other hand, if the quality of the input video decreases, the buffer of the edge (V_1, S_1) remains empty most of the time. The transformation tr_{dec} is applied when the buffer occupancy goes below 10 tokens.

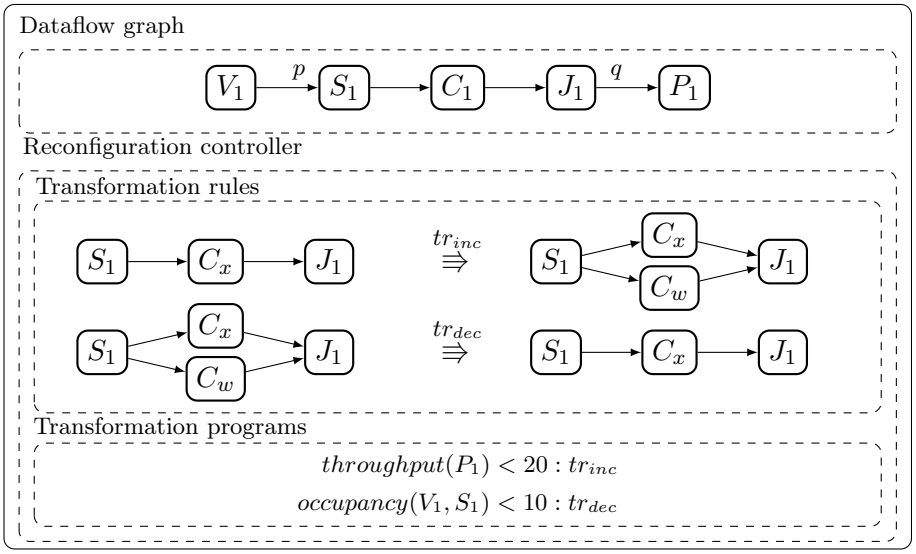


Figure 5.6: An adaptive video processing application

We have implemented this use case partially. Since our input video stream has a constant quality, we simulate the change of quality using a dummy actor whose execution time artificially increases. Moreover, we have only implemented the case where the throughput drops and parallel levels of computations needs to be increased.

The dataflow graph G_3 is shown in Figure 5.7. It contains the same actors as before plus D_1, the dummy actor which produces extra load.

The execution times of actors V_1 and P_1 are $40ms$. In a video streaming scenario where the resolution of the input video may vary, the size of images increases and some image processing actors may take longer than usual. This is modeled using D_1 whose execution time increases after 100 iterations from

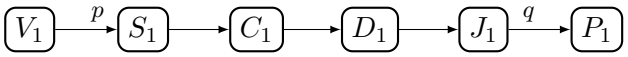


Figure 5.7: Edge detection dataflow graph G_3 with simulation of the changes of video quality

$30ms$ to $60ms$, and then after 200 iterations to $120ms$.

By modeling this application using SDF, we see that the throughput decreases from $\frac{1}{40 \cdot 10^{-3}} = 25$ tokens per second to $\frac{1}{60 \cdot 10^{-3}} = 16.6$ and finally to $\frac{1}{120 \cdot 10^{-3}} = 8.3$ (Figure 5.8).

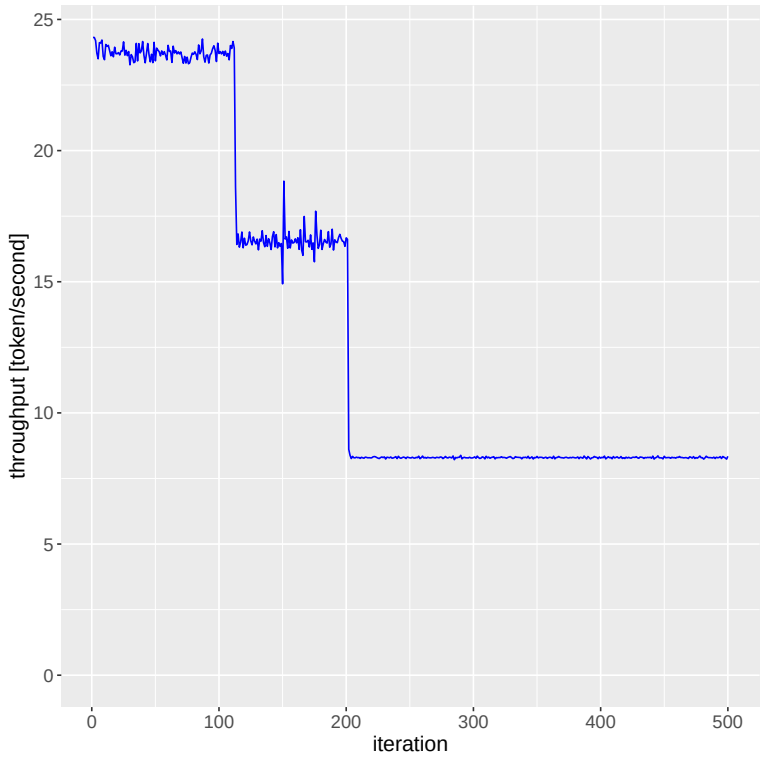


Figure 5.8: Throughput of the edge detection application in SDF

If we model the same application using RDF, we can reconfigure the graph when the throughput decreases and add a parallel level of computation between the split and join and therefore process more tokens (images) per second. The transformation program is specified as:

```
throughput(P1) < 20 :  tr5;
```

The transformation rule tr_5, shown in Figure 5.9, increases the parallelism of computations. The execution time of actors D_1 is increased from

$30ms$ to $60ms$ at iteration 112, the condition becomes true and tr_5 is applied. Edges are detected using two parallel levels of computations (C_1, D_1) and (C_2, D_2). Then again the execution time of actor D_1 and D_2 is increased from $60ms$ to $120ms$ at iteration 202, the condition becomes true and C_3 and D_3 are added. Actor D_1, D_2, and D_3 always share the same execution time. By applying the transformation rule tr_5 twice, we obtain the resulting graph G_4 shown in Figure 5.10.

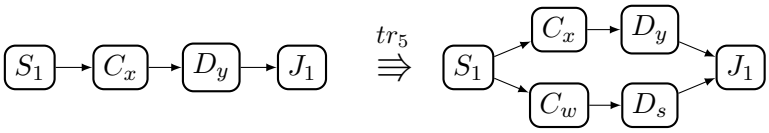


Figure 5.9: Transformation rule tr_5 to add one parallel level of computation in the edge detection application

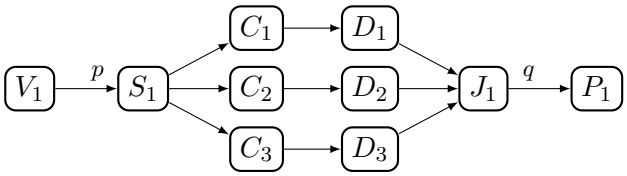


Figure 5.10: The resulting graph G_4 after applying the transformation rule tr_5 to G_3 twice

In Figure 5.11, we see the throughput measured at the sink actor. In the beginning, the actor V_1 is the actor with the highest load, for which the value of $sol(i) \cdot t(i)$ is the largest. According to formula 4.13, the throughput is calculated as $\mathcal{T}_G = \frac{1}{0.040} = 25$ iterations/second. Since one token is consumed per iteration, the token production is $\mathcal{H}_G = 1 \cdot \mathcal{T}_G = 25$ tokens/second. Because of the overheads, the actual value is 23.5.

When the execution time of actor D_1 increases to $60ms$ at iteration 112, the actor D_1 becomes the slowest, (*i.e.,* $sol(i) \cdot t(i) = 0.06$). So the throughput is determined by actor D_1, and $\mathcal{T}_G = \frac{1}{0.06} = 16.6$ iterations/second and $\mathcal{H}_G = 1 \cdot \mathcal{T}_G = 16.6$ tokens/second.

At iteration 112 when the throughput has decreased, the transformation is applied and the throughput is again determined by actor V_1 (*i.e.,* $sol(V_1) \cdot t(V_1) = 0.08$ whereas $sol(D_1) \cdot t(D_1) = 0.06$). The execution time of actor V_1 is $40ms$ and its solution after the reconfiguration is 2. Therefore, $\mathcal{T}_G = \frac{1}{2 \cdot 0.040} = 12.5$ iterations/second and $\mathcal{H}_G = 2 \cdot \mathcal{T}_G = 25$ tokens/second. Note that because of overheads, the throughput is around 24 tokens/second in practice.

When the execution time of actors D_1 and D_2 increases to $120ms$ at iteration 202, the throughput decreases to $\mathcal{H}_G = 2 \cdot \frac{1}{0.12} = 16.6$ tokens/second. The condition becomes true and the transformation rule is applied

for the second time and the throughput returns to $\mathcal{H}_G = 3 \cdot \frac{1}{3 \cdot 0.040} = 25$ tokens/second.

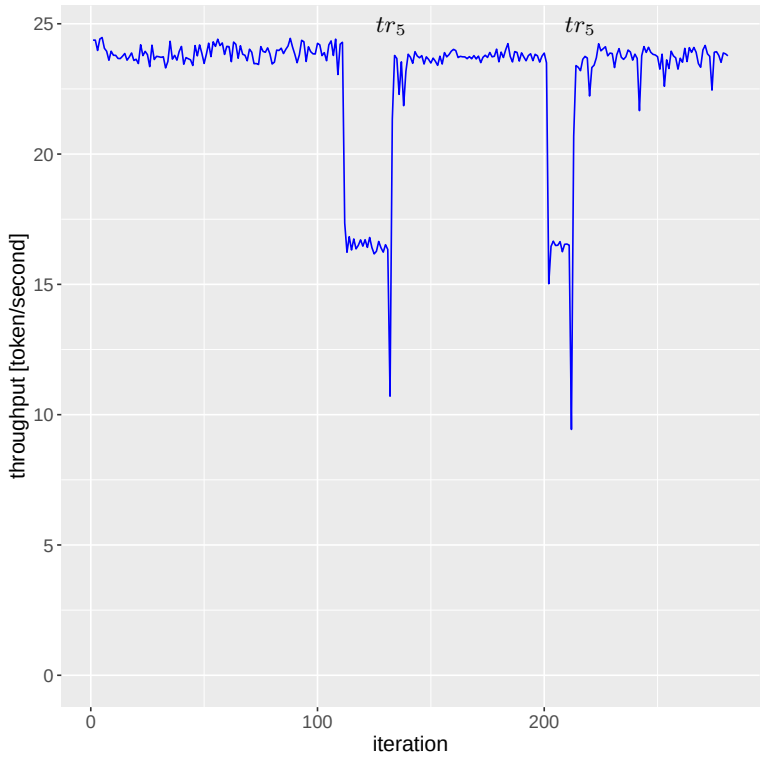


Figure 5.11: Throughput of the edge detection application in RDF

In this case study, we saw how transformations triggered by conditions on throughout can increase the parallelism of computations in a video processing example and how the throughput can be maintained at a desired value.

As for the quality of experience, we noticed in this case study that the reconfiguration costs are small and seamless. Transitions from one graph to another are smooth and the user is not disturbed by the transformations.

5.4 Summary

In this chapter, we described our prototype implementation of RDF. In the first part, we explained how actors are executed, how the reconfigurations take place, and what conditions are used to trigger transformation rules. We also explained solutions for pattern matching and dynamic actor placement.

In the second part, we presented the experiments we have conducted using the prototype. We started by describing the architecture and performance metrics that were used, then we evaluated reconfiguration costs and

showed with an example how RDF can be used to improve dynamically the throughput of a graph.

In the third part, we detailed a case study in which RDF is used for a small concrete application. The application retrieves a video stream, decodes and extracts edges from images, and displays the result. If the computation becomes more time consuming, RDF can cope with the decrease of throughput by changing the parallel levels of computation in order to maintain the throughput at the desired level. The reconfigurations are barely noticed in the displayed video stream.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

An SDF dataflow graph is not adaptive, that is, its topology cannot change at run-time and its rates are fixed during the execution. Some MoCs (such as PSDF) have addressed this problem by replacing the static integer rates with parameters so that they can be changed dynamically. Some MoCs (such as BPDF) have gone further by allowing edges to be enabled and disabled using parameters in such a way that the topology of the graph can be changed to some extent. Some other MoCs (such as SADF) allow dynamic change of the topology of the dataflow graphs by describing an application as a small set of SDF graphs from which the executing graph can be selected when a reconfiguration is needed. Although SADF can handle applications which need to change the topology of graphs, it cannot be used when the number of required configurations is unbounded or even very large.

In this thesis, we proposed RDF, an MoC that allows an unbounded number of reconfigurations. For this purpose, RDF introduces the concept of transformation rule. Conceptually, a transformation rule is a graph rewrite rule that replaces a subgraph of the dataflow graph by another subgraph and therefore modifies the topology of the dataflow graph to which it is applied. RDF dataflow graphs are similar to SDF graphs except that each actor has a type and port are explicit. Actor types allow an unbounded number of new actors to be created during the execution as instances of those types. Explicit ports allow the transformation rules to choose specific ports for connecting actors.

Each RDF application consists of an initial dataflow graph and a reconfiguration controller. The initial graph consists of a set of actor types, a set of actors, a set of edges, and a function returning the initial tokens of each edge. An actor type defines the number of input and output ports of that type along with the rate of each of its ports. The controller has a reconfiguration program that specifies the transformation programs and the

conditions to trigger them. A transformation program is a combination of transformation rules and a condition is a boolean expression. The conditions and the transformation programs determine respectively *when* and *how* the dataflow graph must be reconfigured.

A transformation rule consists of a left-hand side (*lhs*) and a right-hand side (*rhs*). Its *lhs* is a pattern graph, that is, a graph containing variables for the name, type, rate, and ports of actors. An *lhs* matches a subgraph of the dataflow graph if a subgraph is found in the graph which is isomorphic to the *lhs* and if a one-to-one mapping from the set of variable of the *lhs* to the set of constant values of the dataflow graph is found such that each variable in the *lhs* finds a unique value. Once a rule, that is, the *lhs* of the rule matches a subgraph of the dataflow graph, its *rhs* pattern graph replaces the matched subgraph after substituting its variables by constant values.

The graph obtained from applying a transformation rule to a graph must be a valid dataflow graph. A transformation rule must respect the following conditions in order to be valid: (1) The number of input and output edges and rates of each actor must correspond to its type. (2) When an actor is removed, meaning that it is present in *lhs* but not in *rhs*, its type must be known and all its edges must be present in the *lhs* of the transformation rule, otherwise this actor would be suppressed while some of its edges would remain in the graph. (3) When an actor is created, meaning that it is present in *rhs* but not in *lhs*, its type must be known, so that statically we can determine whether it is correctly connected.

Static analyses have been proposed in order to ensure that the important properties of the dataflow graphs, that is consistency and liveness, are preserved by any valid transformation rule. Moreover, the RDF graphs are connected and the transformation rules must preserve its connectivity. The condition for connectivity is that the *rhs* of the transformation rule must be a connected graph. The condition for consistency is that the solutions of actors that remain in the graph do not change and that new actors created by the rule have integer solutions. The condition for liveness is that if there is a path between two actors in the *rhs* of the rule, then there must be also a path between those two actors in the *lhs*. This condition ensures that no new cycle is created by a transformation rule and that the graph remains live. Therefore, an RDF application with an initial graph which is connected, consistent, and live and with a set of transformation rules respecting those conditions always produce connected, consistent, and live graphs.

We have studied the impact of the transformation rules on the performance of the RDF graph. Specifically, we have focused on the two most relevant performance metrics for dataflow applications: the latency and the throughput. We have identified a class of transformation rules that are of practical interest, for which we have computed bounds of their impact on latency and throughput.

We have implemented the RDF MoC in a prototype. In the prototype, a

transformation program consists of a single transformation rule. Therefore, the reconfiguration program is a set of rules triggered by conditions. A condition is a boolean expression measuring throughput, latency, or buffer occupancy. The reconfiguration controller implements the program of the RDF application: In parallel with the execution of the graph, the controller checks which transformation rules match the current graph. It also regularly checks whether the conditions corresponding to the matching rules hold. The first matching rule in the program whose condition holds is applicable. Once an applicable rule is found, the controller applies its corresponding transformation rule by pausing the graph, reconfiguring the graph, and finally resuming the execution. After each reconfiguration, the controller again checks for the matching rules using a pattern matching algorithm. The pattern matching is a costly operation and in order to reduce the costs, we have proposed simple constraints.

Experimentations showed that the reconfiguration costs are small and that RDF can be used in practice. We also have shown that RDF can be used to change the level of parallelism in a dataflow graph and therefore to improve the throughput of the graph. Finally, a simple case study is presented. The application consists of actors for decoding a video stream, processing the images produced by the decoder, and finally to encode the processed images. In this case study, the execution time of one of the actors in dataflow graph increases. In SDF, that would result in a loss of throughput. By using RDF, the loss of throughput is detected by one of the conditions and a transformation rule is applied to increase the level of parallelism of the graph allowing to maintain the throughput at the right level.

6.2 Future work

Several extensions of RDF can be considered to make it more expressive or analyzable and its implementation can be studied for other architectures. We may also find other applications which can be modeled using the RDF MoC. We discuss some possible extensions and applications for future works.

Extensions

The rates in RDF are constant, like in SDF. An interesting extension of RDF would be to parameterize the rates as in BPDF and PSDF. The extension would require to extend the static analyses for consistency and liveness in order to take parameters into account.

The static analyses of the RDF to ensure preserving consistency and liveness enforce some constraints causing some transformation rules to be rejected. We should study how these static analyses could be modified to cope with (1) non-connected graphs and transformation rules, and (2) transformation rules involving cyclic graphs and graphs with initial tokens. A

subject of future work would be to find new conditions for preserving the consistency and liveness while keeping the constraints on the dataflow graph and the transformation rules to a minimum.

The performance analyses we studied address latency and throughput. We have focused on a class of useful transformation rules. In future, we could consider a larger class of transformation rules and to find the impact of those rules on latency, throughput, and possibly other performance metrics such as resource utilization.

RDF has been implemented for a multi-core architecture. The implementation could be extended to more general architectures with multi-servers communicating through messages. In this case, the controller would need to have information about the available servers and processors, and a placement heuristic would map the actors in order to optimize the communication costs. In such an implementation, the actors would need to communicate through messages on the network. Since the network quality changes, the controller should probe the network and decide how to re-map actors. We have studied the use of RDF in a single case-study. More use-cases would help to find other useful extensions.

Applications

Two application domains seem promising for RDF: computer vision and machine learning.

Computer vision algorithms can be used for many applications such as tracking objects in a video, detecting objects in images and so on. Those applications use graphs made of several multimedia processing filters and SDF look suitable for them. OpenCV [14] is an open source software library offering functions for performing computer vision algorithms. Different reconfiguration requirements of the applications described in this library should be studied. RDF could be used to extend the library to handle dynamic reconfigurations. Moreover, using dataflow, those applications can be executed efficiently on multi-processors and, in an environment where the resources change, RDF and its potential extensions might be useful.

Machine learning algorithms are used in many different areas such as automatic translations of texts, automatic recognition of content of speech, and automatic recognition of objects in images. Those algorithms usually use neural networks, where computations are described by acyclic graphs where each vertex is a neuron which has a number of inputs and outputs and performs a specific operation. Dataflow MoCs look suitable to express and implement such applications.

The neural networks are usually large and require efficient scheduling and placement. A research direction would be to study applications of dataflows for the efficient parallel implementation of such networks. In some applications, the topology of the computation graph depends on the input data, and

therefore the topology of the graph must change dynamically. RDF may be applied to model and implement dynamic neural networks.

In [30], the dynamic neural networks are divided into three categories: (1) sample-wise dynamic networks adapting the topology based on input data, (2) spatial-wise dynamic networks adapting the topology based on spatial information in spatial data such as images, and (3) temporal-wise networks adapting the topology based on temporal information in sequential data such as texts. In all those categories, networks of different sizes are to be used for different inputs. Often those networks are similar to each other and differ only in the number of layer of neurons. In such cases, a few RDF transformation rules can add and remove layers of neurons and produce many different networks.

More research is needed to asses if RDF can be used to model the applications in the domains of computer vision and machine learning where reconfigurations are required and to implement them more efficiently.

Appendix A

Implementation Details

We describe here the prototype we have implemented in detail. In section A.1, we describe how an RDF application can be specified in a high level language, that is, how the initial dataflow graph and the reconfiguration program are specified. We explain how the prototype can be used having been provided by some off-the-shelf actor types. In section A.2, we explain how a new actor type can be implemented. In section A.3, we present the details of the implementation by explaining the attributes and functions of all structures used in the implementation. The prototype is implemented in C++ language.

A.1 Specifications

We explain in this section how the initial dataflow graph and the reconfiguration program of an RDF application are specified in a high-level language.

Dataflow graph. In order to specify an RDF graph, we use a subset of the Dataflow Interchange Format (DIF) [44, 32]. DIF is a general specification format for arbitrary dataflow models. Its objective is to facilitate reusing dataflow graphs from different implementations of dataflow-based tools, by providing a common and yet extensible semantic to represent dataflow graphs. It captures all necessary information for modeling a dataflow graph along with information required for static and performance analyses, while hiding the details of implementation of computations and communications.

The DIF specifies a dataflow application using a number of nested blocks. Each block contains parts of the attributes of the application. The outermost block contains the name of the MoC and the name of the application (*e.g.,* `rdf canny`). The topology of the dataflow graph is specified in the `topology` block. The production and consumption rates are specified in the `production` and `consumption` blocks. If rates are not specified they take the default value 1. For instance, the dataflow graph $V_1 \rightarrow S_1 \rightarrow C_1 \rightarrow D_1 \rightarrow$

$J_1 \rightarrow P_1$ is specified as below.

```
rdf canny {
  topology { nodes = V1, S1, C1, D1, J1, P1;
    edges = e1(V1,S1), e2(S1,C1), e3(C1,D1),
            e4(D1, J1), e5(J1,P1); }
  production { e1 = 1; e2 = 1; ... }
  consumption { e1 = 1; e2 = 1; ... }
}
```

Attributes of actors are specified in the `actor` block of DIF. These attributes are metadata that the actor needs during its execution. The computational behavior (*i.e.,* the type) of all actors must be specified by the `computation` keyword followed by the type of the actor. Other attributes include the name of the file an actor needs as its input or output, calibration values for image processing tasks, etc. We can also assign a port to an edge. For instance, the assignment `output1 = e2;` as an attribute of the actor `S1` connects its output port `output1` to the edge `e2`.

```
rdf canny { ...
  actor V1 { computation = VideoCapture; file_name = in.mp4; }
  actor C1 { computation = Canny; }
  actor S1 { computation = Split; output1 = e2; } ...
}
```

There are certain parameters that are required by the reconfiguration controller. For instance, the controller needs to know whether the user needs to print all log messages on the screen or not. The log messages contain information concerning the execution of actors, that is, at what time a firing is performed, at what iteration an actor is, what messages an actor has printed, at what time a reconfiguration is performed, how the topology of the graph is changed during a reconfiguration, and so on. Such information can be used to debug an application. Another parameter concerns the scheduling policy of the dataflow graph. The scheduling either is performed using the default Linux scheduler, or the user manually sets the computation cores on which each actor must execute.

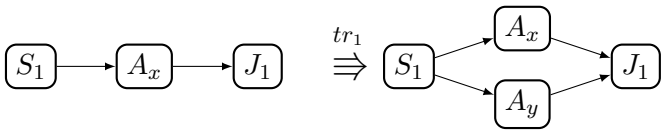
Parameters are specified in the `parameter` block as below. When the value of the variable `logging` is set to true, the debugging messages are printed out. By setting the value of the variable `scheduling` to false and setting the `cpu` attributes actors, processing cores are assigned to them.

```
rdf canny { ...
  actor V1 { cpu = 1; ... }
  parameter { logging = true; scheduling = false; }
}
```

Transformation rules. We have extended DIF to specify RDF applications. Besides the initial graph, an RDF application must also specify the set of transformation rules which are used by the reconfiguration program. A transformation rule is specified in the `rule` block, and the left-hand side and the right-hand sides of the rules are specified in the `lhs` and `rhs` blocks.

A constant name starts with an uppercase (*e.g.,* `S1`) and a variable starts with a lowercase (*e.g.,* `x`). During the pattern matching, each variable is substituted by an actor name, an actor type, or a rate. New actors are specified by variables and are given fresh names when added to the graph.

The following rule



is specified in DIF as:

```
rdf canny { ...
  rule tr1 {
    lhs {
      topology { nodes = S1, x, J1;
        edges = e1(S1, x), e2(x, J1); }
      actor x { computation = A; } ...
    }
    rhs {
      topology { nodes = S1, x, y, J1;
        edges = e1(S1, x), e2(x, J1), e3(S1, y), e4(y, J1); }
      actor y { computation = A; } ...
    }
  }
}
```

Reconfiguration program. The controller monitors the execution of the dataflow graph. It has a reconfiguration program consisting of pairs of conditions and transformation rules. It regularly checks in order the conditions in program. As soon as one condition is evaluated to true, its corresponding transformation rule is applied to the graph. The reconfiguration

program is specified in the `main` block as a pair of conditions and transformation rules (*i.e.,* `condition : rule`). A condition consists of four parts. It is of the form of `actor.metric operator value`. A performance `metric` at a given `actor` is measured and compared using a comparison `operator` with a certain `value`.

For instance, we can measure the throughput at a given actor and apply a transformation only if its throughput is less than, equal to, or greater than a certain value. In the following example, the `throughput` at the actor `P1` is compared with the value 40. If the throughput is less than 40 tokens per second, then the transformation rule `tr2` is applied. Each actor has also a `timer` that measures for how long an actor has been executed. In the following example, if the `timer` of the actor `V1` reaches the value $5000ms$, the transformation rule `tr1` is applied.

```
rdf canny { ...
  main {
    V1.timer = 5000 : tr1;
    P1.throughput < 40 : tr2;
  }
}
```

A.2 Actor types

In the previous section, we saw how an RDF application can be specified. There are built-in types of actors which can be used to build an application. In this section, we explain how new types can be developed.

A new actor must inherit from the `Actor` class and define a set of ports along with their data types during the construction. Primitive token types `Int` and `Str` for two-byte integers and strings are provided. The token type `Mat` for matrices containing image structures of OpenCV library can also be used.

At initialization, actors can read the attributes set in the dataflow graph specified as DIF. Those attributes include the address of the file a video reader needs to read, or the threshold values an actor needs for its internal algorithms. During the execution, each actor reads from its input ports and write to its output ports. Finally, once an actor finishes its execution, it destroys its ports and it is itself destroyed by the controller.

Construction. An actor can create input and output ports using the functions `createInputPort` and `createOutputPort`. These ports can be connected only to one other port. Variable arity actors have ports which can connect to multiple ports. The function `createInputPortVector` is used for creating an input port of variable arity. Similarly, for creating an output port of variable arity the function `createOutputPortVector` is used .

The following code snippet illustrates the construction of the actor type `Canny`. This type has one input and one output port both of type `Mat`. It receives an image, extracts the edges from the image using the canny edge detection algorithm, and finally writes the resulting image on its output port. For its edge detection algorithms, the actor has an attribute `threshold` specifying the granularity of the edge detection. Its value can be set by a user as an attribute in DIF specification of the application.

```
class Canny : public Actor {
private:
  InputPort<Mat> * input;
  OutputPort<Mat> * output;
  int threshold;
public:
  Canny(const string& name) : public Actor(name) {
    input = createInputPort<Mat>("input");
    output = createOutputPort<Mat>("output");
  }
}
```

Initialization. During the initialization phase, the actor reads its attributes. The function `propEmpty` checks if the value of an attribute (property) is empty. To retrieve the values of attributes, two functions `getProp`, `getPropInt`, and `getPropFloat` are provided (for string, integer, and float).

In the example below, if the attribute `threshold` is set, then the actor reads its value, otherwise it sets it to a default value.

```
class Canny : public Actor { ...
  void init() {
    if (!propEmpty("threshold"))
      threshold = getPropInt("threshold");
    else
      threshold = 100;
  }
}
```

Execution. During the execution, the actor needs to read and write. The functions `consume` and `produce` are used respectively for reading from input ports and writing to output ports. Once the `consume` function is called, a number of places in the buffer corresponding to the rate of a given port is blocked. The actor fills those places and release them afterwards. The `release` function is called for releasing the buffer of a given port. By

releasing a port by all consumers, the producer is allowed to produce a new token on that buffer's place. Similarly, once the `produce` function is called, a number of places in the buffer corresponding to the rate of a given port is blocked for the producer and no consumer is allowed to read from those places until the place is released using the `release` function.

The functions `consume`, `produce`, and `release` can be called on normal ports (*i.e.,* `InputPort` and `OutputPort`) as well as variable arity ports (*i.e.,* `InputPortVector` and `OutputPortVector`). By calling `consume` (resp. `produce`) on a variable arity port, a vector v of vectors u_i is returned to be read by a consumer (resp. to be written by a producer). Each element of the vector v corresponds to one of the ports (i) and each element u_i is itself a vector of tokens whose size is the rate of the port i.

Once a consumer blocks places on a buffer, it can get references of the tokens by invoking `get` function. Similarly, a producer blocks places on the buffer and sets the values of those places by invoking the `set` function. The `clone` returns a copy of the value of a given place of a buffer. For variable arity ports, `get` and `set` must be called on the elements of the vector returned by `consume` and `produce`.

The example below illustrates how a `Canny` actor reads from its input and writes to its output. It blocks one place on its input port and one on its output port and retrieves the references `in` and `out`. Using the function `get`, it gets the image on the buffer. By invoking the `cv::Canny` function, it extracts the edges. It fill the output buffer by calling the `set` function. Finally, the actor releases the buffers by calling `release` on both its input and output ports and let its predecessor to produce and its successor to consume tokens on their corresponding buffers. The type `cv::Mat` is used to store images in OpenCV and the type `Mat` is a token type holding `cv::Mat`.

```
class Canny : public Actor { ...
  void run() {
    cv::Mat image, edges;
    vector<Mat *> in = consume(input);
    vector<Mat *> out = produce(output);
    image = *in[0]->get();
    cv::Canny(image, edges, threshold, 2*threshold);
    out[0]->set(edges);
    release(input);
    release(output);
  }
}
```

Destruction. To destroy ports, the `destroyPort` function is called. As a result, all the resources acquired by the port are released. This function

is called when the actor finishes its execution. The controller destroys the actor and the actor destroys its ports before its destruction.

```
class Canny : public Actor { ...
  ~Canny() { destroyPort(input); destroyPort(output); }
}
```

A.3 System structure

The structure of RDF prototype consists of these main classes: `Actor`, `Edge`, `Buffer Port`, `Token`, `Graph`, `Rule`, and `Controller`. In this section, we describe those classes and their attributes and functions in a high level.

Actor

An actor has a `name` and a `type`. It has a list of input ports (`inputPorts`) and output ports (`outputPorts`) accessed by their name. It also has a `solution`, the number of its current firing (`stepno`), the number of its current iteration (`iterno`) and the number of its current firing in an iteration (`fireno`). We have the following the equality: `stepno = (iterno-1) * solution + fireno`. The boolean `paused` indicates that the actor must pause for a reconfiguration. The value of the `iter_max` is set by the controller to indicate until which iteration the actor must continue executing before pausing for a reconfiguration.

```
class Actor {
  string name, type;
  map<string, IPort*> inputPorts;
  map<string, OPort*> outputPorts;
  int solution, stepno, iterno, fireno;
  bool paused; int iter_max;
}
```

Initialization and execution. Each concrete actor has to implements the virtual function `init` which is called to initialize the actor. At the initialization, an actor sets the attributes it need during the execution. When an actor is reinitialized after a reconfiguration, the function `reinit` is called. It may be the case that the attributes an actor needs change. For instance, a rule may change the thresholds used in algorithms of an actor at each reconfiguration. If this function is not implemented by the concrete actor, by default the `init` function is called. The virtual function `run`, which defines the behavior of an actor, needs to be implemented by each concrete actor.

```
virtual void init() = 0; virtual void reinit();
virtual void run() = 0;
```

Creating ports. The function `createInputPort` creates a new input port and adds it to the list of input ports. The type of tokens on the created input port is specified by `T`. The type `T` must be a subclass of the `Token` class (see section Token). Similarly, the function `createOutputPort` creates a new output port.

```
template <typename T>
InputPort<T>* createInputPort(string name);
OutputPort<T>* createOutputPort(string name);
```

Consumption and production. The function `consume` consumes as many tokens as the consumption rate of a given input port. It first locks the buffer of the port by calling `port.lock()`. Then it returns a number of places equal to the consumption rate of the `port` from its buffer by calling `port.get()`. Similarly, the function `produce` produces as many tokens as the production rate of a given output port. Once it has read from or written to a buffer, the actor uses the function `release` to unlock the buffer of its input or output port.

A user can set an attribute of a source actor so that it terminates its execution at a certain moment (for instance, when it reaches a certain iteration). Once that source actor reaches the last token, it sets the status of its produced tokens to end-of-stream by calling the function `setEOS`. Other actors terminate their execution once they read a token with such status.

```
template <typename T>
vector<T*> consume(InputPort<T> * port);
vector<T*> produce(OutputPort<T> * port);
void setEos(OutputPort<T> * port);
void release(InputPort<T> * port);
void release(OutputPort<T> * port);
```

Pausing, resuming, and running actors. Using the function `pause` before each reconfiguration, the controller pauses the actor. This function sets the values of `paused` and `iter_max` which are protected by mutexes. Those values are read in the function `runActor`. The function `resume` is used by the controller to resume the execution of the actor. It also manipulates the value of `paused` and notifies the actor to wake up.

The function `runActor` executes an actor by executing its iterations until an end-of-stream is reached. The status of an actor is set to end-of-stream once it reads a token with such status or when the actor itself sets the status of a token to end-of-stream. Before execution, the actor checks whether all its ports are connected. During the execution, if the function `pause` has set the value of the `paused`, the actor waits until woken up using a signal sent by `resume` function. The function `startRun` creates a thread to start running an actor. If the user has chosen the computation core, then the thread is put on the specified core. The system function `pthread_setaffinity_np` is used for assigning the thread of an actor to a core. The function `waitRun` waits for the thread which is created in `startRun`. The controller uses this function for synchronizing actors.

```
void pause(); void resume();
void startRun(); void waitRun(); void runActor();
```

Non-functional metrics. The function `getTime` gets the value of the actor's timer. The timer starts counting when the actor starts executing and counts in milliseconds. The function `getThroughput` gets the value of the actor's throughput in one iteration. The difference between the start time of the last firing of an actor in the iteration i and the last firing in the the iteration $i-1$ is the period of the iteration i. The throughput of an iteration is the inverse of the period of that iteration. The result is returned in terms of iteration per seconds. The function `getLatency` gets the value of the actor's latency in one iteration. The difference between the end time of the last firing of an actor in iteration i and the first firing in the iteration i is the latency of the iteration i. The result is returned in terms of milliseconds. The function `getOccupancy` returns the number of tokens in the buffer of a given `port`. All these functions are used by the controller in the function `getApplicableRule` for verifying the conditions.

```
int getThroughput(); int getLatency();
int getOccupancy(string port); int getTime();
```

Edge

An edge has a `name`, a source actor `src_actor`, a sink actor `snk_actor`. It has the name of the port of its source actor (`src_port`) and the name of the port of its sink actor (`snk_port`) as well as their rates, *i.e.,* `src_rate` and `snk_rate`.

```
class Edge {
  string name, src_actor, snk_actor, src_port, snk_port;
  int src_rate, snk_rate;
}
```

Buffer

A buffer is a circular list of data. The output ports are the owners of buffers and the input ports are the users. The output ports create buffers while the input ports have only a reference to the buffers A buffer has an array of `tokens` of the generic type `T` with a specified `size` .

```
template <typename T>
class Buffer { T ** tokens; int size; }
```

Accessing the buffer. The constructor `Buffer<T>` creates an array of a given size. The function `at` is used to read an element of the buffer at a given index.

```
  Buffer<T>(int s); T * at(int idx);
```

Port

Two interfaces for input and output ports inherit from the class `Port`. Each port has a `name` and a `rate`.

```
class Port { string name; int rate; }
```

Input port. The input port inherits from the `Port` class. Each input port has a generic type `T` which is the type of the tokens on its buffers. An input port has a reference to the `buffer` created by an output port and an `index` for accessing the buffer.

```
template <typename T>
class InputPort: public Port {
  Buffer<T> * buffer; int index;
}
```

The input port has a constructor to create the port with a given `name`. The function `setBuffer` sets the buffer of the input port. When connecting two ports through an edge, an output port uses this function to give a reference of its buffer to the input port. The function `unsetBuffer` clears the buffer's pointer of the input port. When removing an edge and disconnecting two ports, an output port uses this function to disconnect itself from the input port.

```
InputPort<T>(string name);
void setBuffer(Buffer * buf); void unsetBuffer();
```

The function `lock` calls `consumerLock()` on `buffer.at(index)` and locks one place in the buffer. The function `unlock` unlocks the last place in the buffer and increases the value of index. The function `get` gets the reference of a given token from the port by calling `buffer.at(index)`.

```
void lock(); void unlock(); T * get();
```

Output port. Similar to the input port, the output port inherits from the `Port` class. It has a generic type `T` which is the type of the tokens of its buffers. An output port has a `buffer` of the generic type `T` with a specified `size`. It has also `index` for accessing the buffer.

```
template <typename T>
class OutputPort: public Port {
  Buffer<T> * buffer; int size, index;
}
```

The output port has a constructor `OutputPort<T>` to create the port with a given `name`. The function `setBufferSize` sets the size of the buffer of the port. The function `connectPort` connects the output port to a given input port. It sets the buffer's reference of its associated input port by calling `in.setBuffer(buffer)`. The function `disconnectPort` detach the connection between the output port and a given input port. It also clears the buffer's reference of its associated input port using `in.unsetBuffer()`. Similar to the input port, the functions `lock`, `unlock`, and `get` are used to lock, unlock, and get one place from the buffer respectively.

```
OutputPort<T>(string name); void setBufferSize(int s);
int connectPort(Port* in); int disconnectPort(Port* in);
void lock(); void unlock(); T * get();
```

Token

Each token has a status. The enumeration `Status` lists the different statuses of a token. The status `OK` indicates that the token has been read or written correctly, `ERROR` indicates that an error has occurred while reading or writing the token, and `EOS` indicates that the end-of-stream has reached. A source actor can set the status of tokens to end-of-stream to terminate the execution.

```
enum Status { OK, ERROR, EOS };
```

The `Token` class has a `data` of generic type `T` and a `status`. Concrete token classes must inherit from this class.

```
template <typename T>
class Token { Status status; T * data; };
```

Manipulating tokens. The function `get` returns a pointer to the data of the token. The virtual function `set` sets the data. Its implementation uses a simple copy of the memory. A concrete token, implemented as a sub-class of the `Token` class, can override this function for other kinds of copying data. The virtual function `clone` returns a cloned copy of the data. This function can also be overridden by concrete tokens. A token is cloned when an actors needs to manipulate the data of the token locally without impacting the data on the buffer.

```
  T * get(); virtual void set(T& data); virtual T clone();
```

Synchronizations. The functions `consumerLock` and `consumerUnlock` are used to lock and unlock the token for the consumers and the functions `producerLock` and `producerUnlock` to lock and unlock the token for the producer. Input and output ports use these functions on each place of a given buffer.

```
  void consumerLock(); void consumerUnlock();
  void producerLock(); void producerUnlock();
```

Graph

The class `Graph` is made of the `name` of the dataflow graph, the set of all `actors` and `edges` of the graph accessed by their names, and a list of parameters `params`. For instance, one of the parameters is `scheduling`. If it is set

to false, then Linux chooses the placement of actors on cores, otherwise if it is set to true, then the user chooses the core for each actor. When graphs are used in rules, they may contain variables. When the name or type of a given actor is variable, it starts with a lowercase letter. The production and consumption rates can also be variables starting with a lowercase letter.

```
class Graph {
  string name;
  map<string, Actor *> actors;
  map<string, Edge *> edges;
  map<string, string> params;
}
```

Connectivity, consistency and liveness. The function `connected` returns true if the graph is connected. The function `solve` finds the solutions of all actors in the graph. It returns 0 if the graph is consistent, and -1 if inconsistent. The function `sas` returns a list of single appearance schedules (SASs) of the graph. If no SAS is found, an empty list is returned.

```
  bool connected(); int solve(); vector<string> sas();
```

Connecting and disconnecting actors. The function `connectActors` create an edge between two actors in the dataflow graph with the corresponding rates. The function `disconnectActors` disconnects two actors. It is used when an edge between to actors is to be removed and each input port has to clear its pointer to the buffer of its associated output ports. The parameters `src` and `snk` are the source and the sink actors of the edge. The parameter `edge` gives the name of the edge. The parameters `p` and `c` denote the production and consumption rates of the edge respectively.

```
  void connectActors(Actor * src, Actor * snk,
                     string edge, int p, int c);
  void disconnectActors(Actor * src, Actor * snk, string edge);
```

Initialization and termination. The function `init` initializes the dataflow graph, by calling the `init` functions of each actor. Actors initialize the attributes they need at execution in this phase. The function `check_eos` returns true if any of the actors reaches the end-of-stream.

```
  void init(); bool check_eos();
```

Pausing and resuming. The function `pause` pauses the dataflow graph according to the following steps : (1) the value of the attribute `paused` is set to true for all actors, (2) the iteration number of all actors is retrieved and among those iteration numbers, the maximum value `iter_max` is computed, and (3) all actors are requested to resume until the iteration `iter_max` and pause. The function `resume` resumes all actors of the dataflow graph by setting the value of their `paused` attribute to false and signaling them.

```
int pause(); void resume();
```

Rule

A rule has a `name` and a reference to the current dataflow graph `g` on which the the rule should be applied. It has a left-hand side graph `lhs` and a right-hand side graph `rhs`. The rule also has the resulting graph after applying the rule.

```
class Rule { string name; Graph * g, lhs, rhs, res; };
```

Static analyses. The function `staticConditions` returns true if all static conditions hold. The functions `connectivity`, `consistency`, and `liveness` return true if the rule preserves connectivity, consistency, and liveness respectively. The function `verify` checks whether the transformation is valid, that is, all static conditions hold and it preserves connectivity, consistency, and liveness.

```
bool connectivity(); bool consistency(); bool liveness();
bool staticConditions(); int verify();
```

Applying a rule. The function `matching` returns true if the rule is matched in the current dataflow graph. The names and types of actors and production and consumption rates can be parametric. The function `apply` applies the rule on the current dataflow graph. It replace the sub-graph corresponding to the *lhs* of the rule by its corresponding *rhs* after substituting the variables. The function return the resulting graph and keeps a reference in the attribute `res`.

```
bool matching(); Graph * apply(Graph * graph);
```

Reconfiguration controller

The class `ProgramElement` is used for the structure of an element of the reconfiguration program which is in the form of (condition : rule) and is used by the controller. An element of the program is in the form of `actor.metric operand threshold : rule`. A non-functional `metric` is measured and requested from an `actor`. Then, its value is compared using an `operand` with a `threshold`. If the condition holds, the `rule` is applied.

```
class ProgramElement {
  string actor; string metric;
  char operand; int threshold; string rule;
};
```

The `Controller` class has a reference to the initial `graph` and a reference to the currently executing graph, that is, `curr_graph`. It has the list of all transformation `rules` accessed by their names and the reconfiguration program which is the list of all program elements `progs`.

```
class Controller {
  Graph * graph, curr_graph;
  map<string, Rule *> rules; vector<ProgramElement> progs;
};
```

Executing an application. The function `getApplicableRule` returns an applicable rule, that is, a rule whose condition holds. It uses the functions `getTime`, `getThroughput`, `getLatency`, and `getOccupancy` provided by actors. The function `run` runs the dataflow graph according to the following steps: (1) while the end-of-stream is not reached (`!graph.check_eos()`), (2) get the applicable rule `r = getApplicableRule()`, (3) if there is no applicable rule, wait and then continue the while loop, (4) if an applicable rule exists, then pause the graph (`graph.pause()`), (5) apply the rule to the current graph (`r.apply(cur_graph)`), and finally (6) resume the graph (`graph.resume()`) and continue the while loop.

```
  Rule * getApplicableRule(); void run();
```

Bibliography

[1] Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[2] Vagelis Bebelis, Pascal Fradet, and Alain 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, June 2014. ACM.

[3] Vagelis Bebelis, Pascal Fradet, Alain Girault, and Bruno Lavigueur. BPDF: A statically analyzable dataflow model with integer and boolean parameters. In *International Conference on Embedded Software, EMSOFT'13*, pages 1–10, 2013.

[4] Evangelos Bempelis. *Boolean Parametric Data Flow Modeling - Analyses - Implementation*. PhD thesis, Université Grenoble Alpes, 2015.

[5] Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Trans. on Signal Processing (TSP)*, 49(10):2408–2421, 2001.

[6] Shuvra S. Bhattacharyya, Ed F. Deprettere, and Bart D. Theelen. Dynamic dataflow graphs. *Handbook of Signal Processing Systems*, pages 905–944, 2013.

[7] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. *Software synthesis from dataflow graphs*, volume 360. Springer Science & Business Media, 2012.

[8] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cycle-static dataflow. *IEEE Transactions on signal processing*, 44(2):397–408, 1996.

[9] Adnan Bouakaz, Pascal Fradet, and Alain Girault. Symbolic buffer sizing for throughput-optimal scheduling of dataflow graphs. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–10. IEEE, 2016.

[10] Adnan Bouakaz, Pascal Fradet, and Alain Girault. A survey of parametric dataflow models of computation. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 22(2), March 2017.

[11] Adnan Bouakaz, Pascal Fradet, and Alain Girault. Symbolic analyses of dataflow graphs. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 22(2):39, 2017.

[12] Jani Boutellier and Ilkka Hautala. Executing dynamic data rate actor networks on opencl platforms. *arXiv preprint arXiv:1611.03226*, 2016.

[13] Jani Boutellier, Jiahao Wu, Heikki Huttunen, and Shuvra S. Bhattacharyya. Prune: Dynamic and decidable dataflow for signal processing on heterogeneous platforms. *arXiv preprint arXiv:1802.06625*, 2018.

[14] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[15] Joseph T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Proceedings of 1994 28th Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 508–513. IEEE, 1994.

[16] Joseph T. Buck and Edward A. 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*, volume I, pages 429–432, Minneapolis (MN), USA, April 1993. IEEE.

[17] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[18] Aonzo Church. The calculi of lambda-conversion. *Princeton University Press*, 1941.

[19] Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra S Bhattacharyya, and Slaheddine 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*, pages 41–48, Samos Island, Greece, July 2013. IEEE.

[20] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[21] Pascal Fradet, Alain Girault, Leila Jamshidian, Xavier Nicollin, and Arash Shafiei. Lossy channels in a dataflow model of computation. In *Principles of Modeling*, pages 254–266. Springer, 2018.

[22] Pascal Fradet, Alain Girault, and Peter Poplavko. SPDF: A Schedulable Parametric Data-Flow MoC (Extended Version). Research Report RR-7828, INRIA, December 2011.

[23] Pascal Fradet, Alain Girault, and Peter Poplavko. Spdf: A schedulable parametric data-flow moc. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 769–774. IEEE, 2012.

[24] Michael R. Garey and David S. Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[25] Marc Geilen. Synchronous dataflow scenarios. *ACM Trans. on Embedded Computing Systems (TECS)*, 10(2):16, 2010.

[26] Marc Geilen and Sander Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 125–134. IEEE, 2010.

[27] Amir Hossein Ghamarian, Marc Geilen, Sander Stuijk, Twan Basten, Bart Theelen, Mohammad Reza Mousavi, Arno Moonen, and Marco Bekooij. Throughput analysis of synchronous data flow graphs. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 25–36. IEEE, 2006.

[28] Amir Hossein Ghamarian, Sander Stuijk, Twan Basten, Marc Geilen, and Bart Theelen. Latency minimization for synchronous data flow graphs. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 189–196. IEEE, 2007.

[29] Soonhoi Ha and Hyunok Oh. Decidable dataflow models for signal processing: Synchronous dataflow and its extensions. In *Handbook of Signal Processing Systems*, pages 1083–1109. Springer, 2013.

[30] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. Dynamic neural networks: A survey. *arXiv preprint arXiv:2102.04906*, 2021.

[31] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison Wesley, 2006.

[32] Chia-Jui Hsu, Fuat Keceli, Ming-Yung Ko, Shahrooz Shahparnia, and Shuvra S. Bhattacharyya. Dif: An interchange format for dataflow-based design tools. In *International Workshop on Embedded Computer Systems*, pages 423–432. Springer, 2004.

[33] Gilles Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475, 1974.

[34] Hojin Kee, Chung-Ching Shen, Shuvra S. Bhattacharyya, Ian Wong, Yong Rao, and Jacob Kornerup. Mapping parameterized cyclo-static dataflow graphs onto configurable hardware. *Journal of Signal Processing Systems*, 66(3):285–301, 2012.

[35] Edward A. Lee et al. Computing for embedded systems. In *Conference on Instrumentation and Measurement Technology*, volume 3, pages 1830–1837. IEEE, 2001.

[36] Edward A. Lee et al. Overview of the Ptolemy project. Technical Report UCB/ERL M03/25, University of California at Berkeley, Berkeley (CA), USA, July 2003.

[37] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35, 1987.

[38] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[39] Shuoxin Lin, Lai-Huei Wang, Aida Vosoughi, Joseph R. Cavallaro, Markku Juntti, Jani Boutellier, Olli Silvén, Mikko Valkama, and Shuvra S. Bhattacharyya. Parameterized sets of dataflow modes and their application to implementation of cognitive radio systems. *Journal of Signal Processing Systems*, 80(1):3–18, 2015.

[40] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[41] Orlando Moreira, Twan Basten, Marc Geilen, and Sander Stuijk. Buffer sizing for rate-optimal single-rate data-flow scheduling revisited. *IEEE Trans. on Computer*, 59(2):188–201, 2010.

[42] Carl Adam Petri. Communication with automata. Technical report, Rome Air Development Center, 1966.

[43] Carl Adam Petri and Wolfgang Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008.

[44] William Plishker, Nimish Sane, Mary Kiemb, Kapil Anand, and Shuvra S. Bhattacharyya. Functional dif for rapid prototyping. In *2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 17–23. IEEE, 2008.

[45] Claudius Ptolemaeus. *System Design, Modeling, and Simulation: Using Ptolemy II*, volume 1. ptolemy.org, Berkeley, 2014.

[46] Jean-Claude Raoult and Frédéric Voisin. Set-theoretic graph rewriting. In *Graph Transformations in Computer Science*, pages 312–325. Springer, 1994.

[47] Raymond Reiter. Scheduling parallel computations. *Journal of the ACM (JACM)*, 15(4):590–599, 1968.

[48] Mladen Skelin, Marc Geilen, Francky Catthoor, and Sverre Hendseth. Worst-case latency analysis of sdf-based parametrized dataflow mocs. In *2015 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 1–6. IEEE, 2015.

[49] Mladen Skelin, Marc Geilen, Francky Catthoor, and Sverre Hendseth. Parameterized dataflow scenarios. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(4):669–682, 2016.

[50] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. CRC press, 2018.

[51] Sander Stuijk, Marc Geilen, and Twan Basten. Sdf^3: Sdf for free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 276–278. IEEE, 2006.

[52] Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 404–411. IEEE, 2011.

[53] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196. Springer, 2002.

[54] Maarten Wiggers, Marco Bekooij, and Gerard Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 183–194. IEEE, 2008.

[55] Maarten Wiggers, Marco Bekooij, and Gerard Smit. Buffer capacity computation for throughput-constrained modal task graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(2):1–59, 2011.