Lossy channels in a dataflow model of computation

Pascal Fradet, Alain Girault, Leila Jamshidian, Xavier Nicollin, Arash Shafiei

Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP[*], LIG, 38000 Grenoble, France

October 30, 2017

Abstract

In this paper, we take into account lossy channels and retransmission protocols in dataflow models of computation (MoCs). Traditional dataflow MoCs cannot easily cope with lossy channels, due to the strict notion of iteration that does not allow the re-emission of lost or damaged tokens. A general dataflow graph with several lossy channels will indeed require several phases, each of them corresponding to a portion of the initial graph's schedule. Correctly identifying and sequencing these phases is a challenge. We present a translation of a dataflow graph, written in the well-known Synchronous DataFlow (SDF) MoC of Lee and Messerschmitt, but where some channels may be lossy, into the Boolean Parametric DataFlow (BPDF) MoC.

1 Introduction

The Internet of Things (IoT) has led to the deployment of billions of small devices that are interconnected mainly by wireless communication protocols. A lot of IoT applications use a form of dataflow communication between the nodes, so it seems a good idea to use a dataflow Model of Computation (MoC) to program such applications. One great advantage is the possibility to perform formal reasoning at compile time, ensuring bounded memory, absence of deadlock, schedulability, and performance properties. The problem is that IoT applications are subject to communication losses, which can arise for various reasons: e.g., electromagnetic interferences, low bandwidth, power shortage (frequent in tiny devices which are typical of the IoT). There are many communication protocols, such as Automatic Repeat Request (ARQ) protocols, to deal with lossy channels and to achieve reliable transmission. These techniques are all based on *retransmissions*.

[*]Institute of Engineering Univ. Grenoble Alpes

Traditional dataflow MoCs cannot easily cope with lossy channels, due to the strict notion of iteration that does not allow the retransmission of lost or damaged tokens. Consider a simple dataflow graph of the form

$$X \rightarrow Y \rightsquigarrow Z \rightarrow T$$

where $\rightsquigarrow$ denotes a lossy channel. Executing such a graph consists in executing X, Y, Z, and T consecutively. But if Z reads corrupted data it has to produce immediately data which most probably depends on its input. Furthermore, if Z then asks for a retransmission, then executing Y again would entail reading new data from X. The partial re-executions asked by ARQ protocols do not fit within the standard dataflow model.

In this paper, we propose to use the Boolean Parametric DataFlow (BPDF) MoC [2] to deal with lossy channels and the necessary retransmissions when tokens are damaged or lost. BPDF extends the classical Synchronous DataFlow (SDF) MoC of Lee and Messerschmitt [5] with *Boolean parameters* on the dataflow edges, which permits to *disable* and *enable* edges. By carefully controlling the Boolean conditions, we can model the execution phases of dataflow graphs with lossy channels.

Section 2 presents the necessary background, namely the SDF and BPDF MoCs. SDF with lossy channels and its translation into BPDF are described in Section 3. Section 4 suggests several future work directions. Finally, Section 5 summarizes our contributions and concludes.

2 Background

Since our goal is to extend SDF with a notion of lossy channel and to show how to translate this model into BPDF, we present these two MoCs in turn.

2.1 Synchronous DataFlow (SDF)

Formally, an SDF graph $G = \langle \mathcal{V}, \mathcal{E}, \iota, \rho \rangle$ consists of:

- a finite set of actors (computation nodes) $\mathcal{V}$;
- a finite set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$; edges can be seen as unbounded FIFO channels; if $e = (X, Y)$, also written XY, is an edge, then e is an *outgoing edge* of X, and an *incoming edge* of Y.
- a function $\iota : \mathcal{E} \rightarrow \mathbb{N}$ that returns, for each edge, its number of initial tokens (possibly zero);
- a function $\rho : \mathcal{E} \rightarrow \mathbb{N}_{>0} \times \mathbb{N}_{>0}$ that returns, for each edge a tuple containing the *production rate* of its source actor and the *consumption rate* of its sink actor.

The execution of an actor (called *firing*) first consumes data tokens from all its incoming edges (its inputs), then computes, and finishes by producing data tokens to all its outgoing edges (its outputs). The number of tokens consumed (resp. produced) at a given incoming (resp. outgoing) edge at each firing is called its consumption (resp. production) rate and is specified by function ρ. An actor can fire only when all its incoming edges have enough tokens, *i.e.,* at least the number specified by the corresponding rate (edges may have a non-null number of initial tokens, defined by function ι). For instance, Fig. 1 shows a simple SDF graph G with three actors A, B, C.

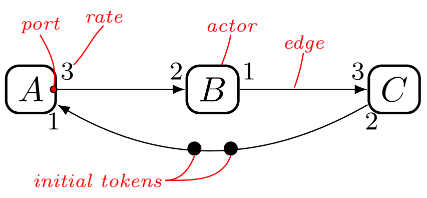


Figure 1: A simple SDF graph with 3 actors and 3 edges.

Each edge carries zero or more tokens at any moment. The state of a dataflow graph is the vector of the number of tokens present at each edge. The initial state of a graph is defined as the vector of the number of initial tokens on its edges. For instance, the initial state of the graph of Fig. 1 is the vector $[0; 0; 2]$.

Because all rates in SDF are fixed values, a static schedule can be produced and a number of analyses can be performed at compile time (*e.g.,* boundedness, liveness, throughput, latency, ...).

An *iteration* of an SDF graph is a non empty sequence of firings that returns the graph to its initial state[1]. For the SDF graph in Fig.1, firing actor A twice (consuming 2 tokens and producing 6 tokens), actor B thrice (consuming 6 tokens and producing 3 tokens), and finally actor C once (consuming 3 tokens and producing 2 tokens) forms an iteration. We write $\#X$ the number of firings of actor X in the iteration.

The *basic repetition vector* $\vec{Z} = [\#A{=}2, \#B{=}3, \#C{=}1]$ indicates the number of firings of actors per (minimal) iteration, and the iteration is noted (A^2, B^3, C). The repetition vector is obtained by solving the system of *balance equations*: each edge $X \xrightarrow{p \quad q} Y$ is associated with the balance equation $\#X.p = \#Y.q$, which states that all produced tokens during an iteration must be consumed within the same iteration. If non-null solutions exist, the graph is said to be consistent [5], and the smallest solution defines the basic repetition vector. Consistency ensures that the graph can be executed infinitely in bounded memory.

[1]We only consider here the *minimal* iteration. Any multiple of the minimal iteration is also a valid iteration.

The goal of the deadlock analysis is to check that the graph G admits a schedule that is always live, called an *admissible schedule.* A simple algorithm to find such a schedule performs a symbolic execution of the SDF graph [5]. Among the admissible schedules, we distinguish *flat single appearance schedules* [4] (FSAS) where, once factorized (*i.e.,* any sequence $A; \ldots ; A$ of n firings of A is replaced by A^n), each actor appears exactly once. The SDF graph G of Fig. 1 admits only one FSAS: $\{A^2; B^3; C\}$. An acyclic SDF graph always admits a FSAS, while a cyclic SDF graph admits a FSAS if and only if each cycle includes at least one *saturated* edge, that is, an edge XY that contains enough initial tokens to fire Y at least $\#Y$ times.

2.2 Boolean Parametric DataFlow (BPDF)

The Boolean Parametric DataFlow (BPDF) MoC [2] extends SDF with two features: *integer parameters* for rates, similar to [3], and *Boolean parameters* annotating edges. Only the second feature is of interest for the present paper, so we focus on it.

The general idea is that any edge of a BPDF graph can be labeled with a Boolean expression built from the following grammar:

$$\mathcal{B} ::= \mathit{tt} \mid \mathit{ff} \mid b \mid \neg\mathcal{B} \mid \mathcal{B}_1 \wedge \mathcal{B}_2 \mid \mathcal{B}_1 \vee \mathcal{B}_2 \tag{1}$$

where tt is true, ff is false and b denotes *Boolean parameters*. Each Boolean parameter b is modified by a single actor called its *modifier.* Each modifer has annotations of the form "$b@\pi_w$" where b is the Boolean parameter to be set and π_w is the *period* of the Boolean parameter, that is, the exact number of firings of its modifier between two successive assignments[2].

Formally, a BPDF graph is a tuple $G = \langle \mathcal{V}, \mathcal{E}, \iota, \rho, P_b, \beta, M, \pi_w \rangle$ (for the sake of simplicity, integer parameters are omitted here) where:

- $\mathcal{V}$ (actors), $\mathcal{E}$ (edges), ι (initial tokens), and ρ (rates) are defined as in SDF graphs (see Section 2.1);
- P_b is the set of Boolean parameters;
- $\beta : \mathcal{E} \to \mathcal{B}$ returns, for each edge, its Boolean expression;
- $M : P_b \to \mathcal{V}$ returns, for each Boolean parameter, its modifier;
- $\pi_w : P_b \to \mathbb{N}_{>0}$ returns, for each Boolean parameter, its writing period.

In general, a Boolean parameter can take several values during an iteration of a BPDF graph. However, in the context of this paper, Boolean parameters take only one value per iteration. In other words, $\forall b \in P_b, \pi_w(b) = \#M(b)$. Figure 2 shows a BPDF graph with three actors and two Boolean parameters.

[2]Obviously, an assignment does not necessarily change the value.

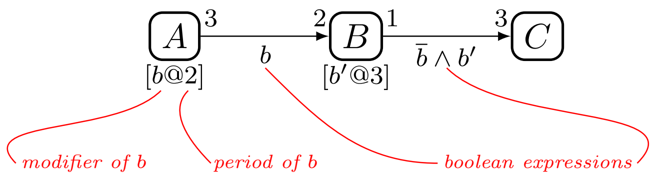


Figure 2: A simple BPDF graph.

An edge labeled by a Boolean expression is *disabled* whenever its expression evaluates to false, and *enabled* otherwise. An edge not labeled by a Boolean expression can be seen as labeled by tt, and thus behaves exactly as in SDF. When an edge $X \to Y$ is disabled, X fires but does not emit any token to Y (but emits tokens on its enabled outgoing edges) and Y fires but does not read any token from X (but must read tokens from all its enabled incoming edges). When an actor X is such that all its edges are disabled, it still fires; such firings are referred as *dummy*. However, a modifier of one or more Boolean parameters may still update their value during a dummy firing.

A *user* of a Boolean parameter b is an actor with one of its edge labeled by a Boolean expression that depends on b. Formally, the set of users of b is defined as:

$$\mathit{Users}(b) = \{X \in \mathcal{V} \mid \exists Y \in \mathcal{V} : b \in \beta(XY) \vee b \in \beta(YX)\}$$

Once a new value for b is produced, it is propagated to all users of b.

Whenever it fires, a BPDF actor X performs the following steps:

1. Read the value of each Boolean parameter b for which $X \in \mathit{Users}(b)$ (only at its first firing in the iteration);
2. Consume tokens on the enabled incoming edges, which must have enough tokens (otherwise the actor is blocked);
3. Compute its new internal state and outputs;
4. Produce tokens on the enabled outgoing edges;
5. If X is the modifier of a Boolean parameter b and the current firing corresponds to its period ($\pi_w(b)$), then the value of b is propagated to all its users ($\mathit{Users}(b)$). In this paper, $\pi_w(b)$ is restricted to be equal to $\#M(b)$ so such propagations take place only during the last firing of each modifier in the iteration.

In constrast, an SDF actor only performs steps 2, 3, and 4, and of course all its edges are always enabled.

Consistency analysis in BPDF requires to check, as in SDF, rate consistency. We ignore the Boolean expressions and solve the system of balance equations to check that there exists a non-null solution. In general, a second condition called period safety, should be checked (see [2]). However, in this paper, since Booleans parameters are changed at most once by iteration the second condition is trivially true. Liveness has also to be checked using a refinement of the algorithm used in SDF (see [2]). It is easy to check that the BPDF graph of Fig. 2 is consistent, live, and that its iteration is (A^2, B^3, C).

In general, integer parameters prevent the generation of static schedules for BPDF graphs [1]. In the context of this paper, we do not consider integer parameters and we are able to generate static schedules. For instance, the only FSAS of the BPDF graph of Fig. 2 is $\{A^2; B^3; C\}$. Note that A and B are the modifiers of b and b' respectively, whereas A is a user of b, and B and C are users of b and b'. Therefore, the first firing of A reads the value of b produced in the *previous* iteration, whereas the second (and last) firing produces the value of b that will be used in the *next* iteration. Similarly, the first firing of B reads the values of b and b' produced in the *previous* iteration, while the third (and last) firing of B produces the value of b' that will be read in the *next* iteration. Finally, the first (and only) firing of C reads the values of b and b' produced in the *previous* iteration.

3 From lossy SDF to BPDF

We call *lossy SDF* the SDF model enriched with the information on whether edges are lossy or not. Informally a lossy SDF graph should behave exactly as if all its channels were non lossy. Its high-level semantics is therefore given by the SDF semantics. One assumption needs to be formulated though: on each lossy channel, tokens are eventually transmitted correctly. If this cannot be guaranteed, a maximum number of retransmissions can be specified for each lossy channel, with a default token value (and, in that case, the semantics departs from SDF's).

We show in this section how a lossy SDF graph can be translated automatically into a BPDF graph with an equivalent semantics.

The intuitive semantics of lossy SDF can be implemented based on selective retransmissions. Consider the same simple dataflow graph as in the introduction

$$X \to Y \rightsquigarrow Z \to T$$

where $\rightsquigarrow$ denotes a lossy channel. We saw that the standard dataflow execution does not suit potential re-executions. Our solution is to divide the execution of this graph in three phases: first the X-Y part where X fires and Y only reads (called the *upstream* phase), then the Y-Z part where Y only writes and Z only reads (called the *lossy* phase), and finally the Z-T part where Z only writes and T fires (called the *downstream* phase). This division allows re-executions of Y-Z until the token sent by Y is correctly received by Z. Of course when there are multiple lossy channels or cycles in the graph, many phases should be considered and combined. We present how to implement such phases in BPDF using Boolean conditions to enable/disable individual edges.

3.1 Translation of a simple SDF graph with one lossy channel

Consider the SDF graph of Fig. 3, where the edge BC is lossy, indicated by a curly arrow. Its iteration is (A^2, B^3, C^3, D^3) and its only FSAS is $\{A^2; B^3; C^3; D^3\}$.

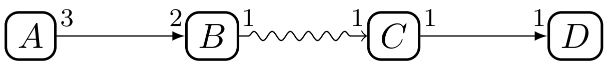


Figure 3: A simple SDF graph with one lossy channel BC.

To account for the lossy channel BC, this graph is executed into three consecutive phases:

1. **Upstream phase:** First $\{A^2; B^3\}$ where B reads the tokens produced by A but does not send any token to C;
2. **Lossy phase:** Then $\{B^3; C^3\}$ which may be repeated until all tokens sent by B are correctly received by C; in this phase, B does not read any token on channel AB;
3. **Downstream phase:** Finally $\{C^3; D^3\}$ where C does not read tokens from the edge BC and sends the tokens to D.

The BPDF graph implementing these three phases is shown in Fig 4. Its FSAS is $\{A^2; B^3; C^3; D^3\}$. The first phase is when $b = tt \wedge b' = tt$. The Boolean expressions of both edges BC and CD evaluate to ff. The actor B fires three times and reads its incoming tokens from A but does not send any. Since both C and D are disconnected their three firings are dummy. The second phase corresponds to $b = ff \wedge b' = tt$. Now the firings of A and D are dummy, B does not read any tokens from A, and C does not write any token to D. The only exchange of tokens takes place between B and C. This phase can be repeated as long as $b = ff \wedge b' = tt$. The third phase is when $b = ff \wedge b' = ff$, yielding the iteration $\{A^2; B^3; C^3; D^3\}$ where the firings of A and B are dummy firings, and the only exchange of tokens takes place between C and D. This three phase cycle can now be repeated by returning to $b = tt \wedge b' = tt$.

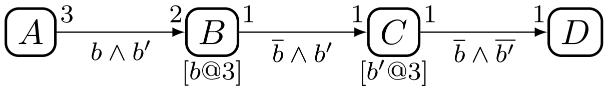


Figure 4: The translation into BPDF of the graph of Fig. 3.

The Boolean parameter b could be set by any actor in the graph. Here we have chosen B to set b, thereby controlling the end of the first phase which always occurs after one iteration. In contrast, the Boolean parameter b' must mandatorily be set by actor C, because C is the only actor capable of asserting when the tokens produced by B have been received correctly. We assume that the communication system layer provides information about token corruption and/or loss. This can be performed by using error-detecting codes and/or time out mechanisms. For instance, one of the Automatic Repeat-Request (ARQ) protocols, *e.g.,* Stop-and-Wait ARQ, Go-Back-N ARQ, or Selective Repeat ARQ, can be used [9]. In general, the SDF graph will include several lossy channels, yielding more than three phases and requiring more Boolean parameters, as we see in the next section.

3.2 General translation algorithm

In this section, we propose a general translation from a lossy SDF graph into an equivalent BPDF graph. By "equivalent", we mean that the semantics of the resulting BPDF graph must coincide with the semantics of the original lossy SDF graph.

Let $G = \langle \mathcal{V}, \mathcal{E}, \iota, \rho \rangle$ be the initial SDF graph and let $\mathcal{L} \subseteq \mathcal{E}$ be the subset of lossy channels. We assume that G admits a sequential FSAS denoted by S_G. The translation from G into a semantically equivalent BPDF graph proceeds as follows:

1. We number the actors from 1 to n ($n = |V|$) according to their order of appearance in S_G. They are now uniquely identified as V_1, V_2, $\dots V_n$.

2. S_G also induces a total order on the edges of $\mathcal{E}$. An edge AB occurs before another XY, if A occurs before X in the FSAS S_G and AB occurs before AC if B occurs before C in S_G. Formally,

   $$\forall (V_iV_j), (V_kV_\ell) \in \mathcal{E}, (V_iV_j) < (V_kV_\ell) \Leftrightarrow (i < k) \vee (i = k \wedge j < \ell)$$

   We number all edges from 1 to p ($p = |\mathcal{E}|$) which are now uniquely identified as E_1, E_2, $\dots$, E_p.

3. The total order on $\mathcal{V}$ can be projected onto $\mathcal{L}$, yielding a total order on $\mathcal{L}$, so we can number lossy channels from 1 to $|\mathcal{L}| = q$. All lossy channels in $\mathcal{L}$ are now uniquely identified as L_1, L_2, $\dots L_q$. Moreover, for each $j \in [1, q]$, there exists a unique $i \in [1, p]$ such that $L_j = E_i$. We denote this index $i = \varphi(j)$.

4. Then, G is translated into the BPDF graph $G' = \langle \mathcal{V}, \mathcal{E}, \iota, \rho, P_b, \beta, M, \pi_w \rangle$ such that:

   - Actors $\mathcal{V}$, edges $\mathcal{E}$, production/consumption rates ρ, and number of initial tokens ι remain the same as in G.

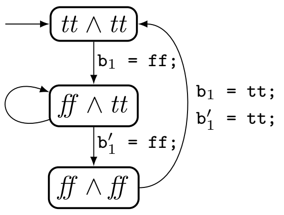
- For each lossy channel L_i, we introduce two Boolean parameters b_i and b'_i. The resulting set of Boolean parameters is defined as: $P_b = \{b_i, b'_i | 1 \leq i \leq q\}$.
- For all $1 \leq i \leq p$, we set $\beta(E_i) = bc_1 \wedge bc_2 \wedge bc_3$ with:
  - bc_1 accounts for all lossy channels that are *after* E_i in S_G: $bc_1 = \bigwedge_{j=u}^{q} (b_j \wedge b'_j)$ with $u = \min\{j \in [1,q] \,|\, \varphi(j) > i\}$.
  - bc_2 accounts for the fact that E_i may be itself a lossy channel: if $\exists j \in [1,q]$ such that $E_i = L_j$, then $bc_2 = \overline{b_j} \wedge b'_j$ else $bc_2 = tt$.
  - bc_3 accounts for all lossy channels that are *before* E_i in S_G: $bc_3 = \bigwedge_{j=1}^{\ell} (\overline{b_j} \wedge \overline{b'_j})$ with $\ell = \max\{j \in [1,q] \,|\, \varphi(j) < i\}$.
- For all $1 \leq i \leq q$ and $L_i = S_i R_i$, we set $M(b_i) = S_i$ and $M(b'_i) = R_i$ with $\pi_w(b_i) = \#M(b_i)$ and $\pi_w(b'_i) = \#M(b'_i)$.

The BPDF actors connected by a lossy channel $S \rightsquigarrow R$ must also be instrumented. The receiver R needs to detect when the received tokens are correct so that it can change the phase by propagating a new Boolean value. As already mentioned, we assume that the communication system marks tokens as correct or incorrect. The sender S needs to keep a copy of its transferred tokens in order to resend them when necessary. It knows not to resend tokens when the phase changes.

3.3 Sequencing the phases

The BPDF graph of Fig. 4 runs according to three phases. These three phases are summarized in the following table on the left (dummy firings are omitted):

Phase	Partial schedule	b_1	b'_1
1 (upstream)	$\{A^2; B^3\}$	*tt*	*tt*
2 (lossy)	$\{B^3; C^3\}^*$	*ff*	*tt*
3 (downstream)	$\{C^3; D^3\}$	*ff*	*ff*



On the right we have shown the labeled transition system (LTS) of the three phases. We adopt the convention that all the Boolean parameters are initially equal to *tt*, so its initial state is phase 1, corresponding to $tt \wedge tt$. To implement these three phases, the modifiers of the two Boolean parameters must implement the following pseudo-code:

Actor B	Actor C
`if (phase==1) then b₁=ff;`	`if (phase==2) then b'₁=ff;`
`if (phase==3) then b₁=tt;`	`if (phase==3) then b'₁=tt;`

Note that an actor can easily determine the current phase by looking at the current values of the Boolean parameters.

We now address the general case. A BPDF graph with q lossy channels exhibits at most $2q+1$ phases, due to the fact that we totally order the edges according to the FSAS. We have chosen to implement these $2q+1$ phases with $2q$ Boolean parameters (the b_is and b'_is), although one may think that $\lceil log_2(2q+1)\rceil$ Booleans would be enough. Yet, there is a restriction that, for each lossy channel $L_i = S_iR_i$, only R_i can control the end of the lossy phase of L_i, because only R_i can tell whether or not the tokens sent by S_i have been received correctly. It follows that at least q Boolean parameters are required for this, one for each lossy channel (the b'_is). Yet, the remaining q Booleans could be optimized (the b_is). This could be the topic of future work.

The sequencing of the phases for a general graph can be represented by a similar LTS as the one shown above for the graph of Fig. 4. To implement such an LTS, we must provide the pseudo-code for each actor that modifies one (or more) Boolean parameter(s). For each lossy channel $L_i = S_iR_i$, recall that we have two Boolean parameters, b_i and b'_i, respectively modified by S_i and R_i, such that S_i controls the switching from phase $2i-1$ to $2i$ while R_i controls the switching from phase $2i$ to $2i+1$. After the last phase, each modifier must also reset its Boolean parameters to *tt* to return to the initial phase. As a consequence, S_i and R_i must implement the following pseudo-code:

S_i	R_i

`if (phase==2*i-1) then` b_i`=ff;`
`if (phase==last)  then` b_i`=tt;`
	`if (phase==2*i)  then` b'_i`=ff;`
	`if (phase==last) then` b'_i`=tt;`

As we have said, the maximum number of phases with q lossy channels is $2q+1$. Yet, there are several cases when this number can be reduced. For instance when there are two lossy channels in sequence, say XY and YZ, then Y will update both the Boolean b'_i corresponding to XY and the Boolean b_{i+1} corresponding to YZ. As a consequence, there is one phase less because two Booleans are set to *ff* at the same firing of Y. Another typical case is when there is a fork of two lossy channels, say XY and XZ. As in the previous case, X sets two Booleans to *ff* during its firing, so there is one less phase. It follows that the precise number of phases must be computed prior to obtain the value of `last` used in the above table.

Finally, a particular case occurs when the first lossy channel L_1 is also the first edge E_1. In this case, there is no real first phase, because when all the Boolean parameters are equal to *tt* and all the edges of the BPDF graph are disabled, so each actor performs a dummy firing. A similar case occurs when the last lossy channel L_q is also the last edge E_p. In this case there is no real last phase since all the actors perform only a dummy firing. Of course, these special situations could be optimized out.

3.4 Cyclic graphs

Cyclic lossy SDF graphs pose a problem because the backward edges appear both in the upstream phases so that the destination actor can consume the initial tokens, *and* in the downstream phase so that the source actors can produce the initial tokens for the next iteration of the graph.

Recall that we have assumed that each cycle contains at least one saturated edge (see Section 2.1).

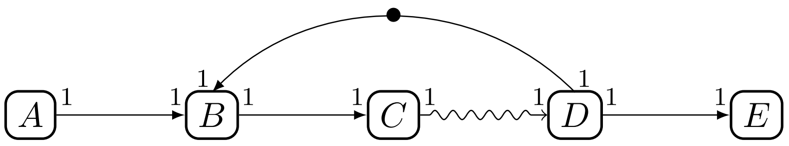


Figure 5: An SDF graph with a cycle and a lossy channel.

Consider the example of Fig. 5. For the sake of simplicity, all the production and consumption rates are equal to 1. Assuming the FSAS $\{A; B; C; D; E\}$, its translation into BPDF is shown in Fig. 6.

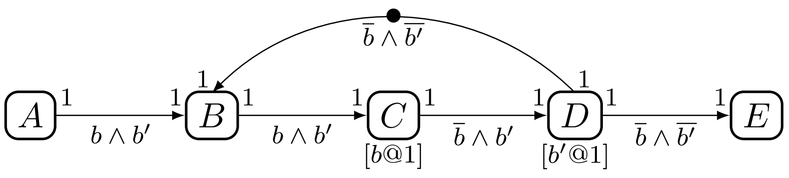


Figure 6: The BPDF graph obtained by translating the SDF graph of Fig. 5.

As we can see, the backward edge DB (which is saturated thanks to the initial token) belongs only to the *downstream* phase, hence it is not executed during the upstream phase. It follows that, when B fires during the upstream phase, it cannot read any initial token from the backward edge DB because this edge is disabled. To solve this issue, we consider that all initial tokens are in fact stored directly in the *internal memory* of the destination actor of the edge to which they belong. In the case of Fig. 6, this means that the initial token of the backward edge DB is stored in the internal memory of actor B. As a consequence, during the upstream phase, B reads the token sent by A and the initial token stored in its internal memory, and sends a token to C. During the lossy phase D reads the token sent by C until this token is correctly received. Finally, during the downstream phase D sends a token to E and a token to B on the backward edge, this last token being in fact directly written in the internal memory of B.

4 Future work

This work is still in progress and we present in this section a number of issues that remain open.

Influence of the chosen FSAS

The translation algorithm is based on an arbitrary FSAS of the considered graph G. An interesting question is what is the influence of this choice whenever G admits several FSASs.

Let us first remark that the choice of the FSAS can change the number of phases. Consider for instance the SDF graph G of Fig. 7(a) with one lossy channel, with its translation into BPDF G' shown in Fig. 7(b).

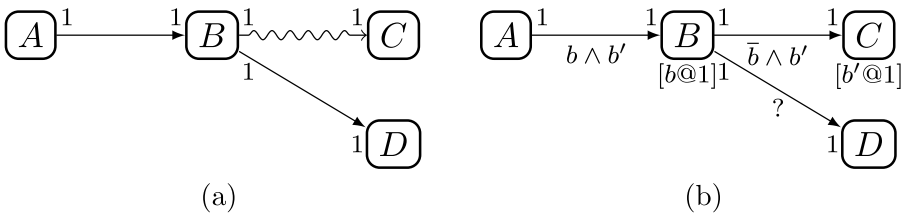


Figure 7: (a) A graph G that admits two FSASs. (b) Its translation G'.

The Boolean expression attached to BD is marked with a '?' because it depends on the FSAS of G. Indeed, G admits two FSASs: $\{A; B; C; D\}$ and $\{A; B; D; C\}$. With the first FSAS, BD belongs to the downstream phase, thereby getting the Boolean expression $\overline{b} \wedge \overline{b'}$. With the second FSAS, BD belongs to the upstream phase, thereby getting $b \wedge b'$. The second FSAS results in two phases against three phases for the first FSAS. A follow up question is the impact of the reduction of the number of phases. More generally, we should strive to choose a FSAS that optimizes performance criteria.

Parallel schedules

Another topic of future work is how to consider *parallel schedules*. So far, we start from a sequential FSAS to sequence the phases. This is adequate when the SDF graph is sequential, but not when it is parallel. An idea to preserve the parallelism in phases would be to identify, for each lossy channel L_j, its *upstream cone* $UC(L_j)$ and its *downstream cone* $DC(L_j)$. Intuitively, the upstream cone of XY is the set of all predecessor edges of X. This is a classical graph traversal problem. Each edge E_i in the resulting BPDF graph will therefore get the Boolean expression $\beta(E_i) = bc_1 \wedge bc_2 \wedge bc_3$ where:

- bc_1 accounts for all the lossy channels L_j such that E_i belongs to $UC(L_j)$;

- bc_2 accounts for the fact that E_i may be itself a lossy channel;
- bc_3 accounts for all the lossy channels L_j such that E_i belongs to $DC(L_j)$.

This would result in a BPDF graph with more parallelism than with the algorithm proposed in Section 3.2, because the phases would not be totally ordered as in Section 3.3. This would allow the generation of parallel code (for which there is a huge literature, *e.g.,* [6] to cite just one).

A related topic is to handle joins of lossy channels in parallel instead of sequentially. Consider, for instance, a set of lossy channels $\{X_1Y, \ldots X_nY\}$. The phases of the lossy channels X_iY can actually be run in parallel: Y would handle all the Boolean parameters b'_i, setting them from tt to $f\!f$ as soon as the tokens from the corresponding actors X_i are received correctly, and moving to the next phase only when the predicate $\bigwedge_{i=i}^{n} \overline{b'_i}$ becomes tt.

Optimization and performance evaluation

When the initial graph contains q lossy channels, we use $2 * q$ Boolean parameters to make up the phases in the BPDF graph. As explained in Section 3.3, this could be optimized.

The general topic of the performance evaluation of the BPDF graphs raises many issues because the number of token retransmissions necessary for each lossy channel cannot be known in advance. Therefore, the *exact* worst case response time cannot be computed. Instead, we may compute the *expected* worst case response time, based on the probability distribution of the damaged/lost tokens on each lossy channel.

5 Conclusion

Modeling lossy channels in a dataflow MoC is relevant for the future IoT applications where mobile devices communicate through wireless channels that are subject to packet loss or damage. In order to model dataflow applications with unreliable communications, we have presented a translation from an SDF graph with lossy channels into the Boolean Parametric DataFlow (BPDF) MoC. This translation isolates the necessary phases of execution of the graph to cope with the retransmissions caused by lost or damaged tokens transmitted over the lossy communication links, and to sequence correctly those phases.

There are a few dataflow MoCs that could express such phases. For instance, we could adopt the Scenario Aware DataFlow MoC (SADF) [8] or its FSM extension [7]. However, this has the inconvenient that all phases must be made explicit, resulting in a potential state space explosion if many lossy channels must be modeled. For this reason, we have chosen BPDF, which uses Boolean parameters to encode the phases and to keep them implicit.

A final issue is the interaction between actors and the system in charge of detecting the lost and/or damaged tokens sent over the lossy channels. We have assumed a communication system layer that implements the error-detecting code (in charge of detecting damaged tokens), a timeout mechanism (in charge of detecting lost tokens) and some ARQ protocol in charge of propagating the Boolean parameters from their modifier to all their users. These hypotheses should be validated by an actual implementation.

Acknowledgments

The authors are grateful for the numerous discussions with Prof. Edward Lee, the invitations to visit Berkeley, and all the good moments spent at various conferences around the world. In more than one way, the work presented in this paper originated in those discussions.

References

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

[2] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur. BPDF: A statically analyzable dataflow model with integer and boolean parameters. In *International Conference on Embedded Software, EMSOFT'13*. ACM, September 2013.

[3] B. Bhattacharya and S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Trans. Signal Processing*, 49(10):2408–2421, October 2001.

[4] S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Pub., Hingham, MA, 1996.

[5] E.A. Lee and D.G. Messerschmitt. Synchronous data-flow. *Proceedings of the IEEE*, 75:1235–1245, September 1987.

[6] G.C. Sih and E.A. Lee. A compile-time scheduling heuristic for interconnection constraint heterogeneous processor architectures. *IEEE Trans. Parallel and Distributed Systems*, 4(2):175–187, February 1993.

[7] M. Skelin, M. Geilen, F. Catthoor, and S. Hendseth. Parametrized dataflow scenarios. In *International Conference on Embedded Software, EMSOFT'15*, pages 95–104, Amsterdam, Netherlands, October 2015. IEEE.

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

[9] A.S. Tanenbaum and D. Wetherall. *Computer networks, 5th Edition*. Pearson, 2011.