Delays and States in Dataflow Models of Computation

Florian Arrestier, Karol Desnos, Maxime Pelcat, Julien Heulot, Eduardo Juarez, Daniel Menard

Delays and States in Dataflow Models of Computation

Florian ARRESTIER*, Karol DESNOS*, Maxime PELCAT*, Julien HEULOT*, Eduardo JUAREZ†, Daniel MENARD*
*Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164, Rennes, France
email: farresti, kdesnos, mpelcat, dmenard@insa-rennes.fr
†CITSEM, UPM, Madrid, Spain
email: eduardo.juarez@upm.es

***Abstract*—Dataflow Models of Computation (MoCs) have proven efficient means for modeling computational aspects of Cyber-Physical System (CPS). Over the years, diverse MoCs have been proposed that offer trade-offs between expressivity, conciseness, predictability, and reconfigurability. While being efficient for modeling coarse grain data and task parallelism, state-of-the-art dataflow MoCs suffer from a lack of semantics to benefit from the lower grained parallelism offered by hierarchically modeled nested loops.**

State-Aware Dataflow (SAD) extends the semantics of the targeted MoC with unambiguous data persistence scope. The extended expressiveness and conciseness brought by the SAD meta-model are demonstrated with a reinforcement learning use-case.

I. Introduction

Dataflow Models of Computation (MoCs) are commonly used to model stream processing applications in a wide range of domains such as video and audio processing [1], [2], telecommunications [3], and computer vision [4]. Dataflow MoCs and related languages are subject to an increasing popularity due to their advanced analyzability and their natural expressiveness of parallelism. The recent specialized dataflow-based programming language TensorFlow [5] is an evidence of this popularity in the context of neural networks training on massively parallel hardware architectures. In the computer vision applications field, the recent OpenVX [6] standard aims at providing high performances on heterogeneous architectures based on a dataflow MoC. Representing an application with a Dataflow Process Network (DPN) [7] consists of dividing it into persistent processing entities, named actors, connected by First-In First-Out queues (FIFOs). An actor *fires*, meaning that it performs some processing, when enough data tokens are available on its input FIFOs. A data token is an abstract, indivisible data unit. When an actor completes a firing, it may produces data tokens on its output FIFOs. The number of data tokens consumed and produced by an actor for each firing is defined by a set of *firing rules* [7]. Different firing rules are specified in the semantics of existing dataflow MoCs. For instance, in the Synchronous DataFlow (SDF) MoC the number of data tokens exchanged by an actor at each firing is constant. FIFOs can have an initial state corresponding to an initial number of data tokens present in the FIFO, available prior to any actor firing. The initial data tokens of a FIFO are called *delays*.

Dataflow MoCs have proven efficient for modeling continuous streaming applications with coarse grain data and task parallelism. However, iterative structures with finer grained parallelism such as nested loops are not efficiently modeled by current dataflow MoCs. Dataflow MoCs are well suited for static stream oriented applications but in modern streaming applications such as adaptive filtering or machine learning, parameters are updated dynamically and thus need to persist across graph iterations.

In this paper, we show that solutions found in the literature to tackle the problem of persistent data do not offer the necessary flexibility in the application design process. As an answer to these limitations, we introduce the State-Aware Dataflow (SAD) meta-model, applicable to any dataflow MoC with a well defined concept of graph iteration. SAD introduces explicit initialization semantics for delays and unambiguous persistence scope for data tokens. The semantics of delays proposed by SAD aims at offering a concise and expressive representation of complex applications containing nested loops. Persistent data tokens are used to store the state of a graph across successive graph iterations. SAD enforces the possibility of local persistent state at low levels of hierarchy with no impact on the data parallelism of top-level actors. Thus, SAD provides programmers with flexibility in the design of complex applications by enforcing the compositionality of hierarchical graphs.

Previous work is presented in Section II. Then, the new semantics of delays of SAD is formally introduced in Section III. Section IV presents the persistence semantics of SAD and the application of the SAD meta-model to the Parameterized and Interfaced Synchronous DataFlow (πSDF) MoC [3]. Finally, an application of SAD to a reinforcement learning use case is detailed in Section V before concluding in Section VI.

II. Background and Related Work

A. Synchronous DataFlow MoC

The Synchronous DataFlow (SDF) MoC [8] is the most popular static specialization of the DPN MoC. Firing rules of the SDF MoC define data token production and consumption

rates of actors as fixed scalars. The graphical semantics of the SDF MoC and an example of SDF graph are presented in Figure 1.

Figure 1: Semantics of the SDF MoC and example graph.

Formally, an SDF graph $G = (A, F)$ contains a set of actors A that are interconnected through a set of FIFOs F. An actor $a = (P^{in}, P^{out}) \in A$ contains a set of input ports P^{in} and a set of output ports P^{out}. Each port $p \in P$ is used to anchor a unique FIFO between actors. Functions $src : F \rightarrow P^{in}$ and $sink : F \rightarrow P^{out}$ associate source and sink ports to a given FIFO. The data rate specific to each port is defined by the function $rate : (P^{in} \cup P^{out}) \rightarrow \mathbb{N}^*$ corresponding to the fixed firing rule of an SDF actor.

The initial data tokens of a FIFO $f \in F$ are called delays. Formally, a delay $d = (f, n)$ is associated to a FIFO $f \in F$, and a value $n \in \mathbb{N}^*$. The value *n* of the delay is the number of initial data tokens of f. Some properties of dataflow graphs are defined hereafter as they will be used in the rest of the paper.

Consistency is the property of a dataflow graph to be bounded in memory, which means that data tokens will not accumulate indefinitely in any FIFO of the graph. Consistency is checked through the analysis of the topology matrix Γ associated with an SDF graph [8]. Formally, $\Gamma(i, j)$ is the amount of data tokens produced or consumed by actor i on FIFO j. $\Gamma(i, j)$ is a positive number if the actor i produces data tokens on the FIFO j and a negative number if the actor consumes data tokens. The graph is consistent if $rank(\Gamma) = |A| - 1$, with $|A|$ the number of connected actors in the graph [8]. The Repetition Vector (RV), noted q, is defined as the smallest non-zero integer vector verifying $\Gamma * q = 0$.

Liveness is the property of a graph to run indefinitely without any deadlocks. A live graph has sufficient initial data tokens to execute a full iteration and each graph iteration starts with the same number of initial data tokens. Different approaches to the problem of verifying liveness of a graph exist in the literature. The most common approach is called the Symbolic Execution (SE). SE consists of performing a symbolic execution of a graph to check if the graph can go through a full iteration and returns to the same initial state. SE is not suited for large graph as it takes a long time to perform the symbolic execution. Mathematical approaches have been researched to make this analysis faster. In [9], Ghamarian et al. give a necessary and sufficient condition based on the analysis of the strongly connected components of an SDF graph.

Compositionality is the property of a dataflow graph to be analyzable independently from the internal specification of the actors composing the graph. In other words, in a compositional graph, modifying the internal behavior of an actor does not affect the analyzable properties (consistency, liveness, ...) of the graph.

The popularity of the SDF MoC comes from its great analyzability. Indeed, using static analyses, the consistency and liveness properties of an SDF graph can be verified. When an SDF graph is schedulable, i.e it is consistent and live, a minimal sequence of firings of the actors exists for achieving an infinite execution with bounded memory. Such minimal sequence is called a *graph iteration* and the number of firings of each actor is given by the coefficients of the RV of the graph. Figure 1 presents an SDF graph that is consistent and live. For each graph iteration, actor A is executed 1 time, actor B 4 times, and actor C 16 times.

Static dataflow models have been proposed to extend the expressiveness and conciseness of the SDF MoC while maintaining its predictability. The Cyclo-Static Synchronous Dataflow (CSDF) MoC [10] has the same expressiveness as the SDF MoC but is more concise. The Interfaced Based Synchronous Dataflow (IBSDF) MoC [11] enforces the compositionality and expressiveness of the SDF MoC. The Parameterized DataFlow (PDF) [12], Schedulable Parametric Dataflow (SPDF) [13], and πSDF [3] are dynamic extensions of the SDF that enforce dynamic reconfigurations of dataflow graphs.

The next section presents the πSDF MoC as it is the reference MoC used to formally present the SAD meta-model (Section IV).

B. Parameterized and Interfaced Synchronous DataFlow MoC

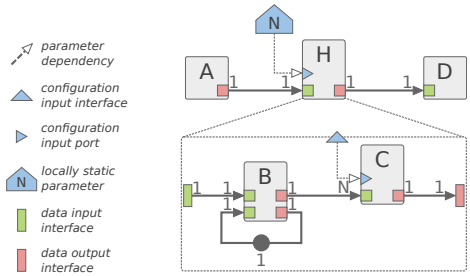


Figure 2: Example of a πSDF graph.

The Parameterized and Interfaced Synchronous DataFlow (πSDF) MoC [3] is a hierarchical and dynamically reconfigurable extension of the SDF MoC. A hierarchical actor is a dataflow actor whose internal behavior is defined by a dataflow graph. The encapsulation of sub-graphs is similar to function calls in imperative languages. Figure 2 presents an example of a hierarchical πSDF graph with the associated graphical semantics. Actor H is a hierarchical actor defined by the sub-graph formed by actors B and C.

Formally, a πSDF graph $G = (A, F, I, \Pi, \Delta)$ contains in addition to a set of actors A and a set of FIFOs F, a set of hierarchical interfaces I, a set of parameters Π, and a set of parameter dependencies Δ.

Interfaces $I = (I^{in}, I^{out})$ decorrelate the inner definition of a sub-graph from the apparent behavior of the hierarchical actor. A *source* interface $i^{in} \in I^{in}$ is a vertex transmitting to the sub-graph the data tokens received on the corresponding data input port of the corresponding hierarchical actor. If more

data tokens are consumed by the sub-graph during an iteration, the source interface behaves as a circular buffer and produces the same data tokens as many times as needed. Symmetrically, a *sink* interface $i^{out} \in I^{out}$ only transmits the last data tokens produced during a sub-graph iteration, and discards any excess of data tokens relatively to the production rate of the parent actor. These hierarchical interfaces, inherited from the IBSDF MoC [11], make the πSDF MoC a compositional MoC.

Parameters $\pi \in \Pi$ are associated with parameter values $v \in \mathbb{N}$. Parameter values can either be statically defined, or dynamically set by actors at runtime. Reconfigurability of the πSDF MoC comes directly from parameters whose values are used to influence different properties, namely the computation of an actor, the rates of the data ports of an actor, the value of another parameter and the number of delays in a FIFO. In Figure 2, parameter N sets the number of firings of actor C inside the hierarchical actor H. Thanks to the hierarchical interfaces, the value of N does not affect the number of firing of the actor H, and the graph is compositional.

C. Dataflow Delays in the Literature

Delays are defined as the initial state of a FIFO [8], which corresponds to the number of data tokens present in the FIFO when a graph iteration starts. As illustrated in Figure 3, delays are mainly used for two purposes: to pipeline actor firings, and to ensure liveness of cyclic data-paths.

(a) Example of a pipeline in an SDF graph.

(b) Example of an SDF graph with a cyclic path.

Figure 3: Examples of delay usage in dataflow graphs.

Having a *pipelining* delay on a FIFO f means that an offset exists between the iteration in which data token are produced on f, and the iteration in which these data token are consumed on f. In Figure 3a, the token produced by actor A at iteration n is consumed by actor B at iteration $n+1$.

Initial tokens inside FIFOs also prevent deadlocks in dataflow graphs that contain cyclic data paths. For example, during an iteration of the graph of Figure 3b, actor B fires only when actor A produces 1 data token, and actor A needs a data token from actor B to fire. Hence, a sufficient number of delays are needed in this cyclic data path to initiate the first actor firing, and prevent a deadlock.

Although the concept of delays exists in most dataflow MoCs, the initial values given to the corresponding data tokens is hardly mentioned, let alone specified, in the literature. When specified, initial values are set to 0 [8], [14]. The lack of specification on the initial values of delays leads to inconsistent behaviors across different programming tools. Initialization of delays is made explicit with the new semantics of delay proposed in SAD.

The persistence of the data tokens of delays across levels of hierarchy and across graph iterations also differs between MoCs. In a flat model, i.e without hierarchy, like the SDF MoC [8], the last data tokens produced during an iteration n are used for the initial conditions of iteration $n+1$. However, in hierarchical MoCs, actors can be defined by dataflow graphs. Thus, what happens to the persistence of the delays contained in a hierarchical actor ? In the IBSDF MoC [11], levels of hierarchy are clearly delimited with the use of interfaces. Therefore, in the IBSDF MoC if a hierarchical actor is fired several times per iteration of the graph to which it belongs, data parallelism makes it possible for these firings to occur in parallel. Data parallelism is the property of an actor to be independent from its input data, and thus to have multiple firings executed concurrently. In this context, if the sub-graph of a hierarchical actor contains a delay, the corresponding data tokens may not persist across firings of this hierarchical actor, as it would force the sequential execution of the parent actor. In other words, the delay is only persistent within the scope of the sub-graph.

In the DPN MoC [7], *Lee et al.* state that delays in hierarchical actors may result in non-deterministic behaviors even with a consistent and live sub-graph. *Lee et al.* propose to make delays persistent across all levels of hierarchy with implicit feedback loops around a hierarchical actor in order to maintain the precedence relationship between multiple successive firings of the actor. However, serializing the execution of an actor induces a loss of data parallelism. Losing data parallelism significantly impacts performance in graphs with deep nested hierarchy levels as it forces the serialization of the execution of the higher-level actor.

In recent dataflow-based domain-specific programming language, the semantics of delays is also problematic. In OpenVX [6], a computer vision dataflow-based programming language, delays are considered persistent across all levels of hierarchy and cyclic data path are not allowed. In TensorFlow [5], there is no explicit notion of delay. Tensors, the basic data type in TensorFlow, are considered to be globally persistent during the lifetime of the application.

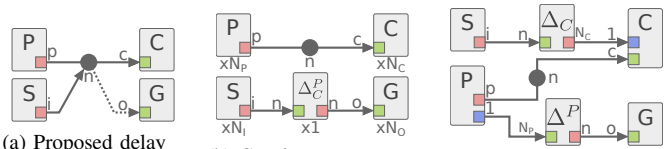
In the rest of this paper, we introduce the State-Aware Dataflow (SAD) meta-model. SAD can be used similarly to the Parameterized and Interfaced Meta-Model (PiMM) [3] or the *Parameterized DataFlow* [12] to extend the semantics of any dataflow MoC implementing a well-defined notion of graph iteration. SAD adds both explicit initialization of delays and hierarchical state awareness through the use of a customizable persistence scope of delays to the extended MoC. The next section presents the new semantics of delays of SAD and demonstrates its efficiency to model simple algorithm structures.

III. State-Aware Dataflow: Delay Semantics

The proposed semantics offers a novel scheme for initializing delays through dataflow actors. Symmetrically, the last data tokens of a delay produced during a graph iteration can be output. The proposed semantics fosters conciseness, as demonstrated in Section III-C.

A. Delays Semantics

The SAD semantics extends the definition of delays of Section II-A and is applicable to any dataflow MoC with a well-defined concept of graph iteration. In the proposed semantics, a delay $d = (f, n, c_{in}, c_{out})$ contains in addition to f and n, two optional data connections c_{in} and c_{out}. The input data connection c_{in} of the delay associates a *Setter* actor responsible for initializing the data tokens of f. The output data connection c_{out} of the delay associates a *Getter* actor receiving the last held values by the delay. The dataflow rates of c_{in} and c_{out} are such as $rate(c_{in}) = rate(c_{out}) = n$. However, the production rate of actor *Setter* and the consumption rate of actor *Getter* are not required to be equal to the rates of c_{in} and c_{out}.



(a) Proposed delay semantics.

(b) Consistency Equivalent Graph of Figure 4a.

(c) Consistency and Liveness Equivalent Graph of Figure 4a.

Figure 4: Proposed semantics of delay and equivalent graphs used for consistency and liveness analyses.

Delays are usually represented by a filled circle positioned on a FIFO as displayed in Figure 1. Figure 4a introduces a new graphical representation of delays with the additional data connections. Actors P and C are the production and the consumption actors of the delay (i.e the actors connected to f), respectively; and actors S and G are the setter and getter actors, respectively. The FIFO between the delay and the getter actor G is drawn with a dashed line to explicit which actor is the getter actor and which actor is the consumption actor.

The new data connections induce the following precedence rules in the firing sequence of actors during each graph iteration.

R1. All firings of the *Setter* actor of a delay must occur prior the first firing of the *Consumption* actor of this delay.

R2. All firings of the *Getter* actor of a delay must occur after the last firing of the *Production* actor of this delay.

On the initialization side, **the data tokens of a delay must be explicitly initialized** for the delay to be fully specified. The default initialization of the proposed semantics is to set all data tokens of a delay to zero. Explicitly initializing the delays means that new initialization tokens are produced on each graph iteration. Thus, if no actor is connected to the output connection of a delay, the produced data tokens have to be discarded to ensure bounded memory execution. Importantly, making the initialization of delays explicit for each graph iteration unambiguously removes memory persistence across graph iterations. Indeed, each graph iteration starts with initial data tokens independent from previous computations. Therefore, delays are no longer allowed to transfer data tokens from iteration n to iteration $n+1$. Section IV introduces new unambiguous semantics for modeling this persistence of data tokens across graph iterations. The next section details how the SAD delay semantics impacts the consistency, liveness, and scheduling analyses of a dataflow graph.

B. Delays Analysis: Consistency, Liveness and Scheduling

1) Analysis: A key feature of the SAD meta-model is its compatibility with state-of-the-art methods for analyzing the consistency and liveness of a graph. The proposed method for verifying the consistency and liveness of a graph consists of 4 steps.

Step 1. The analyzed dataflow graph is transformed into a Consistency Equivalent Graph (CEG). The CEG is an intermediate representation used for the consistency analysis of SAD graphs. To build a CEG, every delay with a *Setter* actor or a *Getter* actor is replaced with a delay with no c_{in} or c_{out} connection. The setter and getter actors of every delay are now connected to *virtual delay* actors. The virtual delay actor is noted Δ_C^P, with P and C being the names of the *Production* and *Consumption* actors connected to the delay, respectively. The virtual delay actor has a unique input data port and a unique output port. The rates of the data ports of the virtual delay actor are equal to the value *n* of the delay it replaces. Figure 4b illustrates the CEG transformation of the graph of Figure 4a. Actors S and G are now connected to the virtual delay actor Δ_C^P instead of the delay. The rates of the input and output data ports of Δ_C^P are equal to the value n of the delay.

Step 2. The consistency of a CEG is verified by analyzing the topology matrix Γ of the CEG using the same method as for SDF graphs [8]. The transformation into the CEG may result in disconnected graphs as illustrated by Figure 4b. Thus, the consistency of every graphs in the CEG has to be verified for the CEG to be consistent. A necessary but not sufficient condition for the liveness of a SAD graph is that every *virtual delay* actor must have a Repetition Vector (RV) value of 1, using the RV of the CEG. In Figure 4b, the RV values are noted below each actor of the CEG.

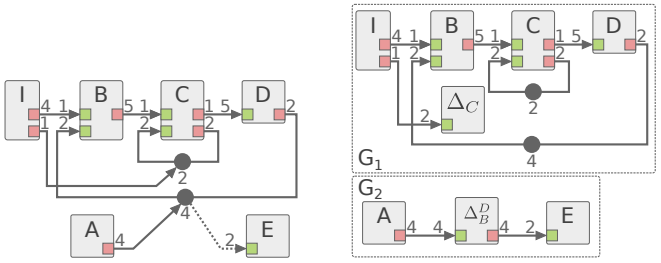
Step 3. The CEG is transformed into a Consistency and Liveness Equivalent Graph (CLEG) using the RV values computed during step 2 to verify the liveness of the original graph. The CLEG enforces and models the precedence rules **R1** and **R2** of Section III-A. The CLEG transformation splits virtual delay actors in two virtual actors and adds *virtual data ports* and FIFOs to every production and consumption actors connected to delays.

The virtual actors are illustrated in Figure 4c. Figure 4c shows the CLEG of the graph of Figure 4a. Virtual actors Δ_C and Δ_P replace the actor Δ_C^P of Figure 4b and enforce the rules **R1** and **R2**, respectively. N_P and N_C are the RV values of actors P and C in the CEG of Figure 4b and n is the number of delays. Actor Δ_C has a consumption rate of n on its input port and a production rate of N_C on its output port. Symmetrically, actor Δ^P has a consumption rate of N_P on its input port and a production rate of n on its output port. The virtual data ports of actors P and C, represented

in blue, have a production and consumption rate equal to 1, respectively. The CLEG in Figure 4c exposes both the precedence relationships and the explicit data productions and consumptions of Figure 4a.

Step 4. The liveness of the CLEG is verified using methods of the state-of-the-art such as the Symbolic Execution method [8] or the mathematical analysis in [15].

The scheduling of a graph using the proposed delays is compatible with the scheduling techniques used by current dataflow MoCs. Indeed, the CLEG gives all the dependencies between firings of actors and can be used to derive a schedule for the original graph. Note that the *virtual ports*, *actors*, and FIFO*s* are used for analysis purposes only. The virtual FIFOs do not convey actual data tokens, and the *virtual actors* have a null execution time.



(a) An SDF graph with the proposed semantics of delays.

(b) CEG of the graph of Figure 5a.

Figure 5: CEG transformation of an SDF graph using the proposed semantics of delays.

2) Example: Figure 5a presents an example of a more complex synthetic SDF graph with the newly introduced semantics for delays. The graph of Figure 5a is used as an illustration of the analyses steps presented in Section III-B1. In this graph, the delay on the (D, B) FIFO is used to avoid a deadlock and the delay on the self-loop of actor C is used to specify explicitly the transmission of a state between successive firings of actor C. The two delays of Figure 5a are initialized by actors A and I. Finally, actor E is fired by the delay of the (D, B) FIFO after the last iteration of the cycle composed of actors B, C and D. Figure 5b gives the CEG transformation of the original graph of Figure 5a. Actors A and E are now connected to the virtual delay actor Δ_B^D and actor I is connected to the virtual actor Δ_C. The original graph of Figure 5a is now split into two unconnected graphs in the CEG, namely G_1 and G_2. Thus, checking the consistency of the original graph is equivalent to checking the consistency of both graphs in the CEG. Equations 1 and 2 gives the topology matrices Γ_1 and Γ_2 of the graphs G_1 and G_2, respectively.

$$\Gamma_1 = \begin{array}{c} \\ I\Delta_C \\ IB \\ BC \\ CD \\ DB \end{array} \begin{array}{c} \begin{array}{ccccc} I & \Delta_C & B & C & D \end{array} \\ \begin{bmatrix} 1 & -2 & 0 & 0 & 0 \\ 4 & 0 & -1 & 0 & 0 \\ 0 & 0 & 5 & -1 & 0 \\ 0 & 0 & 0 & 1 & -5 \\ 0 & 0 & -2 & 0 & 2 \end{bmatrix} \end{array}, q_1 = \begin{array}{c} I \\ \Delta_C \\ B \\ C \\ D \end{array} \begin{bmatrix} 2 \\ \mathbf{1} \\ 8 \\ 40 \\ 8 \end{bmatrix} \quad (1)$$

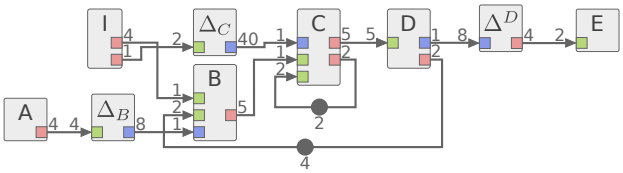


Figure 6: CLEG of the graph of Figure 5a.

$$\Gamma_2 = \begin{array}{c} \\ A\Delta_B \\ \Delta_B E \end{array} \begin{array}{c} \begin{array}{ccc} A & \Delta_B & E \end{array} \\ \begin{bmatrix} 4 & -4 & 0 \\ 0 & 4 & -2 \end{bmatrix} \end{array}, q_2 = \begin{array}{c} A \\ \Delta_B \\ E \end{array} \begin{bmatrix} 1 \\ \mathbf{1} \\ 2 \end{bmatrix} \quad (2)$$

On top of Γ_1 and Γ_2 are noted the names of the actors to which the columns of the matrices refer. On the left of Γ_1 and Γ_2 are noted the names of the FIFOs to which the lines of the matrices refer. $rank(\Gamma_1) = 4$ and $rank(\Gamma_2) = 2$, thus the graphs G_1 and G_2 are consistent and so is the graph of Figure 5a. The RVs q_1 and q_2 of respectively G_1 and G_2 give a repetition factor of 1 for both Δ_C and Δ_B actors. Figure 6 shows the CLEG of the graph of Figure 5a. Actor Δ_B is now connected to actor B through a virtual port with a production rate of $q_1(B) = 8$ and a consumption rate of 1 as specified by Section III-B1. Similarly actor Δ_C is connected to actor C with a production rate of $q_1(C) = 40$. Finally, actor D is connected to actor E and actor E has a consumption rate of 8 on this FIFO which is equal to $q_1(D)$. The CLEG of Figure 6 is live and thus the graph of Figure 5a is both consistent and live.

The next section demonstrates the advantages of the proposed semantics in term of conciseness, readability and memory usage.

C. Improved Conciseness and Memory Efficiency

In this section, we use a simple algorithmic structure to demonstrate the lack of proper semantics to expose efficiently fine grained parallelism application with the SDF MoC. In many applications, iterative computations similar to the one shown in Algorithm 1 are used. Algorithm 1 is decomposed in 3 distinct phases, namely the *prologue* (line 3), the *loop kernel* (line 4-6) and the *epilogue* (line 7).

Algorithm 1: Iterative Process Example

```
1 Input: Number of iterations N;
2        Parameter file paramFile;
3 dataBuffer = execute(P, inputs_P);
4 for i ∈ [0 : N] do
5     parameters_i = readFile(paramFile);
6     dataBuffer = execute(B, dataBuffer, parameters_i);
7 execute(E, dataBuffer);
```

The *prologue* phase is the phase initializing $dataBuffer$. The *loop kernel* is the phase corresponding to the iterative computations of the loop. Finally, the *epilogue* is the processing done after the final iteration of the loop. The 3 phases of Algorithm 1 are sequential due to the data dependency of $dataBuffer$. This data dependency is enforced at line 6 of

the loop kernel phase, where results from the previous loop iterations are used. In other words, the line 6 computation of iteration $n+1$ and iteration n are not executable in parallel. Nevertheless, parallelism can be exploited inside each of the 3 phases.

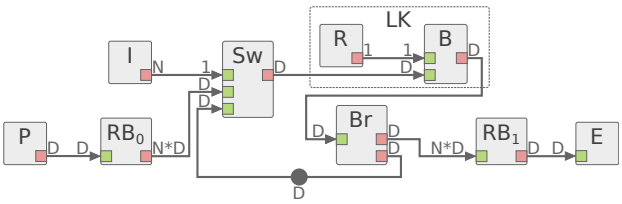


Figure 7: Equivalent SDF graph of Algorithm 1.

Figure 7 shows how Algorithm 1 is expressed in the strict SDF MoC. Note that the inner processing of the loop kernel phase is fully exposed in the strict SDF MoC. Actors P, $\{R, B\}$ and E represent the prologue, the loop kernel and the epilogue phases of Algorithm 1, respectively. The size of $dataBuffer$ is noted D and corresponds to the consumption and production rates of actor B. Actor I is used here to set the number N of iterations of the *for loop*. Actors Sw, RB_0, RB_1 and Br are special actors used to manage the loop context of Algorithm 1. RB_0 and RB_1 are used to guarantee unique execution of the prologue and epilogue phases. RB_0 duplicates N times the tokens received on its input port to its output port and symmetrically, RB_1 forwards only the last D tokens received on its input port to its output port. Sw is a *switch* actor used to select which tokens are forwarded to actor B. Since actors are stateless in the SDF MoC, the Sw actor distinguishes the first iteration from the rest of the loop based on the values produced by actor I. On the first firing of actor B, tokens produced by actor P are used. For every other firings, actor B uses the tokens produced by its previous firing through a feedback FIFO. Br is used symmetrically to Sw to forward the data tokens produced by actor B to both Sw and RB_1.

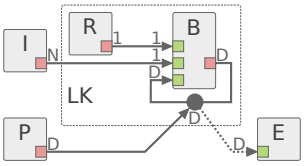


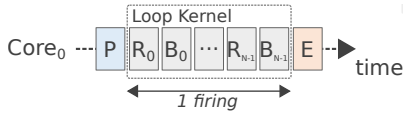
Figure 8: Equivalent SAD graph of Algorithm 1.

Figure 8 shows the representation of Algorithm 1 using the SAD delay semantics presented in Section III-A. The initialization of the delay is used as the prologue phase of Algorithm 1, then actor B is fired sequentially N times. Finally, the last data tokens produced by B are automatically forwarded to actor E through the delay. Figure 8 demonstrates the conciseness improvement offered by the SAD semantics of delay.

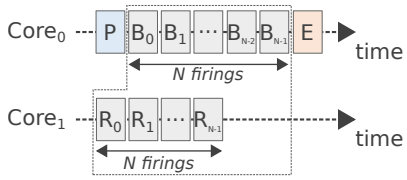
It would be possible to simplify the graph of Figure 7 by encapsulating all iterations of the loop kernel as a single firing of a unique actor. The interest of exposing multiple iteration of the loop kernel in dataflow is demonstrated by analyzing the resulting schedules of both approaches. When the iterations of the loop kernel are not exposed in dataflow, the resulting schedule is illustrated in Figure 9a. In the schedule of Figure 9a, P and E correspond respectively to the prologue and epilogue phases of Algorithm 1. R_i and B_i are the computations of lines 5 and 6 of the i^{th} iteration of the loop. The entire loop kernel is done in 1 firing of the unique actor in which R and B are encapsulated. In the schedule of Figure 9a, the total execution time of Algorithm 1 is defined by

$$T1_{critical} = T_P + N * (T_R + T_B) + T_E \quad (3)$$

where T_x is the execution time of the corresponding actor x and N the number of iterations of the loop.



(a) Sequential schedule of Algorithm 1.



(b) Pipelined schedule of Algorithm 1.

Figure 9: Schedules of Algorithm 1.

In Algorithm 1, the *readFile* function calls (line 5) are independent from the loop iteration. This independence of the executions of the actor R is naturally represented in the equivalent dataflow graphs (Figures 7 and 8) where R has no incoming dependency. Thus, it is possible to execute the N firings of actor R in parallel, without interleaving them with the N firings of actor B. Figure 9b shows the schedule of the latter scenario. Executions of actor R starts before the start of the loop and allow actor B to immediately starts its computation after the end of the prologue. The resulting total execution time of Figure 9b is defined by

$$T2_{critical} = \max(T_B + N * T_R,\ N * T_B + T_P) + T_E \quad (4)$$

Given $T1_{critical}$ and $T2_{critical}$ definitions, exposing the inner-loop parallelism gives a significantly shorter execution time. In addition, exposing such loop structures in dataflow is also relevant in the context of Field Programmable Gate Array implementation of nested loop kernels [16]. Thus, in the following, we only consider the exposed dataflow representation of the inner-loop.

In Figure 7, actors Sw, RB_0, RB_1 and Br added to manage the loop context have a non negligible impact on memory. In the graph of Figure 7, $dataBuffer$ values are stored simultaneously in 4 different FIFOs for each iteration of the loop: *(Br, Sw)*, *(Sw, B)*, *(B, Br)* and (Br, RB_1). The main issue is that the values of $dataBuffer$ are only useful for actor B but get copied 3 times. The remaining FIFOs (RB_0, Sw),

(P, RB_0) and (RB_1, E) lead to a total memory allocation of the graph for $dataBuffer$ defined by

$$M = D * (2 * N + 5) \quad (5)$$

with D the size of $dataBuffer$ and N the number of loop iterations.

In Figure 8, using the delay to manage the loop means that only 1 FIFO is needed for the entire loop structure. In the graph of Figure 7 the size of the allocated memory is dependent on the number N of iterations of the loop (Equation 5) whereas with the proposed semantics the allocated memory size is always constant and equal to the size of $dataBuffer$. In a dynamic context where the number of iterations of the loop is resolved at runtime, dynamic allocations of all the buffers combined to the memory transfer operations can have a great overhead on the performance of an application.

IV. State-Aware DataFlow: Persistence Scope

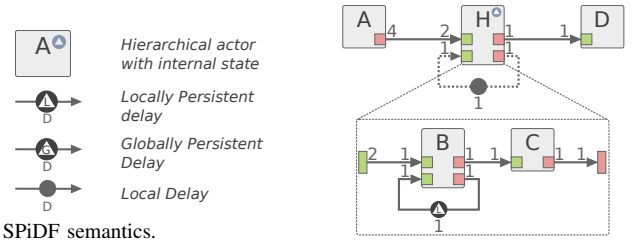
Following the concept of graph iteration, a delay in a graph G associated with an explicit initialization is initialized once per iteration of G. Therefore, *Consumption* actors of delays always have new data tokens for their first firing of each graph iteration.

Having the fixed initial conditions at every iteration of a SAD graph G means that pipelined behavior, as the one depicted in Figure 3a, are no longer modelable. To reconcile pipeline functionality with the new semantics of delay introduced in Section III, it is necessary to define unambiguous persistence scopes for delays. The persistence of delays defines whether tokens inside a delay FIFO should be discarded or preserved for the next graph iteration.

In this section we present the application of the SAD meta-model on the πSDF MoC, resulting in the State-Aware Parameterized and Interfaced DataFlow (SPiDF). The graphical semantics of the SPiDF MoC is presented in Figure 10a.

A. Persistence Scope of Delays: Semantics

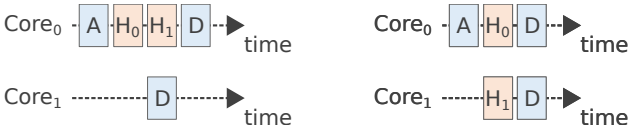
As explained in Section II-C, having a delay in a hierarchical sub-graph G_H of an actor H induces an internal state for H. Such a state can either be discarded at the end of the firing of H or preserved for the next firing. In order to preserve the state of H, it is necessary to expand the persistence scope of the delay outside of the sub-graph G_H. Preserving the state of H induces a precedence relationship between its successive firings. An actor with a persistent state across graph iterations also constitutes a state for its parent actor, which in turns is considered as having a state, and must have serialized firings. Hence, expanding the persistence scope of a delay to all levels of hierarchy, as proposed in [7], induces a precedence relationship between every parent graphs of H. Having such strong constraint on the firing sequences of actors becomes problematic in complex applications with deep hierarchy as it will strongly undermine the data parallelism of the application. To control the persistence scope of a state in a hierarchical graph, SAD introduces 3 different types of



(a) SPiDF semantics.

(b) Example of a SPiDF graph with Locally Persistent Delay.

Figure 10: SPiDF graph example and associated semantics.



(a) Schedule of the graph in Figure 10b with a LPD.

(b) Schedule of the graph in Figure 10b with a LD.

Figure 11: Schedules of the graph in Figure 10b.

delay illustrated in Figure 10a: *Local Delay*, *Locally Persistent Delay* and *Globally Persistent Delay*.

Local Delays (LDs) use the semantics presented in Section III-A. Thus, an LD can be initialized dynamically by dataflow actors. The data tokens contained in the FIFO of an LD are preserved within the scope of a unique graph iteration.

Locally Persistent Delays (LPDs) are delays whose data tokens persist outside of the scope of the graph to which the LPD belongs. An LPD specifies the persistence of a delay for one level of hierarchy and establishes a precedence relationship for successive firing of the parent actor H of the subgraph G_H to which the LPD belongs.

Globally Persistent Delays (GPDs) are LPDs that persist across all levels of hierarchy up to the top-level graph. GPDs are initialized only once in the lifetime of an application, prior to the first firing of the top-level graph. Since dataflow actors are fired once per graph iteration, they cannot be used to initialize a GPD once in the application lifetime. Therefore, a GPD is initialized with a function or a constant value directly associated with the delay. GPDs are equivalent to the delays described in [7]. **By default, any LPD in the top-level of hierarchy is a GPD.**

An LPD can be seen as a feedback loop with an LD around the hierarchical actor to which it belongs. Thus, the LD being explicitly apparent around the hierarchical actor, it is possible to extend furthermore the original persistence scope by promoting the LD into an LPD or to initialize the data tokens of the delay with an actor. This visibility of LPDs is optional in order to maintain good readability of graphs. Moreover, no feedback loops are needed for GPDs due to their persistence across all levels of hierarchy. Hierarchical actors containing persistent delays are represented as in Figure 10a. Figures 10b presents an example of hierarchical graph using the proposed semantics for persistence of delays.

In the graph of Figure 10b, the delay inside the hierarchical actor H is defined as an LPD. In Figure 10b, the persistence of the LPD is made explicit with a feedback loop around actor H. The resulting schedule is shown in Figure 11a. Figure 11b shows the schedule of the same hierarchical graph with an LD in actor H instead of an LPD. Note that using an LPD serializes the firings of actor H (Figure 11a) whereas the LD allows the multiple firings of H to occur concurrently (Figure 11b). The example of Figure 10b illustrates the efficiency of the explicit persistence semantics for delay. Indeed, the persistence of delays offered by the SPiDF MoC lead to controlled data parallelism in hierarchical graphs which can be taken into account during the analysis and scheduling of the graphs. Unambiguous persistence semantics enforces the compositionality of hierarchical actors and their reusability.

B. Analysis of Persistence in Reconfigurable Hierarchical Graphs

In reconfigurable and hierarchical dataflow MoCs such as the πSDF MoC, a graph can dynamically configure parameters of hierarchical sub-graphs. Specifically, a graph can change the number of delays of a sub-graph at runtime. Changing the number of delays of a graph affects the liveness and the consistency of the graph as mentioned in Section III. Thus, the number of delays must be known when the liveness and consistency analysis of a graph are verified.

In SAD, the number of data tokens of a persistent delay can not depend on a parameter located in a level of hierarchy below the highest level of hierarchy in which the delay persists.

The SPiDF MoC inherits the semantics, the compositionality, and the schedulability properties of the πSDF MoC [3]. Thus, in the SPiDF MoC, the static parameter tree inherited from the πSDF MoC naturally enforces this property.

V. Use Case: CACLA Update Parameters

In this section we use the Continuous Actor Critic Learning Automaton (CACLA) algorithm [17] as an application example to demonstrate the conciseness and memory efficiency of the SPiDF MoC. CACLA is part of the reinforcement learning branch of machine learning. Reinforcement learning consists of learning the model of an environment *E* and taking actions accordingly without prior knowledge of *E*. The reinforcement learning algorithm learns the model of *E* based on an abstract representation of *E* called the state (*S*) of the environment. For instance, in the case of the control of a robotic arm, the environment *E* is the arm, the state *S* would be the position and velocity of each motor, and the actions would be the commands of the motors of the arm. Due to a lack of space, the whole application is not detailed here, but it is available in the PREESM tool open-source repository [4].

CACLA uses multiple neural networks to make predictions and the coefficients of the neural networks are updated at each iteration of the main *loop* of the algorithm. Figure 12 illustrates the sub-graph responsible for the update of the coefficients of the neural network that predicts the actions to apply to *E*. The graph takes as inputs the coefficients of the neural network

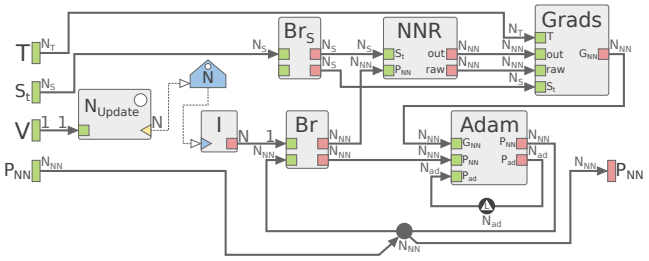


Figure 12: SPiDF graph of the CACLA actor update algorithm. N_x is the size of the associated x parameter.

P_{NN}, the target T toward which the network is updated, the state S_t of the environment and the variance V of the temporal difference error [17]. The graph produces as output the updated coefficients of the neural network P_{NN}. The *NNR*, *Adam* and *Grads* actors are hierarchical actors whose internal behavior are not detailed here.

In the graph of Figure 12, the coefficients of the neural network P_{NN} are updated iteratively N times, N being set by the configuration actor N_{Update}. The *NNR* actor is a Multi Layer Perceptron (MLP) neural network that takes the state of the environment S_t as input and predicts the action based on current coefficients of the network. The *Grads* actor computes the gradients of each coefficient of the neural network based on a target T and the outputs of *NNR*. Finally, the *Adam* actor apply the gradients on the parameters using the Adam optimizer algorithm [18]. The graph of Figure 12 uses both semantics introduced by the SAD meta-model. The Locally Persistent Delay is used to store the hyper parameters and coefficients needed by the Adam algorithm [18]. The Local Delay is used for the iterative update of the coefficients of the neural network.

Table I shows the difference in memory usage between the SPiDF and a strict πSDF implementation of the CACLA algorithm [17]. The memory usage is defined as being the amount of allocated memory needed to run the application. Table I presents the difference in memory usage for both the full CACLA application and the sub-graph of the update of the neural network of Figure 12. The memory comparison is based on a neural networks with 3 layers and a total of 101 parameters per network, where each parameter is encoded on 4 bytes. A gain of 35.4% in total memory usage is observed for the whole application with a gain of 56.53% of memory usage for the update graph alone. The update of the neural network is the part of CACLA benefiting the most from the new semantics due to the use of a persistent delay and due to the iterative loop needed for the update. Indeed, in the strict πSDF MoC, making a delay persistent across levels of hierarchy induce the need of extra actors similarly to Figure 7. The results of Table I show that for applications with iterative computations and with locally persistent data, the new semantics of SAD can drastically reduce the memory footprint of the application. Thus, SAD is particularly well suited for modeling applications on embedded platforms with sparse ressources.

Table I: Comparison of memory usage of SPiDF and πSDF implementations of the CACLA algorithm.

	πSDF	SPiDF	Gain
Update of neural network (in bytes)	9716	**4430**	$56,53\%$
Full application (in bytes)	27492	**17720**	$35,4\%$

VI. CONCLUSION

This paper has proposed a new dataflow meta-model called the State-Aware Dataflow (SAD) with explicit semantics of delays. SAD can be applied to a wide range of dataflow MoCs to extend their expressivity and conciseness, while preserving the analysis tools of the extended MoC. We have shown that SAD is well suited to expose the fine-grain parallelism of nested loops with less memory overhead compared to state-of-the-art dataflow MoCs. SAD brings functional aspect of an application into the model space by expliciting initial conditions and persistence of hierarchical graph at any given point in time, thus ensuring independence of the model from its implementation. Finally, we have demonstrated the conciseness and memory efficiency of SAD through a reinforcement learning example application. Future work will build on the new semantics of delays of SAD for optimizing loops structure in dataflow based on state-of-the-art methods such as polyhedral transformations.

REFERENCES

[1] B. D. Theelen, M. C. Geilen, T. Basten, J. P. Voeten, S. V. Gheorghita, and S. Stuijk, "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," in *Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings. Fourth ACM and IEEE International Conference on*. IEEE, 2006, pp. 185–194.

[2] C. Park, J. Chung, and S. Ha, "Extended Synchronous Dataflow for Efficient DSP System Prototyping." IEEE, Jun. 1999.

[3] K. Desnos, M. Pelcat, J.-F. Nezan, S. S. Bhattacharyya, and S. Aridhi, "Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*. IEEE, 2013, pp. 41–48.

[4] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming," in *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*. IEEE, 2014, pp. 36–40.

[5] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," 2016, pp. 265–283.

[6] K. Group, "The OpenVX API for hardware acceleration," in *http://www.khronos.org/openvx*, 2013.

[7] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

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

[9] A. H. Ghamarian, M. C. W. Geilen, T. Basten, B. D. Theelen, M. R. Mousavi, and S. Stuijk, "Liveness and boundedness of synchronous data flow graphs," in *Formal Methods in Computer Aided Design, 2006. FMCAD'06*. IEEE, 2006, pp. 68–75.

[10] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cycle-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, Feb. 1996. [Online]. Available: http://ieeexplore.ieee.org/document/485935/

[11] J. Piat, S. S. Bhattacharyya, and M. Raulet, "Interface-based hierarchy for synchronous data-flow graphs," in *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*. IEEE, 2009, pp. 145–150.

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

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

[14] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization, Second Edition*, ser. Signal Processing and Communications. CRC press, 2009.

[15] O. Marchetti and A. Munier-Kordon, "A sufficient condition for the liveness of weighted event graphs," *European Journal of Operational Research*, vol. 197, no. 2, pp. 532–540, Sep. 2009. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0377221708005900

[16] M. Milford and J. McAllister, "Constructive Synthesis of Memory-Intensive Accelerators for FPGA From Nested Loop Kernels," *IEEE Transactions on Signal Processing*, vol. 64, no. 16, pp. 4152–4165, Aug. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7468564/

[17] H. Van Hasselt and M. A. Wiering, "Reinforcement learning in continuous action spaces," in *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*. IEEE, 2007, pp. 272–279.

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.