PiMM: Parameterized and Interfaced Dataflow Meta-Model for MPSoCs Runtime Reconfiguration

Karol Desnos, Maxime Pelcat, Jean-François Nezan
IETR, INSA Rennes, CNRS UMR 6164, UEB
Rennes, France
email: kdesnos, mpelcat, jnezan@insa-rennes.fr

Shuvra S. Bhattacharyya
University of Maryland
College Park, USA
email: ssb@umd.edu

Slaheddine Aridhi
Texas Instruments France
Villeneuve Loubet, France
email: saridhi@ti.com

***Abstract*—Dataflow models of computation are widely used for the specification, analysis, and optimization of Digital Signal Processing (DSP) applications. In this paper a new meta-model called PiMM is introduced to address the important challenge of managing dynamics in DSP-oriented representations.**

PiMM extends a dataflow model by introducing an explicit parameter dependency tree and an interface-based hierarchical compositionality mechanism. PiMM favors the design of highly-efficient heterogeneous multicore systems, specifying algorithms with customizable trade-offs among predictability and exploitation of both static and adaptive task, data and pipeline parallelism. PiMM fosters design space exploration and reconfigurable resource allocation in a flexible dynamic dataflow context.

I. Introduction

Dataflow Models of Computation (MoCs) can be used to specify a wide range of Digital Signal Processing (DSP) applications such as video decoding [1], telecommunication [2], and computer vision [3] applications. The popularity of dataflow MoCs is due to their great analysability and their natural expressivity of the parallelism of a DSP application which make them particularly suitable to exploit the parallelism offered by heterogeneous Multiprocessor Systems-on-Chips (MPSoCs). The increasing complexity of DSP applications leads to the continuing introduction of new dataflow MoCs, and the extension of previously developed MoCs for different types of modeling contexts.

Representing an application with a Dataflow Process Network (DPN) [4] consists of dividing this application into persistent processing entities, named actors, connected by First In, First Out data queues (FIFOs). An actor performs processing (it "fires") when its incoming FIFOs contain enough data tokens. The number of data tokens consumed and produced by an actor for each firing is given by a set of firing rules [4]. Firing rules can be static or they can depend on data, as in the CAL language [5], or on parameters, as in the Parameterized SDF (PSDF) MoC [6].

In this paper, we propose a new meta-model called Parameterized and Interfaced dataflow Meta-Model (PiMM) that extends the semantics of a targeted dataflow MoC to enable the specification of dynamically reconfigurable DSP applications. PiMM builds on a novel integration of two previously developed dataflow modeling techniques called parameterized dataflow [6] and interface-based dataflow [7].

PiMM extends the semantics of a targeted dataflow MoC by introducing explicit parameters and a parameter dependency tree. Parameters can influence, both statically and dynamically, different properties of a DPN such as the firing rules of actors. PiMM also adds to the targeted MoC an interface-based hierarchy mechanism that enforces the compositionality of the extended model and improves its predictability by restricting the scope of its parameters and by enabling a top-down parameterization. In this paper, the capabilities of the PiMM meta-model are demonstrated by applying it to the Synchronous Dataflow (SDF) MoC (Section III).

Previous work on dataflow MoCs is presented in Section II and the semantics of PiMM is formally introduced in Section III. Section IV presents an analysis of the model behavior and Section V compares PiMM with existing dataflow MoCs. Finally, an application example from the LTE telecommunication standard is presented in Section VI.

II. Background and Related Work

A. Static Dataflow MoCs

Synchronous Dataflow (SDF) [8] is the most commonly used DPN MoC. Production and consumption token rates set by firing rules are fixed scalars in an SDF graph. A static analysis of an SDF graph ensures consistency and schedulability properties that imply deadlock-free execution and bounded FIFO memory needs.

An SDF graph $G = (A, F)$ (Figure 1) contains a set of actors A that are interconnected by a set of FIFOs F. An actor $a \in A$ comprises a set of data ports $(P^{in}_{data}, P^{out}_{data})$ where P^{in}_{data} and P^{out}_{data} respectively refer to a set of data input and output ports, used as anchors for FIFO connections. Functions $src : F \rightarrow P^{out}_{data}$ and $snk : F \rightarrow P^{in}_{data}$ associate source and sink ports to a given FIFO and a data rate is specified for each port by the function $rate : P^{in}_{data} \cup P^{out}_{data} \rightarrow \mathbb{N}$ corresponding to the fixed firing rules of an SDF actor. A delay $d : F \rightarrow \mathbb{N}$ is set for each FIFO, corresponding to a number of tokens initially present in the FIFO.

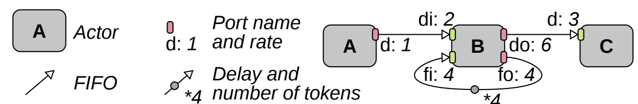

Fig. 1. Example of an SDF Graph

If an SDF graph is consistent and schedulable, a fixed sequence of actor firings can be repeated indefinitely to execute the graph, and there is a well defined concept of a minimal

sequence for achieving an indefinite execution with bounded memory. Such a minimal sequence is called *graph iteration* and the number of firings of each actor in this sequence is given by the graph Repetition Vector (RV).

Graph consistency means that no FIFO accumulates tokens indefinitely when the graph is executed (preventing FIFO overflow). Consistency can be proved by verifying that the graph topology matrix has a non-zero vector in its null space [8]. When such a vector exists, it gives the RV for the graph. The topology of an SDF graph characterizes actor interconnections as well as token production and consumption rates on each FIFO. A graph is schedulable if and only if it is consistent and has enough initial tokens to execute the first graph iteration (preventing deadlocks by FIFO underflow).

Research on dataflow modeling leads to the continuing introduction of new dataflow models. Static extensions of the SDF model such as the Cyclo-Static Dataflow (CSDF) [9], the multidimensional SDF [10], the Interface-Based Synchronous Dataflow (IBSDF) [7], and the Affine Dataflow (ADF) [11] have been proposed to enhance its expressiveness and conciseness while preserving its predictability. The Compositional Temporal Analysis (CTA) model is a non-executable timed abstraction of the SDF MoC that can be used to analyze efficiently the schedulability and the temporal properties of applications [12]. The IBSDF and the CTA models both enforce the compositionality of applications. A model is compositional if the properties (schedulability, deadlock freeness, ...) of an application graph composed of several sub-graphs are independent from the internal specifications of these subgraphs [13].

B. Interface-Based Synchronous Dataflow MoC

Interface-Based Synchronous Dataflow (IBSDF) [7] is a hierarchical extension of the SDF model interpreting hierarchy levels as code closures. IBSDF fosters subgraph composition, making subgraph executions equivalent to imperative language function calls. IBSDF has proved to be an efficient way to model dataflow applications [2]. IBSDF interfaces are inherited by the PiMM meta-model proposed in this paper (Section III).

In addition to the SDF semantics, IBSDF adds interface elements to insulate levels of hierarchy in terms of schedulability analysis. An IBSDF graph $G = (A, F, I)$ contains a set of interfaces $I = (I_{data}^{in}, I_{data}^{out})$ (Figure 2).

A *data input interface* $i_{data}^{in} \in I_{data}^{in}$ in a subgraph is a vertex transmitting to the subgraph the tokens received by its corresponding data input port. If more tokens are consumed on a data input interface than the number of tokens received on the corresponding data input port, the data input interface behaves as a circular buffer, producing the same tokens several times.

A *data output interface* $i_{data}^{out} \in I_{data}^{out}$ in a subgraph is a vertex transmitting tokens received from the subgraph to its corresponding data output port. If a data output interface receives too many tokens, it will behave like a circular buffer and output only the last pushed tokens.

[7] details the behavior of IBSDF data input and output interfaces as well as the IBSDF properties in terms of compositionality and schedulability checking. Through PiMM,

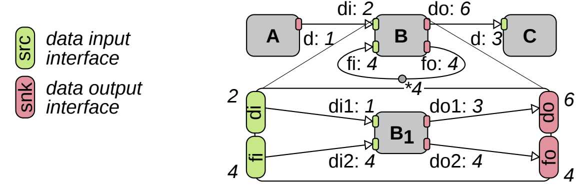

Fig. 2. Example of an IBSDF Graph

interface-based hierarchy can be applied to other dataflow models than SDF with less restrictive firing rules.

C. Parameterized Dataflow MoCs

Parameterized dataflow is a meta-modeling framework introduced in [6] that is applicable to all dataflow MoCs that present graph iterations. When this meta-model is applied, it extends the targeted MoC semantics by adding dynamically reconfigurable hierarchical actors. A reconfiguration occurs when values are dynamically assigned to the parameters of a reconfigurable actor, causing changes in the actor computation and in the production and consumption rates of its data ports. As presented in [14], reconfigurations can only occur at certain points, namely *quiescent points*, during the execution of a graph in order to ensure the runtime integrity of the application.

In *parameterized dataflow*, each hierarchical actor is composed of 3 subgraphs, namely the init ϕ_i, the subinit ϕ_s, and the body ϕ_b subgraphs.

The ϕ_i subgraph sets parameter values that can influence both the production and consumption rates on the ports of the hierarchical actor and the topology of the ϕ_s and ϕ_b subgraphs. The ϕ_i subgraph is executed only once per iteration of the graph to which its hierarchical actor belongs and can neither produce nor consume data tokens.

The ϕ_s subgraph sets the remaining parameter values required to completely configure the topology of the ϕ_b subgraph. The ϕ_s subgraph is executed at the beginning of each firing of the hierarchical actor. It can consume data tokens on input ports of the hierarchical actor but can not produce data tokens.

The ϕ_b subgraph is executed when its configuration is complete, right after the completion of ϕ_s. The body subgraph behaves as any graph implemented with the MoC to which the *parameterized dataflow* meta-model was applied.

PSDF is the MoC obtained by applying the *parameterized dataflow* meta-model to the SDF MoC. It has been shown to be an efficient way to prototype streaming applications [15]. The objective of PiMM is to further improve parameterization compared to *parameterized dataflow* by introducing an explicit parameter dependency tree and by enhancing graph compositionality. Indeed, in a PSDF graph, ports are simple connectors between data FIFOs that do not insulate levels of hierarchy (Section II-B). For example, in the PSDF graph presented in Figure 6 the consumption rate on the hierarchical input port *symbols* depends on the RV of the actor subgraphs.

Other parameterized dataflow MoCs were previously developed such as the Scenario-Aware Dataflow (SADF) [1], an

analysis-oriented model based on a probabilistic description of the dynamic firing rules of actors; or the Compaan generated KPN (CPN) [3], a parameterized extension of the Kahn Process Network (KPN) MoC. In these models, the complexity of the parameterization mechanism is handled by actors that can reconfigure the firing rules of other actors (or their own) via "control channels". This reconfiguration mechanism differs from that of PiMM in that the latter relies on the explicit definition of parameters and their dependencies which enables a precise specification of what is influenced by a parameter, even in multiple levels of hierarchy, leading to an enhanced predictability and quasi-static scheduling potential for the model (Section VI-B).

III. Parameterized and Interfaced Dataflow Meta-Modeling

The Parameterized and Interfaced dataflow Meta-Model (PiMM) can be used similarly to the *parameterized dataflow* to extend the semantics of all dataflow MoCs implementing the concept of graph iteration. PiMM adds both interface-based hierarchy and an explicit parameter dependency tree to the semantics of the extended MoC.

In this section we formally present PiMM through its application to the SDF MoC. The model resulting from this application is called Parameterized and Interfaced Synchronous Dataflow (πSDF) or (PiSDF). The pictograms associated to the different elements of the πSDF semantics are presented in Figures 3 and 4.

A. ΠSDF Semantics

A πSDF graph $G = (A, F, I, \Pi, \Delta)$ contains, in addition to the SDF actor set A and FIFO set F, a set of hierarchical interfaces I, a set of *parameters* Π, and a set of *parameter dependencies* Δ.

1) Parameterization semantics:
A parameter $\pi \in \Pi$ is a vertex of the graph associated to a parameter value $v \in \mathbb{N}$ that is used to configure elements of the graph. For a better analyzability of the model, a parameter can be restricted to take only values of a finite subset of $\mathbb{N}$. A *configuration* of a graph is the assignation of parameter values to all parameters in Π.

An actor $a \in A$ is now associated to a set of ports (P^{in}_{data}, P^{out}_{data}, P^{in}_{cfg}, P^{out}_{cfg}) where P^{in}_{cfg} and P^{out}_{cfg} are a set of configuration input and output ports respectively. A configuration input port $p^{in}_{cfg} \in P^{in}_{cfg}$ of an actor $a \in A$ is an input port that depends on a parameter $\pi \in \Pi$ and can influence the computation of a and the production/consumption rates on the dataflow ports of a. A configuration output port $p^{out}_{cfg} \in P^{out}_{cfg}$ of an actor $a \in A$ is an output port that can dynamically set the value of a parameter $\pi \in \Pi$ of the graph (Section III-B3).

A parameter dependency $\delta \in \Delta$ is a directed edge of the graph that links a parameter $\pi \in \Pi$ to a graph element influenced by this parameter. Formally a parameter dependency δ is associated to the two functions $setter : \Delta \to \Pi \cup P^{out}_{cfg}$ and $getter : \Delta \to \Pi \cup P^{in}_{cfg} \cup F$ which respectively give the source and the target of δ. A parameter dependency set by a configuration output port $p^{out}_{cfg} \in P^{out}_{cfg}$ of an actor $a \in A$ can only be received by a parameter vertex of the graph that will dispatch the parameter value to other graph elements, building a parameter dependency tree. **Dynamism in PiMM relies on parameters whose values can be used to influence one or several of the following properties: the computation of an actor, the production/consumption rates on the ports of an actor, the value of another parameter, and the delay of a FIFO** (Section II-A). In PiMM, if an actor has all its production/consumption rates set to 0, it will not be executed.

A parameter dependency tree $T = (\Pi, \Delta)$ is formed by the set of parameters Π and the set of parameter dependencies Δ. The parameter dependency tree T is similar to a set of combinational relations where the value of each parameter is resolved virtually instantly as a function of the parameters it depends on. This parameter dependency tree is in contrast to the precedence graph (A, F) where the firing of the actors is enabled by the data tokens flowing on the FIFOs.

2) πSDF hierarchy semantics:
The hierarchy semantics used in πSDF inherits from the interface-based dataflow introduced in [7] and presented in Section II-B. In πSDF, a hierarchical actor is associated to a unique πSDF subgraph. The set of interfaces I of a subgraph is extended as follows: $I = (I^{in}_{data}, I^{out}_{data}, I^{in}_{cfg}, I^{out}_{cfg})$ where I^{in}_{cfg} is a set of *configuration input interfaces* and I^{out}_{cfg} a set of *configuration output interfaces*.

Configuration input and output interfaces of a hierarchical actor are respectively seen as a configuration input port $p^{in}_{cfg} \in P^{in}_{cfg}$ and a configuration output port $p^{out}_{cfg} \in P^{out}_{cfg}$ from the upper level of hierarchy (Section III-A1).

From the subgraph perspective, a *configuration input interface* is seen as a locally static parameter whose value is left undefined.

A *configuration output interface* enables the transmission of a parameter value from the subgraph of a hierarchical actor to upper levels of hierarchy. In the subgraph, this parameter value is provided by a FIFO linked to a data output port p^{out}_{data} of an actor that produces data tokens with values $v \in \mathbb{N}$. In cases where several values are produced during an iteration of the subgraph, the configuration output interface behaves like a data output interface of size 1 and only the last value written will be produced on the corresponding configuration output port of the enclosing hierarchical actor (Section II-B).

Figure 3 presents an example of a static πSDF description. Compared to Figure 2, it introduces parameters and parameter dependencies that compose a PiMM parameter dependency tree. The modeled example illustrates the modeling of a test bench for an image processing algorithm. In the example, one token corresponds to a single pixel in an image. Images are read, pixel by pixel, by actor A and stored, pixel by pixel, by actor C. A whole image is processed by one firing of actor B. A feedback edge with a delay stores the previous image for comparison with the current one. Actor B is refined by an actor B_1 processing one Nth of the image. In Figure 3, the size of the image $picsize$ and the parameter N are locally static.

B. ΠSDF Reconfiguration

As introduced in [14], the frequency with which the value of a parameter is changed influences the predictability of the

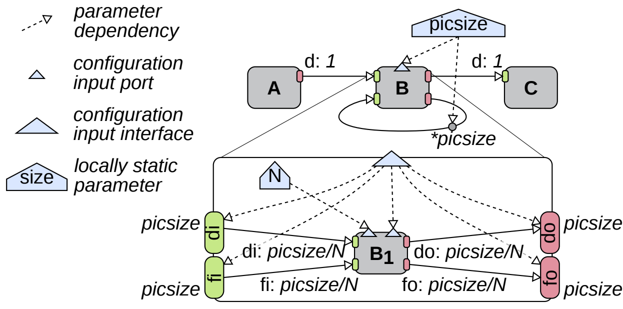

Fig. 3. Example of a πSDF Graph with Static Parameters

application. A constant value will result in a high predictability while a value which changes at each iteration of a graph will cause many reconfigurations, thus lowering the predictability.

There are two types of parameters $\pi \in \Pi$ in πSDF: configurable parameters and locally static parameters. Both restrict how often the value of the parameter can change. Regardless of the type, a parameter must have a constant value during an iteration of the graph to which it belongs.

1) Configurable parameters:
A configurable parameter $\pi_{cfg} \in \Pi$ is a parameter whose value is dynamically set once at the beginning of each iteration of the graph to which it belongs. Configurable parameters can influence all elements of their subgraph except the production/consumption rates on the data interfaces I_{data}^{in} and I_{data}^{out}. As explained in [6], [14], this restriction is essential to ensure that, as in IBSDF, a parent graph has a consistent view of its actors throughout an iteration, even if the topology may change between iterations.

The value of a configurable parameter can either be set through a parameter dependency coming from an other configurable parameter or through a parameter dependency coming from a configuration output port p_{cfg}^{out} of a *configuration actor* (Section III-B3). In Figure 4, N is a configurable parameter.

2) Locally static parameters:
A locally static parameter $\pi_{stat} \in \Pi$ of a graph has a value that is set before the beginning of the graph execution and which remains constant over one or several iterations of this graph. In addition to the properties listed in Section III-A1, a locally static parameter belonging to a subgraph can also be used to influence the production and consumption rates on the I_{data}^{in} and I_{data}^{out} interfaces of its hierarchical actor.

The value of a locally static parameter can be statically set at compile time, or it can be dynamically set by configurable parameters of upper levels of hierarchy via parameter dependencies. For example, a subgraph sees a configuration input interface as a locally static parameter but this interface can take different values at runtime if its corresponding configuration input port is connected to a configurable parameter. In Figure 4, $picsize$ is a locally static parameter both in main graph and in subgraph B.

A *partial configuration state* of a graph is reached when the parameter values of all its locally static parameters are set. Hierarchy traversal of a hierarchical actor is possible only when the corresponding subgraph has reached a partial configuration state.

A *complete configuration state* of a graph is reached when the values of all its parameters (locally static and configurable) are set. If a graph does not contain any configurable parameter, its partial and complete configurations are equivalent. Only when a graph is completely configured is it possible to check its consistency, compute a schedule, and execute it.

3) Configuration Actors:
A firing of an actor a with a configuration output port p_{cfg}^{out} produces a parameter value that can be used via a parameter dependency δ to dynamically set a configurable parameter π (Section III-B1), provoking a reconfiguration of the graph elements depending on π. In PiMM, such an actor is called a *configuration actor*. The execution of a *configuration actor* is the cause of a reconfiguration and must consequently happen only at quiescent points during the graph execution, as explained in [14]. To ensure the correct behavior of πSDF graphs, a *configuration actor* $a_{cfg} \in A$ of a subgraph G is subject to the following restrictions:

R1. a_{cfg} must be fired exactly once per iteration of G before the firing of any non-configuration actor. Indeed, G reaches a complete configuration only when all its configuration actors have fired.
R2. a_{cfg} must consume data tokens only from hierarchical interfaces of G and must consume all available tokens during its unique firing.
R3. The production/consumption rates of a a_{cfg} can only depend on locally static parameters of G.
R4. Data tokens produced by a_{cfg} are seen as a data input interface by other actors of G. (i.e. they are made available using a ring-buffer and can be consumed more than once).

These restrictions naturally enforce the local synchrony conditions of *parameterized dataflow* defined in [6] and reminded in Section IV-A.

The firing of all configuration actors of a graph is needed to obtain a complete configuration of this graph. Consequently, configuration actors will always be executed before other (non-configuration) actors of the graph to which they belong. Configuration actors are the only actors whose firing is not data-driven but driven by hierarchy traversal.

The sets of configuration and non-configuration actors of a graph are respectively equivalent to the subinit ϕ_s and the body ϕ_b subgraphs of *parameterized dataflow* [6]. Nevertheless, configuration actors provide more flexibility than subinit graphs as they can produce data tokens that will be consumed by non-configuration actors of their graph. The init subgraph ϕ_i has no equivalent in PiMM as its responsibility, namely the configuration of the production/consumption rates on the actor interfaces, is performed by configuration input interfaces and parameter dependencies.

Figure 4 presents an example of a πSDF description with reconfiguration. It is a modified version of the example in Figure 3 presented in Section III-A2. In Figure 4, the parameter N is a configurable parameter of subgraph B, while the parameter $picsize$ is a locally static parameter. The number of firings of actor B_1 for each firing of actor B is dynamically configured by the configuration actor $setN$. In this example, the dynamic reconfiguration dynamically adapts the number N

of firings of B_1 to the number of cores available to perform the computation of B. Indeed, since B_1 has no self-loop FIFO, the N firings of B_1 can be executed concurrently.

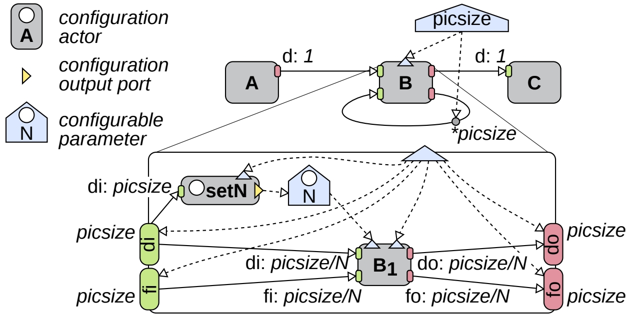

Fig. 4. Example of a πSDF Graph with Reconfiguration

IV. Model Analysis and Behavior

The πSDF MoC presented in Section III is dedicated to the specification of applications with both dynamic and static parameterizations. This dual degree of dynamism implies a two-step analysis of the behavior of applications described in πSDF: a compile time analysis and a runtime analysis. In each step a set of properties of the application can be checked, such as the consistency, the deadlock freeness, and the boundedness. Other operations can be performed during one or both steps of the analysis such as the computation of a schedule or the application of graph transformation to enhance the performance of the application.

A. Compile Time Schedulability Analysis

πSDF inherits its schedulability properties both from the interface-based dataflow modeling and the *parameterized dataflow* modeling.

In interface-based dataflow modeling, as proved in [7], a (sub)graph is schedulable if its precedence SDF graph (A, F) (excluding interfaces) is consistent and deadlock-free. When a πSDF graph reaches a complete configuration, it becomes equivalent to an IBSDF graph. Given a complete configuration, the schedulability of a πSDF graph can thus be checked using the same conditions as in interface-based dataflow.

In *parameterized dataflow*, the schedulability of a graph can be guaranteed at compile time for certain applications by checking their *local synchrony* [6]. A PSDF (sub)graph is locally synchronous if it is schedulable for all reachable configurations and if all its hierarchical children are locally synchronous. As presented in [6], a PSDF hierarchical actor composed of three subgraphs ϕ_i, ϕ_s and ϕ_b must satisfy the 5 following conditions in order to be locally synchronous:

1. ϕ_i, ϕ_s and ϕ_b must be locally synchronous, i.e. they must be schedulable for all reachable configurations.
2. Each invocation of ϕ_i must give a unique value to parameter set by this subgraph.
3. Each invocation of ϕ_s must give a unique value to parameter set by this subgraph.
4. Consumption rates of ϕ_s on interfaces of the hierarchical actor cannot depend on parameters set by ϕ_s.
5. Production/consumption rates of ϕ_b on interfaces of the hierarchical actor cannot depend on parameters set by ϕ_s.

The last four of these conditions are naturally enforced by the πSDF semantics presented in Section III. However, the schedulability condition number 1., which states that all subgraphs must be schedulable for all reachable configurations, cannot always be checked at compile time. Indeed, since values of the parameters are freely chosen by the application developer, non-schedulable graphs can be described. It is the responsibility of the developer to make sure that an application will always satisfy the schedulability condition; this responsibility is similar to that of writing non-infinite loops in imperative languages.

πSDF inherits from PSDF the possibility to derive quasi-static schedules at compile time for some applications. A quasi-static schedule is a schedule that statically defines part of the scheduling decisions but also contains parameterized parts that will be resolved at runtime. An example of quasi-static schedule is given for the application case in Section VI.

B. Runtime Operational Semantics

Based on the πSDF semantics presented in Section III, the execution of a subgraph $G = (A, F, I, \Pi, \Delta)$ associated to a hierarchical actor a_G follows the following steps. The execution of G restarts from step 1 each time the parent graph of a_G begins a new iteration.

1) Wait for a partial configuration of G, i.e. wait for all configuration input interfaces in I^{in}_{cfg} to receive a parameter value.
2) Compute the production and consumption rates on the data interfaces I^{in}_{data} and I^{out}_{data} using the partial configuration.
3) Wait until a_G is fired by its parent graph. (enough data tokens must be available on the FIFOs connected to the data input ports P^{in}_{data} of a_G.)
4) Fire the *configuration actors* of A that will set the configurable parameters of G: a complete configuration is reached.
5) Check the local synchrony of G with the rates and delays resulting from the complete configuration and compute a schedule (if possible).
6) Fire the non-configuration actors of A following the computed schedule: this corresponds to an iteration of G.
7) Produce on the output ports P^{out}_{data} and P^{out}_{cfg} of a_G the data tokens and parameter values written by the actors of G on the output interfaces I^{out}_{data} and I^{out}_{cfg}.
8) Go back to step 3 to start a new iteration of G, i.e. a new firing of a_G.

These steps can be divided into two groups: steps 1 and 2 which correspond to a configuration phase of G that is not clocked by data but is the result of the virtually instantaneous propagation of parameter values in the parameter dependency tree T; and steps 3 to 8 which correspond to a firing of the actor a_G that is scheduled during the execution of its parent graph.

If the schedulability of the graph could not be verified at compile time (Section IV-A), it will be checked at runtime in

the 5th execution step. If a non-locally synchronous behavior is detected, i.e. if the graph is not consistent or has deadlocks, the execution of the application is terminated.

The runtime verification of the schedulability in step 5 can be used as a debug feature that can be deactivated to improve the performance of the application, thus assuming that a valid schedule can always be found in this step.

As introduced in Sections III-A1 and III-B3, the operational semantics of the πSDF MoC is equivalent to the one of the PSDF presented in [6]. Steps 1 and 2 are equivalent to the execution of the init subgraph ϕ_i, steps 3 to 5 are equivalent to the execution of the subinit subgraph ϕ_s, and steps 6 to 8 are equivalent to the execution of the body subgraph ϕ_b.

V. Comparison with existing MoCs

Table I presents a comparison of dataflow MoCs based on a set of common MoC features. The compared MoCs are the SDF [8], the ADF [11], the IBSDF [7], the PSDF [6], the SADF [1], the DPN [4], and the πSDF. In Table I, a black dot indicates that the feature is implemented by a MoC, an absence of dot means that the feature is not implemented, and an empty dot indicates that the feature may be available for some applications described with this MoC. It is important to note that the full semantics of the compared MoCs is considered here. Indeed, some features can be obtained by using only a restricted semantics of other MoCs. For example, all MoCs can be restricted to describe a SDF, thus benefiting from the static schedulability and the decidability but losing all reconfigurability.

Feature	SDF	ADF	IBSDF	PSDF	πSDF	SADF	DPN
Hierarchy			●	●	●		
Compositional			●		●		
Reconfigurable				●	●	●	●
Configuration dependency					●	●	
Statically schedulable	●	●	●				
Decidability	●	●	●	○	○	●	
Variable rates		●		●	●	●	●
Non-determinism						●	●

TABLE I. Features comparison of different dataflow MoCs

The features compared in Table I are the following: *Hierarchy*: composability can be achieved by associating a subgraph to an actor. *Compositional*: graph properties are independent from the internal specifications of the subgraphs that compose it [13]. *Reconfigurable*: actors firing rules can be reconfigured dynamically. *Configuration dependency*: the MoC semantics includes an element dedicated to the transmission of configuration parameters. *Statically schedulable*: a fully static schedule can be derived at compile time [8]. *Decidability*: the schedulability is provable at compile time. *Variable rates*: production/consumption rates are not a fixed scalar. *Non-determinism*: output of an algorithm does not solely depends on inputs, but also on external factors (e.g. time, randomness).

A. PiMM versus Parameterized Dataflow

PiMM and the *parameterized dataflow* meta-model are equivalent in many points. However, PiMM also introduces new elements of semantics, such as the parameter dependency tree, that enhances the predictability of the model, and hence increases the performance of applications described with the new meta-model.

1) Faster parameter propagation:
In PiMM, the explicit parameter dependency tree enables the instant propagation of parameter values to lower levels of hierarchy through configuration input ports P^{in}_{cfg} and their corresponding configuration input interfaces I^{in}_{cfg}. With this instant propagation, setting a parameter in a hierarchical graph may instantly influence parameters deep in the hierarchy, causing some subgraphs to reach a partial or a complete configuration.

The instant parameter propagation of PiMM is in contrast with the configuration mechanism of the *parameterized dataflow* where the body subgraph ϕ_b of an actor a_G can be configured by parameters set by the init subgraph ϕ_i or the subinit subgraph ϕ_s but cannot directly depend on parameters defined in the parent graph of a_G [6]. This semantics implies that a complete configuration of ϕ_b cannot be reached before the execution of ϕ_i, even if actors in ϕ_i simply propagate parameters from upper levels of hierarchy. Consequently, a complete configuration of a subgraph may be reached earlier for an application modeled with PiMM, providing valuable information to the runtime management system and leading to better scheduling or resource allocation choices, and therefore better performance.

2) Lighter runtime overhead:
In *parameterized dataflow*, the production and consumption rates on the data interfaces of a hierarchical actor are obtained by computing the Repetition Vector (RV) [8] of its subgraph. For dynamically scheduled applications, two computations of the RV are performed at runtime. The first computation is done using a partial configuration completed with default values for undefined parameters. The second computation is done when a complete configuration is reached. The default parameter values used in the first computation must be carefully chosen to ensure that the two RVs present the same production/consumption rates on the interface of the actor, or otherwise the application will be terminated.

In PiMM, the production and consumption rates on the interfaces of a hierarchical actor only depend on the value of locally static parameters. Since neither the first computation of a RV nor the use of default parameter values are needed, the runtime overhead is lighter and the development of the application simpler as the developer does not need to specify default values for configurable parameters.

3) Improved User-friendliness:
In *parameterized dataflow*, the specification of a parameterized hierarchical actor is composed of three subgraphs, which may lead to a rapid increase in the number of graphs to maintain when developing an application. For example, the development of an application with only a dozen parameterized hierarchical actors requires the specification of almost forty graphs.

In PiMM, a single subgraph is needed to specify the behavior of all hierarchical actors, parameterized or not. The employment of a single subgraph is enabled by the introduced parameter dependency tree that replaces the init subgraph ϕ_i and by the configuration actors that replace the subinit subgraphs ϕ_s. Using explicit parameter dependencies also makes it possible to lower the number of actors of a graph, by eliminating the actors whose only purpose was to propagate parameters from the upper levels of the hierarchy.

Moreover, using parameter dependencies instead of referencing parameters by their names makes it easier to identify what is influenced by a parameter. All these features enhance the readability of πSDF graphs, and hence make the model more user-friendly. We have developed a systematic method to transform πSDF specifications into functionality equivalent PSDF specifications. For the sake of brevity this transformation is not presented in this paper and will be the subject of a future publication.

B. *PiMM versus SADF*

In [1], Theleen et al. introduce the Scenario-Aware Dataflow (SADF): a generalization of the SDF model where dynamism is handled by special actors, called detectors, that can reconfigure other actors of their graph by sending them control tokens sequentially through specific FIFOs called control channels. When consumed by an actor, these control tokens change the scenario in which the actor is running, possibly modifying the nature of its computation, its run time, and its production and consumption rates.

A first difference between SADF and πSDF is that in SADF, each actor has a unique status that denotes the current scenario of the actor. Because of this status, an actor cannot be fired multiple times in parallel. In πSDF as in SDF, actors have no state unless explicitly specified with a self-loop FIFO [8]. Consequently, the parallelism embedded in a πSDF description is implicitly greater than the one of an SADF graph.

A second difference between SADF and πSDF lies in the motivations behind the two models. SADF is an analysis-oriented models that has proved to be an efficient model to quickly derive useful metrics such as the worst-case throughput or the long-run average performance [16]. To provide such metrics, SADF relies on a timed description of the actors behavior which corresponds to the execution time of the actor on a specific type of processing elements. Conversely, like PSDF, πSDF is an implementation-oriented, untimed, and architecture-independent model which favors the development of portable applications for heterogeneous MPSoCs. Moreover, it was shown in [16] that implementation of applications described in SADF are less efficient than PSDF applications. Finally, the hierarchical compositionality mechanism of πSDF has no equivalent in SADF.

VI. Application Case: LTE PUSCH

A. *Application Model*

Figure 5 presents a πSDF specification of the bit processing algorithm of the Physical Uplink Shared Channel (PUSCH) decoding which is part of the Long-Term Evolution telecommunication standard (LTE). The LTE PUSCH decoding is executed in the physical layer of an LTE base station (eNodeB). It consists of receiving multiplexed data from several User Equipments (UEs), decoding it and transmitting it to upper layers of the LTE standard.

Because the number of UEs connected to an eNodeB and the rate for each UE can change every millisecond, the bit processing of PUSCH decoding is inherently dynamic and cannot be modeled with static MoCs such as SDF [2].

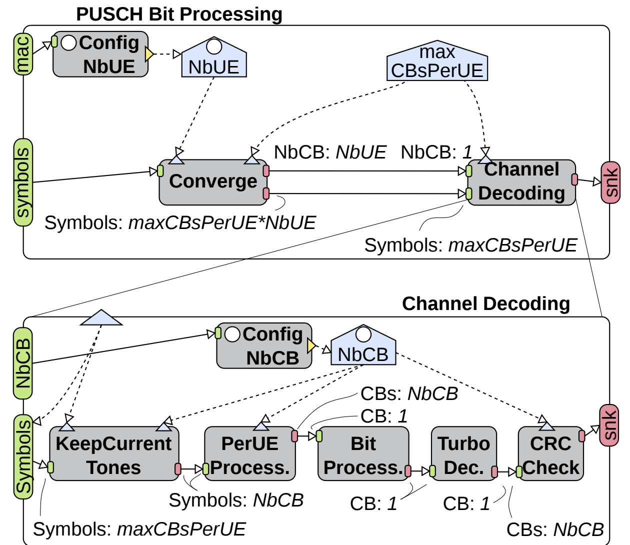

Fig. 5. πSDF Model of the Bit Processing Part of the LTE PUSCH Decoding

The bit processing specification is composed of two hierarchical actors: the *PUSCH Bit Processing* actor and the *Channel Decoding* actor. For clarity, Figure 5 shows a simplified specification of the LTE PUSCH decoding process where some actors and parameters are not depicted.

The *PUSCH Bit Processing* actor is executed once per invocation of the PUSCH decoding process and has a static parameter, *maxCBsPerUE*, that represents the maximum number of data blocks (named Code Block (CB)) per UE. *maxCBsPerUE* statically sets the configuration input interface of the lower level of the hierarchy, according to the eNodeB limitation of bitrate for a single UE. The *ConfigNbUE* configuration actor consumes data tokens coming from the Medium Access Control (MAC) layer and sets the configurable parameter *NbUE* with the number of UEs whose data must be decoded. The *converge* actor consumes the multiplexed CBs received from several antennas on the *symbols* data input interface of the graph, produces *NbUE* tokens, each containing the number of CBs for one UE, and produces *NbUE* packets of *maxCBsPerUE* CBs, each containing the CBs of an UE.

The *Channel Decoding* hierarchical actor fires *NbUE* times, once for each UE, because each UE has specific channel conditions. This actor has a configuration input interface *maxCBsPerUE* that receives the eponymous locally static parameter from the upper hierarchy level. The *ConfigNbCB* configuration actor sets the *NbCB* parameter with the number of CBs allocated for the current UE. An explanation of the role of the remaining actors can be found in [2].

This application illustrates the conciseness of the πSDF model compared to the PSDF model. Indeed, only 2 graphs are needed to specify the application with a πSDF MoC whereas 2 sets of 3 subgraphs (ϕ_i, ϕ_s and ϕ_b) are needed to specify it with PSDF. Figure 6 presents a PSDF model of the same example modeled with PiMM in Figure 5.

B. *Quasi-static schedule*

The use of quasi-static schedules is highly desirable in many contexts compared to dynamic schedules. In particu-

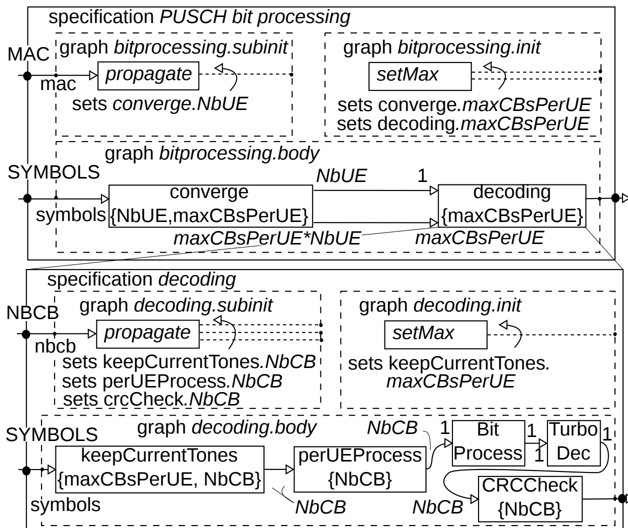

Fig. 6. PSDF Model of the Bit Processing Part of the LTE PUSCH Decoding

lar, quasi-static schedules for *parameterized dataflow* graphs provide significant reductions in runtime overhead, and improvements in predictability (e.g. see [6], [17]). By facilitating systematic construction of parameterized schedules — either manually as part of the design process or automatically as part of a graph transformation flow — our proposed PiMM framework enhances the efficiency and confidence with which dynamically structured signal processing systems can be implemented.

The dynamic topology of a πSDF graph usually prevents the computation of a static schedule since the production/consumption rates are unknown at compile time. In the example of Figure 5, despite the dynamic rates, the production rate on all FIFOs always is a multiple of the consumption rate, or vice versa. Consequently, the dynamic RV is an affine function of the graph parameters and a quasi-static schedule can be computed. Based on an affine formulation, the following quasi-static schedule (Figure 7) can be derived for the graph of Figure 5.

```
while(1){
  /*Execute Top PUSCH*/
  fire ConfigNbUE; //Sets NbUE
  fire Converge;
  repeat NbUE times{
    /*Execute Channel Decoding*/
    fire ConfigNbCB; //Sets NbCB
    fire KeepCurrentTones;
    fire PerUEProcess;
    repeat NbCB times{
      fire BitProcess;
      fire TurboDec;
    }
    fire CrcCheck;
  }
}
```

Fig. 7. Quasi-static schedule for graph in Figure 5

VII. Conclusions

This paper introduces a meta-model of computation called PiMM that can be applied to a dataflow MoC to increase its expressivity, enable the specification of reconfigurable applications, and promote derivation of quasi-static schedules. We have shown that while bringing dynamism and compositionality, the explicit parameter dependency tree and the interface-based hierarchy mechanism introduced by PiMM maintain strong predictability for the extended model and enforce the conciseness and readability of application descriptions. Useful directions for future work include the application of PiMM to dataflow MoCs other than SDF, further compile-time analysis of PiMM specifications, and exploration of optimized runtime management systems for PiMM. The objective of such runtime management includes exploiting the predictability offered by PiMM specifications to perform efficient application scheduling and resource allocation, and incorporate results of relevant compile-time analysis of the model.

References

[1] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, and S. Stuijk, "A scenario-aware dataflow model for combined long-run average and worst-case performance analysis," in *MEMOCODE*, 2006.

[2] M. Pelcat, S. Aridhi, J. Piat, and J.-F. Nezan, *Physical Layer Multi-Core Prototyping: A Dataflow-Based Approach for LTE eNodeB*. Springer, 2012.

[3] H. Nikolov, T. Stefanov, and E. Deprettere, "Modeling and FPGA implementation of applications using parameterized process networks with non-static parameters," in *FCCM Proceedings*, 2005.

[4] E. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE*, 1995.

[5] J. Eker and J. Janneck, "CAL Language Report," University of California at Berkeley, Tech. Rep. ERL Technical Memo UCB/ERL M03/48, Dec. 2003.

[6] B. Bhattacharya and S. Bhattacharyya, "Parameterized dataflow modeling for dsp systems," *Signal Processing, IEEE Transactions on*, 2001.

[7] J. Piat, S. Bhattacharyya, and M. Raulet, "Interface-based hierarchy for synchronous data-flow graphs," in *SiPS Proceedings*, 2009.

[8] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, 1987.

[9] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *Signal Processing, IEEE Transactions on*, vol. 44, no. 2, pp. 397 –408, feb 1996.

[10] P. Murthy and E. Lee, "Multidimensional synchronous dataflow," *Signal Processing, IEEE Transactions on*, vol. 50, no. 8, pp. 2064 –2079, aug 2002.

[11] A. Bouakaz, J.-P. Talpin, and J. Vitek, "Affine data-flow graphs for the synthesis of hard real-time applications," *ACSD*, 2012.

[12] J. P. Hausmans, S. J. Geuns, M. H. Wiggers, and M. J. Bekooij, "Compositional temporal analysis model for incremental hard real-time system design," in *EMSOFT '12 Proceedings*, 2012.

[13] J. Ostroff, "Abstraction and composition of discrete real-time systems," *Proc. of CASE*, vol. 95, pp. 370–380, 1995.

[14] S. Neuendorffer and E. Lee, "Hierarchical reconfiguration of dataflow models," in *MEMOCODE*, 2004.

[15] H. Kee, C. C. Shen, S. S. Bhattacharyya, I. Wong, Y. Rao, and J. Kornerup, "Mapping parameterized cyclo-static dataflow graphs onto configurable hardware," *Journal of Signal Processing Systems*, 2012.

[16] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, 2011.

[17] J. Boutellier, "Quasi-static scheduling for fine-grained embedded multiprocessing." Ph.D. dissertation, University of Oulu, 2009.