Information and Communication Technologies (ICT) Programme

Project N°: H2020-ICT-2016-1-732105

D3.2: Models of Computation

Lead Beneficiary: INSA
Workpackage: WP3
Date: 30/06/2019
Distribution - Confidentiality: Public

Abstract:

This document presents the innovations on Models of Computations (MoCs) for the design of Cyber-Physical Systems (CPSs) resulting from the work of the CERBERO partners. Motivations behind the newly introduced MoC features are presented as well as how these features will ease the adoption and efficiency of model-based methods in the CPS design context.

Disclaimer

The CERBERO Consortium is the following:

Num.	Beneficiary name	Acronym	Country
1 (Coord.)	IBM Israel – Science and Technology LTD	IBM	IL
2	Università degli Studi di Sassari	UniSS	IT
3	Thales Alenia Space Espana, SA	TASE	ES
4	Università degli Studi di Cagliari	UniCA	IT
5	Institut National des Sciences Appliquees de Rennes	INSA	FR
6	Universidad Politecnica de Madrid	UPM	ES
7	Università della Svizzera Italiana	USI	CH
8	Abinsula SRL	AI	IT
9	Ambiesense LTD	AS	UK
10	Nederlandse Organisatie Voor Toegepast Natuurwetenschappelijk Ondeerzoek TNO	TNO	NL
11	Science and Technology	S&T	NL
12	Centro Ricerche FIAT	CRF	IT

For the CERBERO Consortium, please see the http://cerbero-h2020.eu web-site.

Document Authors

The following list of authors reflects the major contribution to the writing of the document.

Name(s)	Organization Acronym
Karol Desnos	INSA
Florian Arrestier	INSA
Alexandre Honorat	INSA
Jean-François Nezan	INSA
Daniel Menard	INSA
Maxime Pelcat	INSA
Eduardo Juarez	UPM
Raquel Lazcano	UPM
Daniel Madroñal	UPM
Francesca Palumbo	UniSS
Tiziana Fanni	UniCA

Document Revision History

Date	Ver.	Contributor (Beneficiary)	Summary of main changes
2019.04.04	0.1	Karol Desnos (INSA)	Table of content draft
2019.06.04	0.2	Karol Desnos (INSA) Eduardo Juarez (UPM) Raquel Lazcano (UPM) Daniel Madroñal (UPM)	Filling all sections
2019.06.06	0.3	Raquel Lazcano (UPM)	Update in Section 5.4
2019.06.17	0.4	Tiziana Fanni (UniCA) Francesca Palumbo (UniSS) Karol Desnos (INSA) Eduardo Juarez (UPM)	Document internal review

Table of contents

1. Executive Summary

This document relates the activities and contributions of the CERBERO consortium related to the use of Models of Computation (MoCs) for the design of Cyber-Physical Systems (CPS). A survey of state-of-the-art MoCs was previously proposed in D3.5, outlining the main characteristics of MoCs used for the design of CPSs by presenting:

- the properties of their semantics (analyzability, decidability, reconfigurability, expressiveness, determinism, ...),
- the kind of algorithm it supports (data-driven, control-driven, …),
- the level of abstraction it captures (system-of-systems, system, component, …)
- the type of implementation it translates into (hardware, software, distributed, …).

Based on this survey, a mapping of the different MoCs supported in the different tools and layers of the CERBERO framework was presented. The lacks in the current semantics of the used MoCs were identified in D3.5, and a set of new MoC features needed to better support the design of CERBERO use-cases was defined. This document presents the result of the work on these new MoC features and how they will allow these MoCs to be more generally and effectively adopted in the CPS context.

This document is an update of former deliverable D3.5. In order to speed up and ease the reading and review process, the text of sections and paragraphs that have NOT been significantly updated and revised are in dark gray. For conciseness, only a brief summary of section 4 from D3.5 is given. Sections that have been deeply updated and revised are most sub-sections within section 5.

1.1. Structure of Document

Section 2 of this document reminds the definition of the notions of abstraction and models, which serve as a basis to the concept of Models of Computation. Section 0 reminds a set of properties of MoCs that are used in Section 0 to characterize and compare the state-of-the-art MoCs commonly used for the design of CPSs. Finally, Section 5 presents the CERBERO innovations in the domain of MoCs for the modeling of CPS.

1.2. Related Documents

- D2.7 - CERBERO Technical Requirements
  - D3.2 contributes to satisfy D2.7 requirements. Details are given in Section □.
- D3.3 - Cross-layer Modelling Methodology for CPS
  - The models of computation described in this document are used to represent one aspect of the CPS, the behavior. This is a key foundation in the cross-layer modeling methodology.
- D3.5 - Models of Computation
  - First version of this deliverable with complete MoC survey, and planned innovations on MoCs.
- D5.2 - CERBERO Framework Components

  - D5.2 gives more technical details on the support of MoCs within the tools that are components of the CERBERO framework.
- D5.5 - CERBERO Framework Components
  - Uses of MoCs as a mean of interconnection between pairs of tools, or within the CERBERO Interoperability Framework (CIF) are detailed in D5.5.

1.3. Related CERBERO Requirements

Deliverable D2.7 of the CERBERO project defines a list of CERBERO Technical Requirements (CTRs) the project should achieve. Each of them is referenced with a unique identifier ranging from 0001 to 0020. MoC exploration and innovation are carried out following the requirements in Table 1. This table was updated since D3.5 to link the different CTRs to the sections of the document describing contributions to address them.

CTR id	**CTR Description**	**Link with the D3.2 document on *Models of Computation***	**Sections describing the contributions**
0001	CERBERO framework SHOULD increase the level of abstraction at least by one for HW/SW co-design and for System Level Design.	Innovations on MoCs help raising the abstraction level for the designer	Section 5.1 introduces new dataflow semantics to ease the design of iterative computations at a dataflow level. Secton 5.2 introduces new semantics for modeling real-time periodic constraints at dataflow level.
0002	CERBERO framework SHOULD provide interoperability between cross-layer tools and semantics at the same level of abstraction.	Formalization of MoCs and homogeneity among partners foster tool interoperability	Sections 5.1, 5.2, 5.3, and 5.4 all introduce contributions that either extend or rely on the dataflow MoC formalism. Formal specification of the MoC foster integration between tools inside (PREESM, SPIDER, PAPIFY, AOW, DYNAA, ARTICO^3, MDC) and outside (APPOLO) of the CERBERO Framework
0007	CERBERO framework SHALL define methodology and SHOULD provide library of reusable functional and non-functional KPIs.	Non-functional KPIs can be influenced in the MoCs using proposed Moldable Parameters	Sections 5.2 and 5.3 introduce new non-functional concepts, periodic actors and moldable parameters, respectively; on top of the existing dataflow semantics.
0020	CERBERO framework SHALL provide methodology and tools for development of adaptive applications.	Proposed innovations on MoCs improve the expressiveness and specify the semantic of PiSDF for designing adaptive applications	Sections 5.1, 5.2 and 5.3 introduce new element of semantics that, by improving the expressiveness of the PiSDF MoCs, ease the design of adaptive applications. The contribution introduces in section 5.5 is a model analysis technique that fosters the usage of dataflow MoCs for fast runtime resource allocation of resources for adaptive applications.

Table 1: Links to CERBERO Technical Requirement

2. Models of computation

This first section briefly defines the core concepts of abstraction, model, and model of computations.

2.1. Abstraction

In general, abstraction is a tradeoff between the level of details and the complexity adopted when describing or representing a thing (e.g., an idea, a system, a place, an object, a phenomenon, etc.). Two distinct representations used to describe the same thing, each adopting a different abstraction tradeoff (i.e., amount of details conveyed about it), can be compared relatively to each other using so-called levels of abstraction.

- The lower level of abstraction gives a representation of the thing which is more detailed, thus giving a more precise and complete description.
- The higher level of abstraction gives a representation of the thing where some details are voluntarily omitted to decrease the complexity of the description. This higher complexity generally translates into a smaller and/or less dense representation of the thing.

2.2. Models

A model is a mathematically grounded representation capturing predictable characteristics of a system. More precisely, a model consists of a set of elements that can be assembled respecting a set of rules to describe a system. For a valid representation built with a model, mathematical equations associated to the elements of the model make it possible to predict some characteristics of the modeled system. Models are commonly used in all scientific fields to represent evolution of physical, computing, chemical, financial, or social systems.

For example, the symbol in Figure 1 – Bipolar Transistor Symbol and its associated equation in Figure 2 - Bipolar Transistor Equation are used to model and predict the voltage and current characteristics of a transistor within a model of an analog circuit.

Figure 1 – Bipolar Transistor Symbol

$$I_c = \beta I_s \left[1 + \frac{V_{ce}}{V_{EA}}\right] \exp\left(\frac{V_{be}}{V_{th}}\right)$$

Figure 2 - Bipolar Transistor Equation

In the context of cyber-physical systems (CPSs) engineering, several models adopting different levels of abstraction can be used to describe separated or nested aspects of a system. More details on the use of heterogeneous models to describe a CPS are presented in D3.3.

2.3. Models of Computation

A Model of Computation (MoC) is a set of operational elements that can be composed to describe the behavior of an application. The set of operational elements of a MoC and the set of relations that can be used to link these elements are called the semantics of a MoC.

As presented in [Savage 1998], MoCs can be seen as an interface between the computer science and the mathematical domains. A MoC specifies a set of rules that control how systems described with the MoC are executed. Each element of the semantics of a MoC can be associated to a set of properties, such as timing properties or resource requirements. These rules and properties provide the theoretical framework that can be used to formally analyze the characteristics of applications described with a MoC. For example, using a mathematical analysis, it may be possible to prove that an application described with a given MoC will never get stuck in an unwanted state or that it will always run in a bounded execution time. Section 0 of this document describes a set of properties that are commonly supported by existing MoCs, which are themselves described in Section 0. A more extensive introduction to CPS modelling with MoCs can be found in [Lee 2017].

3. Characterization of Models of Computation

Since the introduction of modern computing systems in mid-1900s, a plethora of MoCs have been proposed by the scientific community. Very often, a new MoC is introduced to allow the specification of applications or systems that exhibit a set of characteristics whose specification was impossible or difficult to achieve with previously existing MoCs.

When designing a system, it is important to identify its required and desired properties. Once these have been identified, the designer can select the MoC whose semantics will make it easier to express, verify and guarantee those properties by construction.

The objective of this section is to give a definition of the properties used to characterize and compare the MoCs presented in Section 0.

3.1. Properties

This section lists a set of commonly used properties utilized to compare the system characteristics supported by different MoCs.

Analyzability

The analyzability of a MoC evaluates the availability of analysis and synthesis algorithms that can be used to characterize applications modeled with this MoC. For example, in the synchronous dataflow MoC, analysis algorithms can be applied at compile-time to compute the worst-case latency or the maximum memory requirements of a design.

Conciseness

The conciseness (or succinctness) of a MoC captures its ability to express complex system behaviors with a limited description size. This relative property is useful for comparing MoCs with equivalent expressiveness. Indeed, conciseness is often a desired feature for system developers as the design of an identical application with two MoCs (of identical expressiveness) will lead to a smaller design with the more concise MoC.

Compositionality

A modular MoC is compositional if the analyzable properties of a module described with this MoC are independent from the internal specification of the submodules that compose it [Ostroff 1995]. For example, in a compositional MoC, if each submodule used in the design is (independently) deadlock free, then the whole design combining these submodules will be deadlock-free by construction.

Decidability

A MoC is decidable if the schedulability of applications described with this model can be proved statically (i.e. at compile time) [Bhattacharyya 2006]. Hence, using a decidable MoC makes it possible to guarantee at compile-time that a system will never reach a deadlock state and that its execution will require a finite amount of memory. A non-decidable MoC does not mean that applications will not be schedulable, only that their schedulability can only be verified “on the fly” at runtime. Decidability is often obtained as a trade-off for a limited expressiveness of the MoC.

Determinism

A MoC is deterministic if the output of an algorithm only depends on its inputs, but not on external factors such as time or randomness. If determinism is a desired feature for most control and streaming applications, non-determinism may also be needed to describe applications reacting to unpredictable inputs.

Expressiveness

The expressiveness, of a MoC evaluates the complexity of application behaviors that can be described with this MoC. For example, the expressivity of the Dataflow Process Network (DPN) MoC has been proven to be equivalent to a Turing machine. The specialization of a MoC restricts the expressivity of this MoC to increase its analyzability, or to give it new properties such as determinism or decidability. Expressivity is often mistaken for conciseness. For example, the Cyclo-Static Dataflow (CSDF) MoC is often said to be more expressive than the Synchronous Dataflow MoC but meaning instead that it has a better conciseness.

Modularity

In a modular (or hierarchical) MoC, a system description can be broken into several independent modules. The modules that are combined to create a system can be (re-)used either in different systems specification or instantiated several times in the same. The modules themselves can be described using the same MoC as the top-level system description or can encapsulate other compatible MoCs.

Parallelism

In a parallel MoC, several independent elements of a system description may “activate” concurrently and independently from each other, each causing a change in the current state of the system. In a sequential (i.e. non-parallel MoC), all changes of the system state can be broken down to a sequence of actions triggered one after another, according to the system semantics.

Reconfigurability

A MoC is reconfigurable if the behavior of entire parts of a system description can be modified dynamically, to fulfill future execution goals for a foreseeable amount of time. Reconfiguration is used to dynamically adapt the behavior of a system to its environment, notably by enabling or disabling parts of the system, by modifying its functional behavior (e.g. its computations, QoS, …), or by modifying its non-functional properties (e.g. exposed parallelism, energy consumption, …).

Predictability

The predictability property is related to the reconfigurability property of a MoC. This property evaluates the amount of time between a reconfiguration of a part of the system, and the beginning of activity in the reconfigured part. The more predictable a MoC is, the more the time that can be used by a runtime manager to react and perform an optimization of the reconfigured part before using it.

3.2. Additional MoC Comparison Criteria

This section introduces a few other criteria that can be used to compare MoCs. These comparison criteria denote different classes of applications that a MoC can be used to represent. Unlike the properties presented in the Section 3.1, which capture properties supported (or not) by MoCs, this section introduces more subjective comparison criteria. Indeed, if some MoCs seem more suitable to implement a given class of applications, using them to implement another class may still be possible, but less practical or less common.

Algorithms Computation Classification

Algorithms described with a MoC can be classified into several classes depending on the type of involved computation:

- **Stream-based**: A continuous stream of data is steadily processed and produced by the described algorithm. The amount and nature of the computation do not vary depending on the data.
- **Data-driven:** The amount and nature of the computation do not vary depending on the data. Contrary to stream-based algorithms, data does not necessarily arrive continuously.
- **Control Driven**: The amount and nature of the computation depend on the processed data.
- **Event Driven**: Computations are triggered by events on the frontier of the system (i.e. by sensors, users, communication network, …).

Captured Algorithms Granularities

MoCs with different levels of abstractions are inherently suitable for representing behaviors of diverse granularities:

- **Function:** The modeled algorithm captures computations that are building blocks used to assemble an algorithm with a higher granularity.
- **Component:** The modeled algorithm serves a well-specified purpose with clear input and output interfaces and constraints.
- **System:** The modeled algorithm represents a collection of components with diverse objectives but running locally on a unique computing system.
- **System-of-systems:** The modeled algorithm consists of several independent "systems", each *existing and evolving* independently from the others but exchanging information among them through communication channels.

Implementation Types

A MoC is a theoretical representation used to describe the behavior of an application. Implementing a MoC consists in translating this theoretical behavior into an "executable" description. Different types of implementations can be more or less suitable to implement each MoC:

- **Hardware**: Algorithms described with this type of MoC can be efficiently translated into logical gates, signals, and registers on an ASIC or an FPGA.
- **Software**: Algorithms described with this type of MoC can be efficiently translated into a sequence of instructions executed on a processor that manipulates data stored in a memory space.

- **Distributed:** Algorithms described with this type of MoC can be efficiently implemented by splitting them into several parts executed on separate Hardware or Software components, each storing a part of the system state and executing a part of the computations in parallel.
- **Heterogeneous:** Algorithms described with this type of MoC can be efficiently translated into a mix of hardware and software implementations.

4. Surveyed Models of Computation

Using key characteristics of MoCs defined in Section 3, Table **4:** ***SDF***: Synchronous Dataflow; ***PiSDF***: Parameterized and Interfaced Synchronous Dataflow; ***BSP***: Bulk Synchronous Parallel; ***PN***: Petri Networks; ***DPN***: Dataflow Process Network; ***RTL***: Register Transfer Level; ***TS***: Transition System; ***KPN***: Kahn Process Networks; ***DES***: Discrete Event System; ***sCE***: Situated Cognitive Engineering**.** Non-bold MoCs are those inherited from their parent (first bold one above). Blank cells indicate a MoC has no specific traits for the property. can be used to quickly compare the characteristics of state-of-the-art MoCs.

	Analyzability	Conciseness	Compositionality	Decidability	Determinism	Expressiveness	Modular	Parallelism	Predictability	Reconfigurability
RTL	+		++		+		++	++	+	
SDF	++	+		+	+	-	--	+		--
Cyclo-Static	++	++								
Interface-Based SDF	++	+	+							
Deterministic SDFwith Shared FIFO	++	+								
PiSDF	++	++	+	+	+	++	+	+		+
BSP		-	+	+		-	+	++		-
KPN			+	-	+			+		
DPN	-			-	+	++		-		
PN	+	+			-	++	+	+		+
DES	-		++	-		+++	++	++	-	+
SCE					+	+	+	+		+
TS				+	+					

Table 4: ***SDF***: Synchronous Dataflow; ***PiSDF***: Parameterized and Interfaced Synchronous Dataflow; ***BSP***: Bulk Synchronous Parallel; ***PN***: Petri Networks; ***DPN***: Dataflow Process Network; ***RTL***: Register Transfer Level; ***TS***: Transition System; ***KPN***: Kahn Process Networks; ***DES***: Discrete Event System; ***sCE***: Situated Cognitive Engineering**.** Non-bold MoCs are those inherited from their parent (first bold one above). Blank cells indicate a MoC has no specific traits for the property.

A thorough presentation of all MoCs listed in **Table 4:** ***SDF***: Synchronous Dataflow; ***PiSDF***: Parameterized and Interfaced Synchronous Dataflow; ***BSP***: Bulk Synchronous Parallel; ***PN***: Petri Networks; ***DPN***: Dataflow Process Network; ***RTL***: Register Transfer Level; ***TS***: Transition System; ***KPN***: Kahn Process Networks; ***DES***: Discrete Event

System; ***sCE***: Situated Cognitive Engineering**.** Non-bold MoCs are those inherited from their parent (first bold one above). Blank cells indicate a MoC has no specific traits for the property. was given in D3.5. For the sake of conciseness, only the definition of MoCs studied in section 5 are copied hereafter.

4.1. Synchronous Dataflow

MoC brief description

The Synchronous Dataflow [Lee 1987] MoC models an application as a directed graph of computational entities, called actors, that exchange data through a network of First-In First-Out queues (FIFOs). Each time an actor is executed, or fired, it consumes and produces a fixed quantum of data, called data token, on the FIFOs to which it is connected. An example of SDF graph is given in Figure 3 - Example of Synchronous Dataflow Graph.

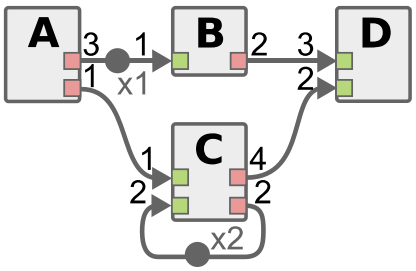

Figure 3 - Example of Synchronous Dataflow Graph

MoC properties

Synchronous Dataflow is a *parallel* and *decidable* MoC that exhibits one of the greatest degrees of *analyzability* among dataflow MoCs. Coupled with the *determinism* of the MoC, its *analyzability* makes it possible to prove algorithms deadlock freeness and boundedness at compile time and is often used to guarantee real-time properties (e.g. throughput, latency, worst-case execution time) of applications modeled with it. This great analyzability comes at the expense of a limited *expressiveness* of the MoC, because of the absence of any *reconfiguration* semantics in the MoC. The original MoC described in [Lee 1987] is *not modular*.

Relationship with other MoCs

The SDF MoC belongs to the family of dataflow models of computation. As one of the dataflow MoCs with the most restrictive semantics, SDF behavior can be expressed in most dataflow models.

As demonstrated in [Klikpo 2016], the MoC implemented in Labview® is equivalent to the SDF MoC.

There exist several dataflow MoCs with an equivalent expressiveness with the SDF MoC:

- The Cyclo-Static Dataflow [Bilsen 1996] and Affine Dataflow [Bouakaz 2012] MoCs which have a greater *conciseness* than the SDF MoC while retaining all its *analyzability*, by specifying sequences of production and consumption rates instead of scalar values.

- The Interface-Based SDF [Piat 2009] and Deterministic SDF with Shared FIFO [Tripakis 2013] MoCs which are two *modular* and *compositional* extensions of the SDF MoC.

MoC Usage

Synchronous Dataflow is mainly used to describe *stream-based* and *data-driven* algorithms, mostly at *function* and *component* levels. The SDF MoC is suitable for all kinds of implementations.

MoC Support

The SDF MoC is natively supported in the following tools: Ptolemy II [Davis 1999], SDF3 [Stuijk 2006], PREESM [Pelcat 2014], MDC [Palumbo 2017], LIDE [Shen 2011].

4.2. Parameterized and Interfaced Synchronous Dataflow

MoC brief description

The Parameterized and Interfaced Synchronous Dataflow (PiSDF) is the result of applying the Parameterized and Interfaced dataflow Meta-Modeling methodology [Desnos 2013] to the SDF MoC. PiSDF adds parameterization and interfaced hierarchy to the SDF MoC. The PiSDF MoC models an application as a directed graph. Besides actors and FIFOs (see section 4.1), parameters, hierarchical interfaces and parameter dependencies can also be vertices of the graph.

Parameters are employed to configure and modify dataflow specifications. Parameters can influence (1) the functionality of an actor, (2) the production/consumption rates of actor ports, (3) the value of another parameter and (4) a delay of a FIFO. Hierarchical interfaces convey data tokens or parameter values between levels of hierarchy. Hierarchical interfaced actors, or simply, hierarchical actors, are univocally linked to PiSDF subgraphs. Parameter dependencies propagate parameter values to other elements of the graph.

Actors, hierarchical or non-hierarchical, can have two types of ports: data ports and configuration ports. Data ports exchange data and configuration ports parameters. Parameters are connected to configuration ports through parameter dependencies. Both types of ports can be declared as input or output ports. An actor with an output configuration port is named a configuration actor. Firing of configuration actors dynamically produces values that set configurable parameters. The firing is only permitted at specific instants of time during a graph execution.

There are two types of parameters in a PiSDF MoC: configurable parameters and locally static parameters. Configurable parameters can be modified in each graph iteration, i.e. at run-time. Locally static parameters can only be modified at design-time. Parameter values passed through input configuration interfaces of hierarchical actors always become locally static parameters of hierarchical (sub)graphs.

Output configuration ports are always connected to configurable parameters. A change in a configurable parameter is the result of a change in either an output configuration port of an actor or another configurable parameter the former depends upon.

MoC properties

PiSDF inherits the properties of SDF (see section 4.1) and adds the *modularity* and *reconfigurability* properties, with the advantage of keeping the *analyzability* of SDF. As the reconfiguration semantics is included into PiSDF, its *expressiveness* is greater than that of SDF. Besides modularity, reconfigurability is extremely handy in the context of cyber-physical systems, which is why in the CERBERO project we intend to use and extend PiSDF (see Section 5).

Relationship with other MoCs

The PiSDF MoC is related to the Interface-Based SDF [Piat 2009], from which it inherits the *compositional* hierarchy mechanism. The PiSDF MoC has the same *expressiveness*, but a better *conciseness*, as the Parameterized SDF MoC [Bhattacharya 2001].

MoC Usage

PiSDF is mainly used to describe stream-based, data-driven and control-driven algorithms (with a reduced number of configurable parameters in practice), mostly at functional and component levels. The PiSDF MoC is suitable for implementations in heterogeneous systems [Heulot 2014].

MoC Support

The SDF MoC is natively supported in the tool PREESM [Pelcat 2014], and the SPiDER runtime [Heulot 2014] is used to support the reconfiguration of graphs during execution. The tools MDC [Palumbo 2017] and ARTICo³ will support this MoC and integrate with PREESM and SPiDER. The objective is to offer new scheduling and mapping choices to the runtime manager when dealing with reconfigurable hardware, i.e. hardware and software implementations for an actor. The decisions will be driven by on-the-fly readings of performance indicators using the Performance API (PAPI).

5. CERBERO Innovation on Models of Computation for CPS

This section presents the contributions of the CERBERO project to the Model of Computation domain. The main motivation behind these contributions is to support the specification of key aspects of CPSs. In particular the proposed contributions aim at:

- Extending the expressiveness of the dataflow MoCs to better capture iterative computations over semi-persistent data;
- Extending the semantics and analyzability of dataflow MoCs for real-time design concerns;
- Extending the semantics of dataflow MoCs to increase optimization opportunities during design space exploration phases;
- Combining dataflow MoCs and polyhedral models and transformation for the optimization of embedded software;
- Proposing a new numerical analysis technique to ease the efficiency of runtime resource allocation.

5.1. Dataflow Extension for Persistent State Representation

Summary of the work published in [Arrestier 2018]

Motivations & Problematic

In synchronous dataflow MoCs, as for example in the Synchronous Dataflow (SDF) and Parameterized and Interfaced SDF (PiSDF) models presented in Section 4, the semantics is dedicated to the processing of infinite streams of data. To this purpose, the semantics of these dataflow MoCs has been tailored to capture in a concise form the data-parallelism and determinism of algorithms executed infinitely repeatedly, with numerous and entangled data dependencies.

Despite the many advantages of the semantics of dataflow models, these cannot currently be used to represent concisely and unambiguously the persistence or the sporadic initialization of data within algorithms. In CPSs, where computing systems must continuously adapt their behavior to the physical environment enclosing them, these persistent data are needed to capture the adaptive state of algorithms. These persistent coefficients of such an adaptive system may be sporadically updated to fit an evolution of their working environment. An example of such persistent data is the coefficients encoding the learning ability of the neurons in online machine learning algorithms.

In the 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 at the beginning of any graph iteration, available prior to any actor firing. As specified in [Lee 1987], these initial data tokens of a FIFO, called delays, can be used to "transmit" persistent data between successive iteration of a graph.

Although the concept of delays exists in most dataflow MoCs, the initial values given to the corresponding data tokens are hardly mentioned, let alone specified, in the literature. In the few publications where they are specified, initial values are set to 0 [Lee 1987, Sriram 2009]. 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 proposed semantics of delay.

The persistence of the data tokens of delays across levels of hierarchy and across graph iterations also differs between MoCs. In a non-hierarchical model, like the SDF MoC, the last data tokens produced during an iteration *n*, on a delayed FIFO, are used as the initial conditions of iteration *n+1*. However, in hierarchical MoCs, delays can appear in subgraphs used for specifying the internal behavior of a hierarchical actor. Contrary to flat MoCs where delayed data tokens generally persist across graph iterations, the persistence scope of delayed data tokens in hierarchical subgraphs, possibly across multiple firing of their parent actors, is unspecified behavior. The CERBERO contribution introduces a clear semantics to control the persistence scope of delays in hierarchical and reconfigurable dataflow MoCs.

Contribution: State-Aware Dataflow

The State-Aware Dataflow (SAD) meta-model was proposed in CERBERO to disambiguate the specification and use of persistent state within hierarchical and reconfigurable dataflow MoCs implementing a well-defined notion of graph iteration. The SAD meta-model comprises a set of semantic elements, that can be used for extending the semantics of an existing dataflow MoC to add both explicit initialization of delays and hierarchical state awareness through the use of a customizable persistence scope of delays.

Initialization semantics: Delays are usually represented by a filled circle positioned on a FIFO as displayed in Figure 3. Figure 4 introduces the graphical representation of the proposed semantics for delays with the additional data connections for initialization purposes. Actors *P* and *C* are the production and the consumption actors, respectively, of the FIFO *f* containing a delay. Actor *S* is the setter actor of the delays on FIFO *f;* delays to which they are each connected with a FIFO, that is, one FIFO from actor *S* to the delay. Symmetrically, actor *G* is the getter actor of the delays on FIFO *f*, and a FIFO connects these delays to actor *G*. 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.

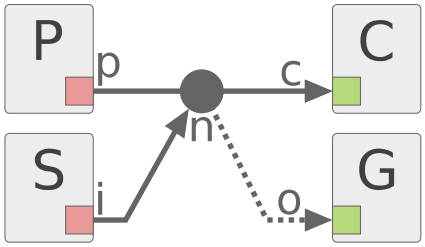

Figure 4 - Delay Initialization Semantics

From the behavioral semantics point of view, the new data connections between the setter and getter actors and the delay induce the following precedence rules in the firing sequence of actors during each graph iteration:

- All firings of the setter actor of a delay must occur prior to the first firing of the consumption actor of this delay.
- All firings of the getter actor of a delay must occur after the last firing of the production actor of this delay.

From the functional point of view: the setter actor of the proposed semantics is responsible for giving its initial value to the delayed token, before their consumption by the consumer actor *C* within each iteration of the graph. In the absence of a setter actor *S* for a given delay, the default initialization of the proposed semantics is to set all data tokens of the delay to zero. Symmetrically, the getter actor *G* can retrieve the final value of the delay, for further processing, after all executions of the producer actor *P* within each iteration of the graph. This construction makes it possible to easily specify iterative computations, similar to for-loops, in the dataflow MoC.

Explicitly initializing the delays means that new initialization tokens are produced on each graph iteration. Thus, if no getter actor is connected to the output connection of a delay, the produced data tokens have to be discarded to ensure bounded memory execution.

While improving its **expressiveness**, the proposed delay semantics of SAD preserves the **analyzability**, the **dependability**, and the **determinism** of the extended dataflow models. In particular, methods provided in [Arrestier 2018] can be used for checking the consistency, schedulability and liveness properties of an application specified using this delay initialization semantics.

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*. A new unambiguous semantics for modeling this persistence of data tokens across graph iterations is presented below.

Delay persistence semantics: The persistence of delays defines whether tokens of a delayed FIFO should be discarded or preserved for the next graph iteration, or for the next firing of the parent hierarchical actor. Persistent delays contained in a subgraph of a hierarchical actor are also called the "state" of this actor.

To control the persistence scope of delays in a hierarchical graph, SAD introduces 3 different types of delays illustrated in Figure 5: Local Delays, Locally Persistent Delays, and Globally Persistent Delays.

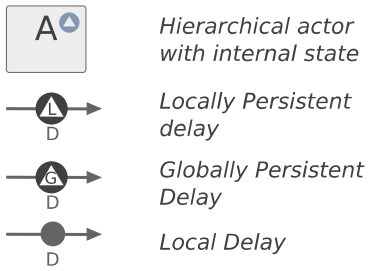

Figure 5- SAD Persistence Semantics

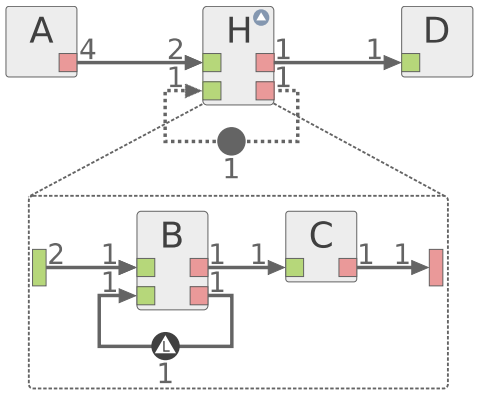

Figure 6 - Example of PiSDF graph with Locally Persistent Delay

Local Delays (LDs) use the initialization semantics presented in the previous paragraph. 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 but do not persist beyond.

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 the hierarchy and establishes a precedence relationship for successive firings 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 the 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 [Lee 1995]. By default, any LPD in the top-level of the hierarchy is a GPD.

In the graph of Figure 6, the delay inside the hierarchical actor *H* is defined as an LPD. The persistence of this LPD is made explicit with a feedback loop around the parent hierarchical actor *H*. Note that using an LPD induces a data precedence relationship between firings of the parent actors, which forces the scheduler of the graph to serialize the firings of actor *H*. In the example of Figure 6, with the LPD, consecutive firings of actor *H* shall be scheduled and executed one after the other. In this example, replacing the LPD within the subgraph of actor *H* with an LD would break this serialization constraint, and would make it possible to execute multiple firings of *H* in parallel.

The customizable persistence scope for delays offered by the SAD meta-model leads to controlled data parallelism in hierarchical graphs which can be taken into account during the analysis and scheduling of the graphs [Arrestier 2018].

Use in CERBERO

This extension of the dataflow model is suitable for the modeling of any CPS system and has been implemented within both PREESM & SPiDER. The use of this contribution has been demonstrated in [Arrestier 2018] with an implementation of a non-trivial reinforcement learning algorithm that is applicable to any control system, such as the robotic arm of the Space Exploration Use-Case.

The reinforcement learning algorithm used to showcase the proposed contribution is called the Continuous Actor Critic Learning Automaton (CACLA) algorithm [Van Hasselt 2007]. This application example is selected to demonstrate the conciseness and memory efficiency of the SAD meta-model application to the PiSDF MoC.

The top-level graph of the CACLA algorithm is depicted in Figure 7 and the subgraph of the *Update* actors, which contain both an LD and an LPD, is depicted in Figure 8.

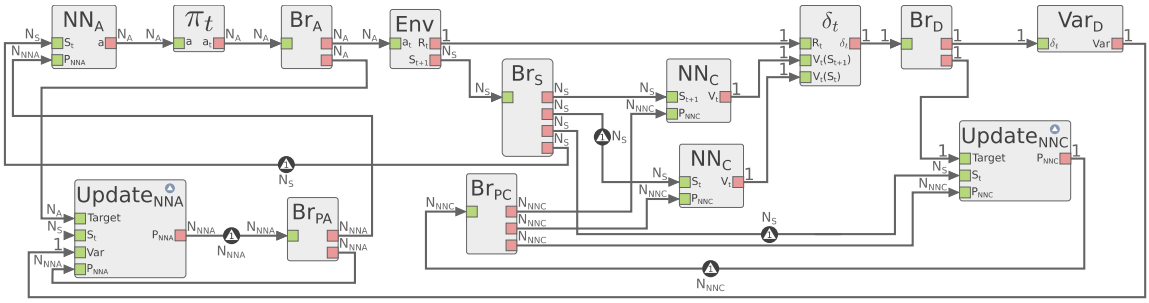

Figure 7 - Top-level PiSDF graph of the CACLA Algorithm

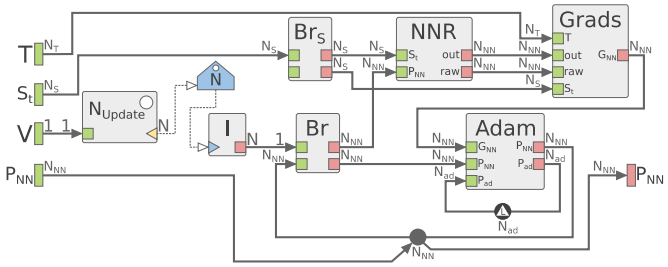

Figure 8 - PiSDF Subgraph of the Update Actor of the CACLA Algorithm

In order to assess the benefits from using the newly introduced semantics for delays initialization and persistence, the proposed implementation was compared with an equivalent implementation of the same application with the PiSDF model, without using the newly introduced semantics. Results presented in [Arrestier 2018] show that the proposed approach makes it possible to reduce the amount of memory needed to run the whole application by 35%. This memory footprint reduction is obtained because in the absence of the SAD semantics, many additional MUX and DEMUX actors and FIFOs, and the memory they require, are needed to model a similar behavior.

5.2. Modeling Periodic Real-Time Constraints in the SDF Model

Summary of a work submitted to RTNS2019 [Honorat 2019]

Motivation & Problematic

Image signal processing systems and visual servoing are typical examples of partially periodic CPSs where certain components are periodic, which means they shall be executed with a fixed repetitive periodic deadline, while other components do not have any real-time constraint. For example, a camera films at a periodic rate and the images arrive at the aperiodic processing components as a stream. Other components may also be periodic, as the input of servo-motors which must be regularly updated. Thus, the processing part often depends on periodic inputs and must provide periodically one or more outputs but does not have to be periodic itself. The flexibility to deviate significantly from periodic operation arises, for example, if data is buffered between components. One possible use-case is the SLAM (Simultaneous Localization And Mapping) application: it constantly retrieves information by camera or lidar and then processes it to reconstruct a map of the environment and move according to it [Wen 2018].

Contribution

This CERBERO contribution focuses on CPSs with periodic and aperiodic components, which are modeled as SDF graphs [Lee 1987]. SDF graphs of CPS often have imposed periodic constraints on all actors of the graph. Our approach is more flexible as any component of the system can be periodic or aperiodic, which leaves more flexibility to the mapping and scheduling process of the application. This flexibility is particularly helpful in the case where several processing parts rely on different sensors.

Given an SDF graph, a number of identical cores where to execute the application, and the Worst-Case Execution Time (WCET) of each actor, the addressed problems are:

- To quickly assess the schedulability of the constrained application, without computing a schedule;
- To compute an offline non-preemptive schedule satisfying the periodicity and precedence constraints.

It is important to note that scheduling time complexity is exponential in the number of tasks to get the optimal solution because it is in general NP-complete [Kwok 1999]. This complexity limits the design of CPSs since optimal schedulers do not scale. In contrast, our approach gives results that are not optimal, but that can be used to quickly build and assess prototypes. In other words, our approach is useful for early-stage design space exploration of scheduling solutions. Optimal schedulers and timing property checkers may still have to be used. However, if they are used, it would only be after the prototyping step, on a small set of prototypes.

Schedulability Necessary Condition: The following set of notations must be introduced to assess the schedulability of an SDF graph containing both actors with and without periodic constraints:

- P – The set of actors of the SDF graph associated with a periodic real-time constraint
- T_π – The period constraint (in seconds) of an SDF actor π.
- C_π – The Worst-Case Execution Time (WCET) of an SDF actor π.
- m – The number of homogeneous processors of the targeted architecture.
- $D_\pi^\uparrow$ – The set of all actors of the SDF graph that are data dependent on the SDF actor π.
- $nblf_\pi^\uparrow(\alpha)$ – Given an SDF actor $\alpha \in D_\pi^\uparrow$, this recursive function computes the number of firings of α that may not be executed before the last firing of SDF actor π.

Given these notations, a necessary condition for the considered SDF graph to be schedulable on m processors is given by the following equation:

$$\forall \pi \in P, \quad \frac{\sum_{\alpha \epsilon D_\pi^\uparrow} nblf_\pi^\uparrow(\alpha) \times C_\alpha}{T_\pi - C_\pi} \leq m$$

The principle of this equation is to verify that within an iteration of the SDF graph, and considering the periodic constraint of periodic actors, there is enough time between two executions of the periodic actor to execute all actors that depend on it on the number of available cores. To be more precise, this equation focuses on the time between the last

execution of a periodic actor within an iteration and the end of this iteration of the SDF graph. Proofs and detailed explanation of these notations, as well as an algorithm to implement the verification of this necessary condition, are available in [Honorat 2019].

Scheduling Algorithm for Partially Periodic SDF Graph: As a necessary schedulability condition, the equation presented in the previous paragraph can be used to discards rapidly unfeasible designs. Indeed, failing to satisfy the aforementioned conditions means that the deployment of the partially periodic SDF graph on the given number of cores is not possible. When the condition is satisfied though, the existence of a valid scheduled is not guaranteed and must still be verified. The purpose of the proposed scheduling algorithm is to make an attempt to find such a valid schedule.

Because the mapping and scheduling of an SDF graph over multiple cores is an NP-Complete optimization problem, finding the optimal solution (i.e. the schedule with the shortest latency in our context) is not possible in polynomial time. For this reason, the proposed scheduling algorithm is a heuristic algorithm that provides no guarantee on the optimality of the obtained schedules but only guarantees their validity with regards to the periodic constraints of the SDF graph.

The proposed scheduling algorithm, detailed in [Honorat 2019], was evaluated on a set of randomly generated graphs in order to evaluate its scalability with regards to the size of applications. Results of this evaluation are reported in Table 1.

m	100 actors	500 actors	1000 actors	5000 actors
2 cores	11 ms	238 ms	898 ms	23286 ms
4 cores	12 ms	251 ms	989 ms	25916 ms
8 cores	11 ms	254 ms	991 ms	26898 ms

Table 1 - Execution Time of the Scheduling Algorithm for Partially Periodic SDF Graphs.

As can be observed in these results, up to 1000 actors the execution time is lower than 1 second, while it reaches around 26 seconds for 5000 tasks. The execution times are slightly increasing with the number of cores. These experimental results confirm the theoretical complexity of the proposed scheduling algorithm in $O(|E| + |V|*log(|V|))$, which is upper bounded by the number of edges $|E|$ in the scheduled graph and the sorting operation on the number of vertices $|V|$. It is important to note that in these experiments, the considered graphs are not the SDF graph themselves, but the equivalent directed acyclic graph derived from them, where each SDF actor is repeated as many times as its number of executions per iteration of the SDF graph.

Use in CERBERO

Time is an essential physical aspect of a CPS and is often translated into real-time constraints for the cyber part. Within CERBERO, both the space exploration and the ocean monitoring use cases require some degree of real-time processing in their cyber part. In the space exploration use-case, for example, the control of the robotic arm requires both monitoring the current position of the different segments of the arm, and the control of its engine with a fixed period. In the case of the ocean monitoring image processing pipeline, the sampling rate of the cameras imposes a fixed periodic constraint to the computations performed by the cyber part of the system.

5.3. Moldable Parameters in Dataflow for Extended Design-Space Exploration

Further work on this topic is scheduled for fall 2019.

Motivations

The DSE phase based on SDF MoCs mostly consists of mapping parallel actor executions on the heterogeneous computational hardware resources of the targeted architecture, and the data transfers on the hardware means of storage and communication. In those cases where the number of parallel actors to map largely exceeds the available resources of the architecture, DSE optimization algorithms face an important increase in the complexity of the mapping problem. In such cases, developers will often manually update the model of their applications to adopt a coarser granularity of description. This coarser granularity translates into less numerous but 'larger' actors to execute. As illustrated in [Hascoet 2017], by carefully adjusting the granularity of the application description, enough elements will be exposed to permit a fair distribution of work on the available hardware resources, with a reasonable complexity exposed to the DSE algorithms.

Envisioned Contribution

The adaptation of the granularity of the application exposed to the DSE algorithm is generally left to the designer of the application. The objective of this contribution is to extend the semantics of SDF models to support the specification of so-called *moldable parameters*. A moldable parameter is a parameter associated to a range of acceptable values, thus leaving the responsibility to the DSE algorithm to select the most appropriate one in its optimization process. In general, moldable parameters are supposed to change only the 'organization' (e.g. like the exposed degree of parallelism) of computations, but not the output they produce. Hence, by specifying moldable parameters in SDFgraphs, it will be possible for the designer to let the DSE algorithms automatically control the parallelism and granularity of the application to obtain the best DSE solution in minimum time.

Use in CERBERO

The moldable parameters will be integrated within PREESM during the CERBERO Project. It is envisioned that DSE optimizations based on this extended semantics will be provided through a connection to AOW.

5.4. Extension of PiSDF MoCs through polyhedral transformations

Motivations

Dataflow applications are modeled as a set of *actors* interconnected through a set of FIFOs, used for sending and receiving information in a streaming fashion. One of the main advantages of modeling an application in such a way is that the structural parallelism can be exploited since it is directly expressed in the graph. Dataflow frameworks take advantage of this and provide mechanisms to automatically parallelize applications to use all the available resources. However, this parallelization is kept within the limits of the application structure: since actors are considered as black boxes, their behavior remains

untouched. As a result, some optimization opportunities seem to be missed, as for example exploiting the intra-actor parallelization is out of the scope of the dataflow MoC.

At this point is where the polyhedral model can become useful. This model is well-known for applying transformations to optimize computationally-intensive applications, focusing on aspects such as data locality or memory usage. Some of the transformations that can be applied to a loop nest are:

- Tiling: This technique splits the loop's iteration space into smaller blocks to improve data locality. Specifically, it tries to improve the usage of the cache memory, maintaining there the data that is going to be reused.
- Loop interchange: Technique to improve memory accesses and, thus, data locality.
- Loop fusion: This technique increases the granularity of the computations to reduce the loop overhead and improve both spatial and temporal data locality.
- Loop unrolling: Technique for improving scheduling and memory usage.
- Loop skewing: As the name implies, this technique skews the execution of an inner loop with respect to an outer one, with the objective of removing dependencies that prevent the code from running in parallel.

As polyhedral transformations are a well-known optimization technique, multiple tools relying on this model can be found in the literature [Bondhugula 2008], [Grosser 2011], [Pop 2006]. However, the restrictions imposed by this model are usually so tight that most codes are not amenable to the model since it does not allow, for instance, dynamic behaviors. That is the reason why most tools only work at compile-time, as Polly [Grosser 2011], Graphite [Pop 2006] or Pluto [Bondhugula 2008] However, in recent years, several tools try to extend the polyhedral scope to overcome these limitations as the APOLLO (Automatic speculative POLyhedral Loop Optimizer)[Caamaño 2017].

APOLLO applies polyhedral optimizations on-the-fly to loop nests that cannot be optimized at compile time. In addition, in contrast to the rest of the existing tools, it can handle not only *for* loops, but any kind of loop nest. To do so, APOLLO relies on a speculative system that builds a prediction model to support dynamic transformations which are applied to the original code thanks to LLVM-JIT (Low Level Virtual Machine - Just In Time compilation).

Contribution

For the previous reason, combining a dataflow framework as PREESM with a tool like APOLLO can lead to finally be able to exploit the optimization possibilities within the actors, hidden until now from the dataflow perspective. These optimization possibilities can result in creating more threads to exploit all the available resources, or just apply optimizations to improve the memory usage, resulting in a speedup but without affecting the number of threads PREESM handles, that is, without modifying the actor-core mapping performed by PREESM.

To efficiently combine PREESM with APOLLO, several actions have been needed. The main limitation of APOLLO is that, in order to be an efficient runtime system, it applies the first transformation that it finds, since evaluating several ones is a computationally expensive task. However, this does not mean that the transformation is the most efficient,

or that it is efficient at all. In this sense, the context of dataflow applications opens new possibilities: since dataflow applications are designed to be executed in a loop, the first iterations of this loop could be considered as a training phase to test different transformations, so as to choose the most efficient one at the end of this phase and maintain it until the end, considering the performance as the main criterion. To do so, APOLLO needs to store information about the transformations already tried and relate it to the actor parameters. This mechanism, developed in the context of CERBERO, is known as *multi-versioning*, and it has already been implemented and tested within APOLLO.

Furthermore, neither APOLLO nor the libraries upon which it is built have been designed to be used in a multithreaded context. This is the case of dataflow applications since actors that are being executed simultaneously can make a call to APOLLO at the same time. As a result, all the modifications needed to make APOLLO thread-safe have been made. This includes both APOLLO runtime system and some libraries, as Pluto and Piplib.

To exemplify the new functionalities, a toy example is going to be presented. In this example, a PREESM application to multiply two matrices has been implemented. This application contains four actors: two for generating the matrices to be multiplied, another one for performing the multiplication, and the last one to store the result. This example has been used to generate two different configurations:

- sequential configuration (1-core): in which the computation is performed by one thread,
- multithreaded configuration (2-cores): which uses two threads to compute the matrix multiplication in parallel.

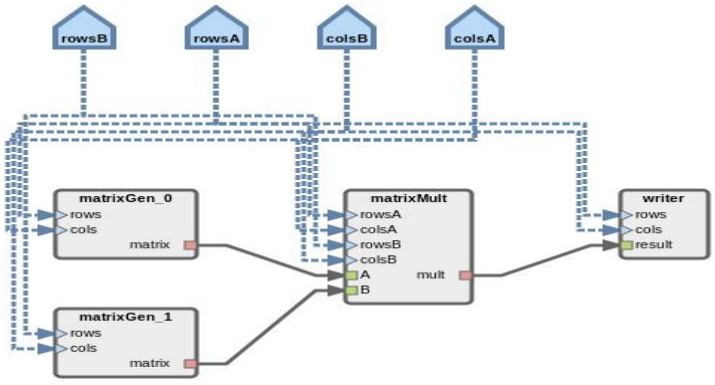

Figure 9 - PREESM toy example to demonstrate CERBERO polyhedral multiversioning mechanism

Table 2 gathers the results, in seconds, for both configurations and three different scenarios: when compiled with *gcc*, when compiled with APOLLO and when compiled with APOLLO and using the *multi-versioning* mechanism. Table I also presents the speedups obtained with the configurations using APOLLO, both with respect to the execution time obtained with GCC. It should be highlighted that, to use APOLLO, the user needs to add a specific pragma (#pragma dcop) enclosing the loop that is going to be optimized, and then compile the code generated by PREESM with APOLLO. Additionally,

please note that these times have been measured in an Intel Core i7-4790 with 4 cores running at 3.6 GHz and 32GB RAM.

As can be observed, the execution times when using APOLLO are drastically reduced, and the *multi-versioning* mechanism makes them even smaller, which validates the initial hypothesis of the potential advantages of applying polyhedral transformations to dataflow applications.

Matrix size	1-core					2-cores				
	GCC	APOLLO	Partial speedup	APOLLO + multi versioning	Global speedup	GCC	APOLLO	Partial speedup	APOLLO + multiVersioning	Global speedup
1000 x 1000	3.1	2.3	1.3x	0.7	3.3x	1.5	1.5	1x	0.4	3.8x
2000 x 2000	80.0	7.1	11.3x	5.4	14.9x	41.5	4.8	8.6x	2.7	15.4x
3000 x 3000	403.4	20.0	20.2x	18.2	22.2x	207.3	12.3	16.9x	9.2	22.5x

Table 2 - Execution time, in seconds, of the matrix multiplication example for three different matrix sizes using one and two cores, and with and without APOLLO.

Use in CERBERO

The proposed combination between APOLLO & PREESM, and their integration into the CERBERO toolchain will make it possible to perform powerful polyhedral optimization seamlessly for all software parts of the CPS systems modeled with the dataflow models of computation. In particular, massively parallel computation with a fine granularity of parallelizable actors, such as the computation used in matrix operations, are particularly well suited to benefit from polyhedral optimizations.

5.5. Numerical Representation of Directed Acyclic Graphs for Dataflow-Based Embedded Runtime Resources Allocation

Summary of a work submitted to ESWEEK2019 [Arrestier 2019]

Motivation & Problematic

In an embedded context, taking fast and efficient resource allocation decisions requires an efficient intermediate representation of the application. Using compact and expressive dataflow MoC, such as the Cyclostatic Dataflow (CSDF) [Bilsen 1996], the PiSDF [Desnos 2013] or the Interface-Based SDF (IBSDF) [Piat 2009] MoC, allows for a high-level description of an application. However, the more compact and expressive the representation, the more costly it can be to extract information. For instance, extracting fine-grain dependencies information from a Directed Acyclic Graph (DAG) is straightforward whereas it is first necessary to compute model transformations on a CSDF-based application to do so. The more expensive stages of expressive model analysis have led to the more frequent use of DAG-based models in programming frameworks. Frameworks such as StarPU [Augonnet 2009], OpenVX [Khronos 2013] or TensorFlow [Abadi 2016] rely on DAG dataflow MoC. DAG efficiently model directed workflows with

task-level parallelism. However, complex structures such as loops are cumbersome to model with DAG due to the fact that the entire loops have to be unrolled.

There is a paradox in developing more expressive and more compact dataflow MoC and the fact that analysis methods often remain oriented toward the need of expanding expressive graphs into DAGs. For example, the SDF graph of Figure 10 with only 4 actors can be transformed into the equivalent DAG presented in Figure 11 with 30 actors. The large size of the resulting DAG is due to the repetition vectors of the original actors, but also to the addition of "special" actors, noted *F* and *J*, responsible for distributing tokens to several consumers, or gathering tokens from several producers.

The SPiDER tool uses a PiSDF input representation of an application and performs a transformation to an expanded intermediate DAG representation to perform the scheduling and mapping of the application onto multi-core platforms. Construction of the intermediate representation is a costly step that needs to be repeated multiple times in the context of dynamically reconfigurable applications.

Contribution

The CERBERO contribution is a numerical model of the expanded DAG representation of the PiSDF MoC, compatible with both the SDF and IBSDF MoCs, which makes it possible not to build the intermediate DAG when scheduling the application, thus improving significantly the performance of the embedded runtime. The objective of the proposed representation is to allow DAG oriented analysis methods while maintaining the compactness and expressiveness of the targeted dataflow MoC.

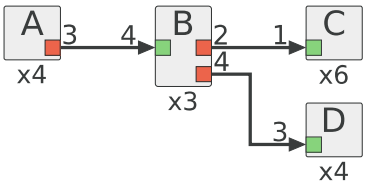

Figure 10 - SDF Graph Example. The number of firings per graph iteration is given below each actor.

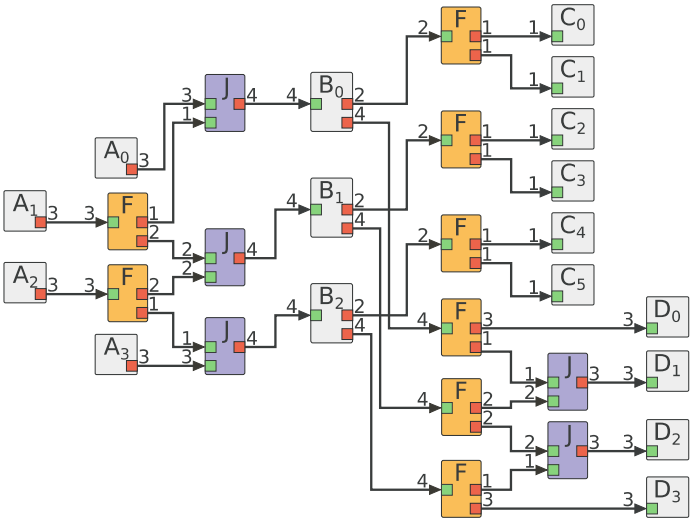

Figure 11 - Directed Acyclic Graph (DAG) derived from the SDF Graph of Figure 10.

Intuitively, the idea behind the proposed numerical representation is to compute on-the-fly the dependencies between the different firings of actors of the original SDF graph when scheduling the application, instead of computing these dependencies when building the DAG. The drawback of this approach is that if a data-dependency needs to be evaluated several times by the scheduling algorithm, for example when checking whether all

predecessors of a given actor firing have completed their execution, then this data dependency will be re-computed every time, while the DAG can be used directly. The advantage of the proposed numerical representation is that it requires very little memory compared to the amount needed to allocate and store all data structures of a DAG. The proposed set of notations and theorem for this numerical representation are thoroughly detailed in [Arrestier 2019].

An experimental evaluation of the proposed numerical representation implement within SPiDER was conducted on four machine learning (SqueezeNet, Reinforcement Learning) and computer vision (Stabilization, Sobel-Morpho) algorithms. The amount of memory required to store the DAG and the numerical model for each application is presented in Table 3. The proposed numerical representation for the four applications is more than 94% more compact than the DAG representation, making it an interesting solution for embedded architectures where memory resources are generally scarce.

Application	DAG	Num. Representation	Gain
1- SqueezeNet	8405 kB	515 kB	94%
2 - Reinforcement Learning	5183 kB	70 kB	99%
3 - Stabilization	782 kB	12 kB	98%
4 - Sobel-Morpho	405 kB	7 kB	98%

Table 3 - Memory Footprint of the DAG and Numerical Representation used for Scheduling in SPiDER.

Then the latency overhead of the numerical representation was compared with the building time of the DAG for the execution of the four applications on three different architectures: an Intel i7 core, a Jetson TX2, and an Odroid-XU3 board. Results of this comparison are presented in Table 4. The IR column corresponds the time taken to initialize the structures of the Numerical Representation minus the time taken for building the DAG. The Sched. column corresponds to the time taken by the scheduling algorithm when the numerical representation, and the on-the-fly computation it involves, is used, minus the time taken by the scheduling algorithm when the DAG is used. A negative number in the IR and Sched column indicates that the scheduling algorithm is faster when using the numerical representation than when using the DAG, and a positive means the opposite. The Gain column corresponds to the overall gain in the latency of the scheduling process when using the Numerical representation, relatively to the total time taken when building and using a DAG.

	i7			TX2			XU3		
App.	IR	Sched	Gain	IR	Sched.	Gain	IR	Sched.	Gain
1	-6.89 ms	+2.34 ms	-47%	-38.77 ms	-8.01 ms	-76%	-77.87 ms	+0.09 ms	-76%
2	-0.69 ms	+0.22 ms	-48%	-5.48 ms	+0.41 ms	-78%	-10.70 ms	+0.93 ms	-76%
3	-0.12 ms	+0.04 ms	-54%	-0.61 ms	+0.05 ms	-75%	-1.51 ms	+0.06 ms	-77%
4	-0.06 ms	+0.00 ms	-79%	-0.21 ms	-0.02 ms	-48%	-0.63 ms	-0.05 ms	-85%

Table 4 - Latency gain and overhead of the Numerical Representation in the Scheduling Process, compared to the legacy DAG-based representation, on three architectures and four application.

Use in CERBERO

Dynamic reconfiguration capabilities are at the core of the CERBERO Toolchain. To be usable in highly reactive self-adaptive scenarios where application computational performance is an important KPI of the CPS, such as in the space exploration or ocean monitoring use cases, the overhead of reconfiguration managers should remain as contained as possible. The proposed numerical representation helps achieve this purpose by drastically lowering the latency overhead of the dataflow mapping/scheduling process. The proposed numerical representation was integrated within the SPiDER runtime to make it seamlessly usable for all users of the CERBERO Toolchain.

6. Conclusions

Models of computations are the foundation of the CERBERO Toolchain as their semantics provides the necessary formalism for the design, the analysis, the optimization, the refinement, and the runtime management of complex CPS. The models of computations supported in the different tools of the CERBERO toolchain were surveyed in D3.5, and are reminded in Table 6 (slightly updated with new supporting tools for the PiSDF MoC).

Dataflow models of computation used in CERBERO, which are tailored for modeling and managing parallel and reconfigurable behaviors for mid to low-level software and hardware system, are a key element of the CERBERO self-adaptation reconfiguration loop. For this reason, MoC-related technical requirements of CERBERO, reminded in Table 1 have been identified and a lot of research effort has been, and will be, spent accordingly to improve the capabilities of these dataflow models by:

- improving their expressiveness for capturing states of applications,
- integrating real-time concerns in the design and analysis of application graphs,
- offering new optimization opportunities for design space exploration with moldable parameters (planned),
- studying the compatibility and use of polyhedral optimization techniques with the dataflow approach,
- improving the performance of dataflow graphs management for runtime mapping and scheduling.

The integration and support for these achievements within tools (PREESM, SPiDER, PAPIFY, ARTICo^3, MDC) of the CERBERO toolchain make them readily available to all, including for the support of the different use-cases implemented with them.

Table 6 – CERBERO Tools to MoC Mapping. S: support, P: planned support within CERBERO duration

	SDF	PiSDF	PN	KPN	DPN	RTL	DES	SCE	TS
MECA								S	
VT									S
DynAA			S	S			S		
AOW									
PREESM	S	S							
SPiDER		S							
PAPIFY		S			S				
JIT HW						S			
ARTICo^3		S				S			
MDC	S	S			S	S			

7. References

[Abadi 2016] Matin Abadi et al. 2016. TensorFlow: A system for large-scale machine learning. 265–283.

[Arrestier 2018] Florian Arrestier, Karol Desnos, Maxime Pelcat, Julien Heulot, Eduardo Juarez, and Daniel Menard. 2018. Delays and states in dataflow models of computation. In *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation* (SAMOS '18). ACM, New York, NY, USA, 47-54. DOI: https://doi-org.insis.bib.cnrs.fr/10.1145/3229631.3229645

[Arrestier 2019] Arrestier F., Desnos K., Juarez E., Menard D. *Numerical Representation of Directed Acyclic Graphs for Efficient Dataflow Embedded Resource Allocation.* Submission under review (2nd round) to ESWEEK 2019. *This publication is currently under review. It can be made available upon request in a confidential manner*.

[Augonnet 2009] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. (2009), 16.

[Bhattacharya 2001] Bhattacharya, Bishnupriya, and Shuvra S. Bhattacharyya. "Parameterized dataflow modeling for DSP systems." *IEEE Transactions on Signal Processing* 49.10 (2001): 2408-2421.

[Bhattacharyya 2006] S.S. Bhattacharyya and W.S. Levine. *Optimization of signal processing software for control system implementation.* In Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE, pages 1562-1567. IEEE, 2006

[Bilsen 1996] Bilsen, G., Engels, M., Lauwereins, R., & Peperstraete, J. (1996). *Cyclo-static dataflow*. *IEEE Transactions on signal processing*, *44*(2), 397-408.

[Bondhugula 2008] Uday Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model International Conference on Compiler Construction (ETAPS CC), Apr 2008, Budapest, Hungary.

[Bouakaz 2012] Bouakaz, A., Talpin, J. P., & Vitek, J. (2012, June). Affine data-flow graphs for the synthesis of hard real-time applications. In *Application of Concurrency to System Design (ACSD), 2012 12th International Conference on* (pp. 183-192). IEEE.

[Caamaño 2017] Caamaño, Juan Manuel Martinez, et al. "APOLLO: Automatic speculative polyhedral loop optimizer." IMPACT 2017-7th International Workshop on Polyhedral Compilation Techniques. 2017.

[Davis 1999] Davis II, J., Goel, M., Hylands, C., Kienhuis, B., Lee, E. A., Liu, J., ... & Smyth, N. (1999). *Overview of the Ptolemy project* (Vol. 99). ERL Technical Report UCB/ERL.

[Desnos 2013] Desnos K., Pelcat M., Nezan J.-F., Bhattacharyya S., Aridhi S. *PIMM: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration*,In *proceedings of* SAMOS XIII, IEEE, 2013.

[Grosser 2011] Grosser, Tobias, et al. "Polly-Polyhedral optimization in LLVM." Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT). Vol. 2011. 2011.

[Heulot 2014] Heulot, J., Pelcat, M., Desnos, K., Nezan, J. F., & Aridhi, S. (2014, September). SPiDER: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in* (pp. 167-171). IEEE.

[Honorat 2019] Honorat A., Desnos K., Bhattacharyya S., Nezan J.-F. *Scheduling Analysis of Partially Periodic Real-Time Constraints in Synchronous Dataflow Graphs.* Submission under review to RTNS 2019
This publication is currently under review. It can be made available upon request in a confidential manner

[Khronos 2013] Khronos Group. 2013. The OpenVX API for hardware acceleration. In http://www.khronos.org/openvx.

[Klikpo 2016] Klikpo, E. C., Khatib, J., & Munier-Kordon, A. (2016, April). Modeling multi-periodic simulink systems by synchronous dataflow graphs. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE* (pp. 1-10). IEEE

[Kwok 1999] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static *Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors*. ACM Comput. Surv. 31, 4 (Dec. 1999), 406–471. https://doi.org/10.1145/344588.344618

[Lee 1987] E.A. Lee and D.G. Messerschmitt. *Synchronous dataflow*. Proceedings of the IEEE, 75(9):1235-1245, sept. 1987.

[Lee 1995] Lee, E. and Park, T. (1995). Dataflow Process Networks. In Proceedings of the IEEE, volume 83, pages 773-799.

[Lee 2017] Edward A. Lee and Sanjit A. Seshia, *"Introduction to Embedded Systems, A Cyber-Physical Systems Approach"*, Second Edition, MIT Press, ISBN 978-0-262-53381-2, 2017.

[Ostroff 1995] J.S. Ostroff. *Abstraction and composition of discrete real-time systems*. Proc. of CASE, pp 370-380, 1995.

[Palumbo 2017] F. Palumbo, C. Sau, T. Fanni, P. Meloni and L. Raffo, SS-design: Dataflow-based design of coarse-grained: Reconfigurable platforms reconfigurable platform composer tool project. In proceedings of the IEEE International Workshop on Signal Processing Systems, 2016.

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

[Piat 2009] Piat, J., Bhattacharyya, S. S., & Raulet, M. (2009, October). Interface-based hierarchy for synchronous data-flow graphs. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on* (pp. 145-150). IEEE.

[Pop 2006] Pop, Sebastian & Cohen, Albert & Bastoul, Cédric & Girbal, Sylvain & Silber, Georges-André & Vasilache, Nicolas. (2006). GRAPHITE: Polyhedral analyses and optimizations for GCC. Proceedings of the GCC Developers' Summit 2006.

[Savage 1998] Savage, J.E. *Models of Computation*, Volume 136, Addison-Wesley Readings, MA, 1998

[Shen 2011] Shen, C. C., Wang, L. H., Cho, I., Kim, S., Won, S., Plishker, W., & Bhattacharyya, S. S. (2011). *The DSPCAD lightweight dataflow environment: Introduction to LIDE version 0.1*.

[Sriram 2009] Sundararajan Sriram and Shuvra S. Bhattacharyya. 2009. Embedded Multiprocessors: Scheduling and Synchronization, Second Edition. CRC press.

[Stuijk 2006] Stuijk, S., Geilen, M., & Basten, T. (2006, June). Sdf^ 3: Sdf for free. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on* (pp. 276-278). IEEE.

[Tripakis 2013] Tripakis, S., Bui, D., Geilen, M., Rodiers, B., & Lee, E. A. (2013). Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, *12*(3), 83.

[Van Hasselt 2007] Hado Van Hasselt and Marco A. Wiering. 2007. Reinforcement learning in continuous action spaces. In Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on. IEEE, 272–279

[Wen 2018] ShuhuanWen, Miao Sheng, Chunli Ma, Zhen Li, H. K. Lam, Yongsheng Zhao, and Jingrong Ma. 2018. Camera Recognition and Laser Detection based on EKF-SLAM in the Autonomous Navigation of Humanoid Robot. Journal of Intelligent & Robotic Systems 92, 2 (01 Oct 2018), 265–277. https://doi.org/10.1007/s10846-017-0712-5