Explicit Control of Dataflow Graphs with MARTE/CCSL

Jean-Vivien Millo, Amine Oueslati, Emilien Kaufman, Julien DeAntoni,
Frederic Mallet and Robert de Simone
Université Nice Cote d'Azur, CNRS, Inria, I3S, Sophia Antipolis, France

Keywords: Dataflow, Platform based Design, Scheduling.

Abstract: Process Networks are a means to describe streaming embedded applications. They rely on explicit representation of task concurrency, pipeline and data-flow. Originally, Data-Flow Process Network (DFPN) representations are independent from any execution platform support model. Such independence is actually what allows looking next for adequate mappings. Mapping deals with scheduling and distribution of computation tasks onto processing resources, but also distribution of communications to interconnects and memory resources.
This design approach requires a level of description of execution platforms that is both accurate and simple. Recent platforms are composed of repeated elements with global interconnection (GPU, MPPA). A parametric description could help achieving both requirements.
Then, we argue that a model-driven engineering approach may allow to unfold and expand an original DFPN model, in our case a so-called Synchronous DataFlow graph (SDF) into a model such that: a) the original description is a quotient refolding of the expanded one, and b) the mapping to a platform model is a grouping of tasks according to their resource allocation.
Then, given such unfolding, we consider how to express the allocation and the real-time constraints. We do this by capturing the entire system in CCSL (Clock Constraint Specification Language). CCSL allows to capture linear but also synchronous constraints.

1 INTRODUCTION

Synchronous Data Flow (SDF) (Lee and Messerschmitt, 1987b) graphs are a popular model of choice to support Platform-Based Design approaches (also called Y-Chart flow). There are obvious reasons for that: SDF makes explicit the (in)dependences of dataflow and the potential concurrency, it abstracts data values while preserving sizes for bandwidth considerations; it is architecture-agnostic. Like most Process Network models it enjoys conflict-freeness properties, ensuring functional determinism. Then the one single issue remaining for efficient model-level abstract compilation is to find a best-fit mapping onto a provided architecture model. Mapping here refers both to spatial allocation of both computations and communications onto processing, memory and interconnect resources, and the temporal scheduling in case some resources need to be shared.

While the original SDF model is rightfully independent from the architecture and inherent mapping constraints, these have to be included and decided upon, possibly incrementally, as design goes down the flow. In a sense the application model has to be taken step-by-step from architecture-agnostic to architecture-aware. There, SDF shows of course limitations, as additional information of different nature must enter the modeling framework. This can be achieved in essentially two ways: either the SDF model itself is refined and complexified (usually out of SDF syntax *stricto sensu*), so that the architectural structure and the temporal organization transpire below it; or, in a modular fashion, additional constructs are added on the side, with precise links to the existing models. The former approach is appropriate when the extension of the expressiveness is limited. Otherwise, the latter way has to be preferred.

Going down the latter way, we suggest that an SDF graph can be augmented with 1/an Occurrence Flow Graph (OFG) where every occurrence of SDF agents are explicit[1]. OFG shows the agent concurrency in addition to the pipeline; 2/ a parametric architecture model in order to elegantly capture modern architectures composed of repeated tiles such as in MPPA or GPU. Such a model comes with the associated parametric allocation model.

[1]Similarly to the transformation of SDF in HSDF

Millo J., Oueslati A., Kaufman E., Mallet F. and de Simone R.
Explicit Control of Dataflow Graphs with MARTE/CCSL.
DOI: 10.5220/0006269505420549
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 542-549
ISBN: 978-989-758-210-3

If we move apart the problem of finding the best binding of computation and communication of the architecture, the next problem is to find the best schedule of the application that satisfies the execution constraints imposed by application functionality, the execution platforms and the real time requirements. The scheduling algorithms are often tailored for a given optimization criteria (*e.g.,* max throughput, single appearance) whereas every case study would have its own objective. It must be possible to reuse the same design flow (Stuijk et al., 2006; Bamakhrama and Stefanov, 2011; Karczmarek et al., 2003) on different case studies with each time an original optimization criteria.

There is a lack for a dedicated model of the control which is able to represent all the acceptable schedules of the application according to constraints of different nature ranging from performance requirements to platform allocation (or any other hardware related concerns).

In the current paper we show how a wide range of constraints inherited from the architecture could be formally expressed in a language that describes mapping conditions (and scheduling constraints) that will further restrict the potential schedules of the original SDF description. For this we use the *Clock Constraint Specification Language (*CCSL*)* (André, 2009) formalism. CCSL is specifically devoted to the temporal annotation of relevant scheduling patterns on top of classical behavioral models. It allows to deal with such scheduling constraints as formal parts of the design, to conduct formal proofs of schedule validity as well as high-level simulation. Even automatic synthesis of optimal schedules can be achieved in some case, using techniques borrowed from model-checking automatic verification.

By exploiting the explicit control in CCSL, it is still possible to find a scheduling that optimizes a original criteria, this time after an explicit consideration of assumptions. It is also possible to drive analysis, directly on the CCSL structure or by using a projection to an existing analysis model as in (Mallet and de Simone, 2015; Yu et al., 2011).

The paper is organized as follows: Section 2 browses the state of the art of design flows for many core architectures based on DFPN. Section 3 introduces Occurrence Flow Graph and a parametric architecture model. Section 4 shows how allocations constraints of different natures can be captured in CCSL. Section 5 concludes this article.

2 RELATED WORK

Abstract representation of streaming applications as dataflow graph models goes a long way back in time, to Kahn process networks (Kahn, 1974), Commoner/Holt's marked graphs (Commoner et al., 1971), or even Karp et *al.* systems of uniform recurrence equations (Karp et al., 1967).

There was a renewal of interest in the 1980's for the class of so-called *Dataflow Process Networks (DFPNs)*, starting with SDF(Lee and Messerschmitt, 1987a) and successive variants (boolean(Buck, 1993), cycloStatic(Bilsen et al., 1995).

More recently, the emergence of many-core architectures has polarized DFPNs as natural concurrent models to design embedded applications for (heterogeneous) parallel architectures. The Holy Grail is a methodology considering a description of the application and a description of the target architecture that computes *automatically* the best allocation according to some optimization criteria. Here allocation has to be taken into its widest sense: (i) binding computations on processing elements, FIFO on memory, and data flows on communication topology, (ii) scheduling computations and memory accesses, (iii) routing communications in space and time. Such a methodology has been approached through different angles.

SDF3 (Stuijk et al., 2006) provides *SDFG-based MP-SoC design flow* (Stuijk, 2007) based on successive refinements and iterations of the original SDF model. It implements self time scheduling and two scheduling policies: list scheduling and single appearance scheduling (minimizes code size).

Streamit experiments different scheduling policies (Karczmarek et al., 2003) of a dataflow graph. Provided a balanced input dataflow graph and a scheduling policy, a static schedule that achieves one period of the graph is generated. These policies are not constrained but they result in different buffer sizes, code sizes and latencies.

Syndex (Grandpierre et al., 1999) also provides a complete environment using allocation and scheduling heuristics.

In some cases, the optimisation criteria is known but the scheduling algorithm is not described. For instance the sigmaC toolchainallows static scheduling and routing decisions on a network on chip architecture.

These methodologies are often couple with a specific platform as in streamIt with Raw/ Tilera (Karczmarek et al., 2003), SDF3 with COMPSoC/ Aelite (Goossens and Hansson, 2010), sigmaC with Kalray/ MPPA [2], and PEDF with SThorM (Melpignano et al.,

[2]http://www.kalray.eu/products/mppa-manycore

2012).

The TIMES tool (Amnell et al., 2004) allows modeling a dataflow process network and adding constraints caused by shared resources and deadlines for each task. The scheduling policy is provided to check if the constraints are satisfied. On the contrary, we provide a structure that allows deriving a schedule which satisfies the constraints, either a static schedule or a schedule or a policy that would cause a valid schedule.

These design flows have the only restriction to focus on predefined optimization criteria conducting the allocation decisions. We think that the binding between the design methodology and the optimization criteria is artificial and unwelcome. This article proposes a framework to capture the application, the architecture and the allocation constraints to enable the user to explore all the possible schedules matching the constraints. Thus, any optimization criteria can be applied to select the best scheduling according to the specific needs of the designer (time, memory, consumption, end to end latency). We offer our framework to enrich existing design flows with the freedom to select original optimization criteria. In our approach, we consider the binding of communications and computations onto architectural elements to be given. Routing is not (yet) considered.

3 SDF, OCCURRENCE FLOW GRAPH AND ALLOCATION

This section has three parts: First, we provide a description of the SDF model. Note that the place are explicitly represented. Second, we show how the agent concurrency can be explicitly extracted from a SDF graph by representing every instance of an SDF agent over a period of execution, this is the Occurrence Flow Graph (OFG). Last, we present a simple parametric model of an architecture and the associated parametric allocation model.

3.1 SDF

The Synchronous Data Flow model (Lee and Messerschmitt, 1987b) (SDF) is a bipartite graph where edges can be divided in two disjoint sets named Agents and Places. An agent is a functional block that consumes and produces a static amount of data (or tokens) in places. The arcs are weighted and relate agents with places and vice-versa but two edges from the same set are never linked together. The marking associates tokens with each place, the initial marking is the initial number of tokens in all places.

The constraint on the number of inputs and outputs of every place guarantees that a token can be used by only one agent; the fact that an agent uses tokens to fire (or run) never disables another agent. Thus SDF is said to be conflict-free in the sense of Petri net (Petri, 1962) or monotonic in the sense of Kahn Process Network (Kahn, 1974) or confluent in the sense of CCS (Milner, 1982).

The operator $^\bullet$ can be applied to any edge e (agent or place) to designate either the preset ($^\bullet e$) or the postset ($e^\bullet$). Note that the preset and the postset of a place are composed of a single agent.

An agent is said fireable when every place in its preset has a marking greater than the input weight of the agent.

When an agent is fired, it consumes tokens in every input place and produces tokens in every output places.

In the scope of this article, an SDF graph models an application where the agents represent the different filters (or actors) that compose the application. The agents can be triggered concurrently. The places represent locations in memory[3]. The arcs give the flows of data, *i.e.,* data dependencies among agents. The presence of a token in a place represents the availability of a data element in memory. An agent without incoming (resp. outgoing) arc represents a global input (resp. output) of the application.

For instance, Figure 1 gives the SDF representation of a parallel sorting algorithm where $n <= k$. A set of 2^n elements are sorted by 2^{n-k} sorters.

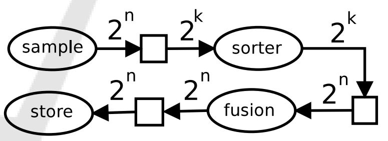

Figure 1: An SDF graph of a parallel sorting algorithm.

Flow Preservation, Repetition Vector and Period. The flow preservation condition is a necessary condition for the existence of an infinite bounded execution. On the contrary, when it is not, the SDF graph is called pathological or inconsistent. Consequently, any infinite execution is either unbounded or leading to starvation (deadlock).

As explained in E. Lee and D. Messerschmitt's original work on SDF (Lee and Messerschmitt, 1987b), any SDF graph G can be represented as a matrix $\Gamma(G)$, called the topology matrix, assigning a column to each agent and a row to each place. An en-

[3]the notion of memory used here is generic. It could be any kind of memory: *e.g.,* central memory (RAM), CPU/GPU register, scratch-pad, communication buffer

try in the topology matrix gives the width of the arc relating the agent to the place (or vice versa).

An SDF graph is *flow preserving* (or balanced) if and only if the rank of the topology matrix is equals to $|N| - 1$. Thus pathological cases occur when the rank of the topology matrix is $|N|$.

When G is flow preserving, the equation $\Gamma(G) * X = 0$ has a solution and X is the repetition vector of G of size $|N|$. X provides the number of firing (activations) of every agent so that the flows are balanced. When G is flow preserving, the repetition vector of the sorting algorithm is $[sample = 1, sorter = 2^{n-k}, fusion = 1, store = 1]$.

A sequence of execution of an SDF graph such that every agent a is fired (or run) $X(a)$ times is called a *period*. The pipelined execution of an SDF graph is the interleaving of periods.

3.2 Occurrence Flow Graph (OFG)

The OFG of a flow preserving SDF graph G is the explicit representation of agent concurrency. An OFG is based on the decomposition of SDF agents into several occurrences over a period of execution similarly to the decomposition of SDF into HSDF (or Marked Graph (Commoner et al., 1971)) (de Groote et al., 2013). In Figure 1, all the (2^{n-k}) occurrences of the agent *sorter* can be run concurrently. The OFG of this SDF graph makes explicit this freedom.

In an OFG, every agent a is decomposed into $X(a)$ *instances* representing the different occurrences during a period of execution. The i^{th} instance (denoted a_i) represents the class of all the $(k * X(a) + i)^{th}$ occurrences of the agent.

The instances of the OFG are related with *control flows* indicating causes or precedences between firing of the instances. There are two kinds of flows in the OFG according to the two following rules.

First rule: the $X(a)$ instances of every agent a are ordered so that the $i+1^{th}$ instance of a cannot be fired before the i^{th} instance. Formally, a_i causes $a_{i+1 mod X(a)}$ (One cannot *re*start if it has not started before). Note that a cause allows the simultaneous execution of successive instances.

Second rule: the flows are derived from the places in the SDF graph. If the i^{th} firing of an agent a produces n tokens that are consumed by the j^{th} firing of an agent a' then a_i precedes a'_j in the OFG. Note that a precedence does not allow the simultaneous execution of successive instances (on the same token however a pipeline execution is still possible). The input and output weights of the flows are n. When the tokens produced by an instance of an agent are used by many instances of the successor, the partition is made on the flows. Similar partitioning is made for the instances of an agent consuming tokens from many instances of its predecessor.

The initial marking of the OFG is computed as follows: if the produced tokens are present initially, the corresponding control flow has these tokens. Moreover, the control flow relating $a_{X(a)-1}$ and a_0 contains a token whereas every other control flow a_i to a_{i+1} has no token.

The OFG is by nature an SDF graph where instances are the agents and control flows are sequences of arc$\rightarrow$place$\rightarrow$arc between pairs of agents. However, the distinction between cause (first rule) and precedence (second rule) cannot be captured in SDF whereas this distinction is natural in CCSL.

Figure 2 shows a flow preserving SDF graph and its corresponding OFG (omitted weights are 1). Dotted lines represent Causes whereas plain lines are for Precedences. The repetition vector is $X = [a = 3, b = 2, c = 1]$. Agent a must be fired twice before b is and one token remains. The third firing of a enough tokens for b to be fired a second time.

There is initially one token in the place between b and c meaning that the last instance of b (b_2) has provided a token for the execution of the first instance of c (c_1). Similarly, the last instance of c (still c_1) has provided a token for the execution of the first instance of b (b_1).

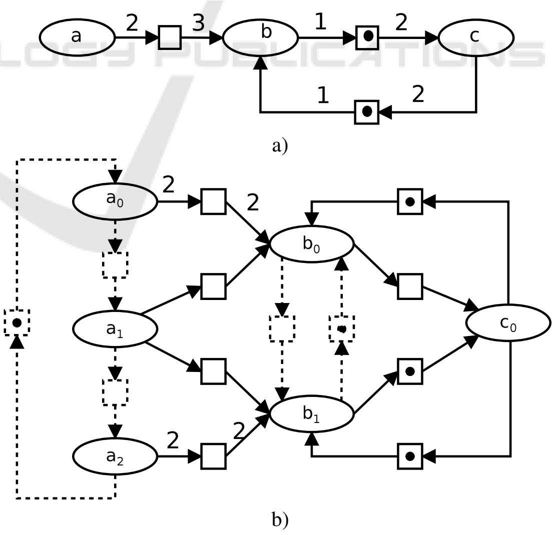

Figure 2: An SDF graph and its corresponding OFG.

Converting SDF Graphs into OFG. As a first approximation, we convert an SDF graph into an Homogeneous SDF (multi)-graph as described in (Sriram and Bhattacharyya, 2012) (p.45). Note that the major drawback of HSDF is the explosion of the number of arcs. Our actual implementation avoids that problem but due to space limitation, we leave the algorithm out of this paper.

3.3 Parametric Architecture Model

To model correctly the upcoming parallel and embedded architectures, we need to take advantage of their regular nature. For example, an MPPA-256 (Kalray, 2014) is the two dimensional repetition on a simpler tile composed of sixteen processors. The architecture is complemented with a NoC interconnecting the sixteen tiles. It is thus convenient to use a parametric architecture model where a tile is uniquely defined and explicitly declared to be repeated.

Let us consider a simple model of components with ports. To capture different types of constraints, we distinguish three types of components: computation resources, memories, and interconnects.

Each component is indexed by its number of repetitions belonging to a finite domain. When two repeated components are connected together, a function maps the indices of the first domain to the indices of the second. The simplest function is $f(i) = i$ that maps the i components together when the two domains are identical. This function can be used either to expand this model or to perform analysis without expansion.

Figure 4 shows how to build a mesh using the two following functions:

$$h(i) \begin{cases} i+1 \text{ if } i\%2 = 0 \\ i-1 \text{ otherwise} \end{cases} \qquad g(i) = i+2 \bmod 4$$

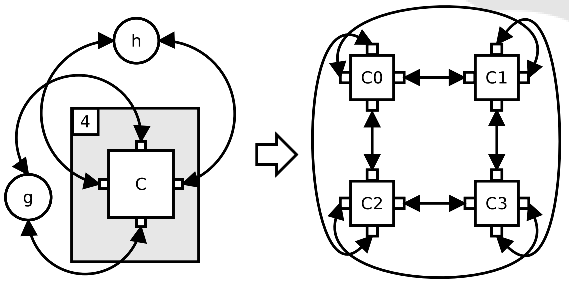

Figure 3: A torus expressed in a parametric way.

Let us consider a generic multi-core with one level of cache and a simple repeated pattern 4. The cache memory can be read/written as a regular memory (this is a "scratchpad memory"). It is admitted that this architecture does not scale much because of the shared interconnect and memory. Our approach allows to explore the possible schedules and for instance to determine at which point some given real time requirements cannot be met (whatever the scheduling policy).

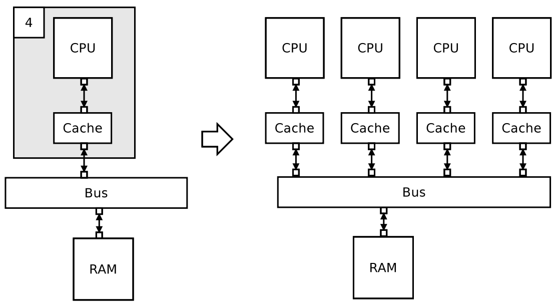

Figure 4: A generic multi-core architecture.

Parametric Allocation Model. Allocating directly the agents on processing elements would mean that every instance of this agent runs on the same processing element. This limits the potential parallelism. However, the OFG shows instances of the same agent that can run concurrently (w.r.t. data flow constraints). Thus we map instances to repeated processing elements.

Definition 1. *Let Pe be the set of P processing elements indexed from 0 to P-1. For each agent, the mapping function allocates every instance of an Agent to processing elements. The agent mapping M is the set of the mapping functions for each agent.*
$M = M_a(i), M_b(i), \ldots$ *where a, b are agents*
$M_{agent} : i \in [0..X(a)-1] \rightarrow p \in [0..P-1]$

Consider the SDF in Figure 2 mapped on a platform with three processing elements, an acceptable mapping is:
$M_A = (M_a, M_b, M_c)$
$M_a(i) = i \mod 2 == 1; \quad M_b(i) = i; \quad M_c(i) = 2$

The OFG also gives all the data exchanges (the places in the OFG) between instances. Each physical memory Mem_i is bounded with its size mem_i. The place mapping function takes the list of places and returns the allocated list of physical memories.

The user specifies an SDF graph which is unfolded into its OFG, then specifies an architecture model with repeated components, and finally allocates the instances and places on the different components of the architecture. The allocation is potentially the result of an automated method. The following section explains how the whole system is captured into a set of CCSL constraints and enriched with real time constraints. Later, the state space representing all the conforming schedules is explored.

4 ENCODING DATA AND CONTROL FLOW CONSTRAINTS IN CCSL

The Clock Constraint Specification Language (CCSL) (André, 2009) is a declarative language to build specifications of systems by accumulation of constraints that progressively refine what can be expected from the system under consideration. The specification can be used and analyzed with our tool TimeSquare (Deantoni and Mallet, 2012). CCSL mainly targets embedded systems and was then designed to capture constraints imposed by the applicative part, the execution platform or also external requirements from the users, like non-functional properties. Constraints from the application and the execution platform are bound together through allocation constraints also expressed in CCSL. The central concept in CCSL are the logical clocks, which have been successfully used for their multiform nature by synchronous languages to build circuits and control-oriented systems, to design avionic systems with data-flow descriptions or design polychronous control systems (Benveniste et al., 2003)). They have also been used outside the synchronous community to capture partial orderings between components in distributed systems (Lamport, 1974). We promote their use here both for capturing the concurrency inherent to the application, the parallelism offered by the execution platform and synchronization constraints induced by the allocation.

Logical clocks are used to represent noticeable events of the system, *e.g.,* starting/finishing the execution of an agent, writing/reading a data from a place/memory, acquiring/releasing a resource; Their ticks are the successive (totally ordered) occurrences of the events.

In CCSL, the expected behavior of the system is described by a specification that constrains the way the clocks can tick. Basically, a CCSL specification prevents clocks from ticking when some conditions hold.

A CCSL specification denotes a set of schedules. If empty, there is no solution, the specification is invalid. If there are many possible schedules, it leaves some freedom to make some choices depending on additional criteria. For instance, some may want to run everything as soon as possible (ASAP), others may want to optimize the usage of resources (processors/memory/bandwidth).

Given a clock c, a *step* $s \in \mathbb{N}$ and a schedule σ, $c \in \sigma(s)$ means that clock c ticks at step s for this particular schedule.A schedule σ satisfies a specification if it satisfies all of its constraints.

Note that there are usually an infinite number of schedules that satisfy a specification, we only consider the ones that do not have empty steps.

4.1 Library of CCSL Constraints

New CCSL constraints can be defined from kernel ones (see (André, 2009)) in dedicated libraries. Before presenting newly-defined constraints, we introduce here some of the kernel constraints needed. Some constraints are stateless, *i.e.,* the constraint imposed on a schedule is identical at all steps; others are stateful, *i.e.,* they depend on what has happened in previous steps.

Two examples of simple stateless CCSL constraints are Union ($u \triangleq a \boxed{+} b$), where u ticks when either of a or b tick, and Exclusion ($a \boxed{\#} b$), which prevents a and b from ticking simultaneously. Note that Union is commutative and associative, we use in next sections an n-ary extension of this binary definition.

For stateful constraints, we use the history of clocks for a specific schedule, i.e., the number of times each clock has ticked at a given step.

A simple example of a primitive stateful CCSL clock constraint is Causality ($a \boxed{\preccurlyeq} b$). When an event causes another one, the effect cannot occur if the cause has not, i.e., the cause must be at least as frequent as its effect.

A small extension of Causality includes a notion of temporality and is called Precedence $a \boxed{\prec} b$, which means that the effect cannot occur simultaneously with its cause.

A bounded version of Precedence forces the effect to occur within n steps after its cause ($a \boxed{\prec_n} b$). Another example of a stateful constraint used in this paper is the DelayFor constraint. Such constraint delay a 'base' clock by counting the ticks of a 'reference' clock. A delayed clock $res \triangleq base \ \$ \ N \ on \ ref$ ticks simultaneously with *base* but with a delay of N steps of ref.

4.2 Encoding Occurrence Flow Graphs in CCSL

In our proposal, the occurrence flow graph produced in Section 3.2 is encoded in CCSL to represent the acceptable schedules with respect to data dependencies. It is further refined with additional CCSL constraints to take into account the allocation and the characteristics of the resources. The resulting specification gives the opportunity to explore the possible schedules according to an explicit representation of the constraints from the platform or performance requirements. Based on this representation it is still possible to apply (ad-hoc and/or efficient) analysis or synthesis tools as the ones already developed in the literature.

Let us consider first the agent instances. Each given agent $A$is unfolded into $X(a)$ instances $(a_i)_{i\in\{1..X(a)\}}$ (see Section 3). For each instance a_i, we associate two clocks, a_i^s that denotes the start of the execution of this instance and a_i^e that denotes the execution end. During the start of the instance, at least one of the input ports is synchronously read. The other ones are either synchronously read or sequentially read (to allow further concurrency limitation imposed by the allocation on the hardware platform). In the same spirit, at the end of the instance execution at least the last output port is synchronously written. The execution cannot end before it starts, it is non re-entrant but it can be instantaneous. This is denoted in CCSL by $\forall i \in \{1..X(a)\}, a_i^s \boxed{\preccurlyeq_1} a_i^e$. Also, different instances of a same agent denote successive occurrences and are thus causally dependent. In CCSL, it becomes $\forall i \in \{1..X(a)-1\}, a_i^s \boxed{\preccurlyeq} a_{i+1}^s$ and $a_1^s \boxed{\preccurlyeq_1} a_{X(a)}^s$.

The places of the occurrence flow graph can also be captured in CCSL. This can be done with kernel CCSL operators but here we use the Precedence constraint previously introduced. This encoding is not safe (not bounded) but will be bounded by the capacity of the memory after allocation. Let us consider the place p such that it connects the instance a_i and the instance b_j with an initial number of tokens $M(p)$. Such a place is encoded by the constraint $a_i^e \boxed{M(p) \prec} b_j^s$.

4.3 Introducing Allocation Constraints

Allocating Instances on Processors. Let use consider a processor P with a non-preemptive scheduler, instances allocated on P can not be executed concurrently. We must consequently capture the acquisition and release of the resource. This is done with two clocks P_{acq} and P_{rel}. We consider for P that only one instance can be executed at a time, this is captured in CCSL by $P_{rel} \boxed{\prec_1} P_{acq}$. Then the resource is acquired when one of the allocated instance starts its execution and released any time an executing instance finishes its execution. This is captured in CCSL with a union constraint: $P_{acq} \triangleq a_i^s \boxed{+} b_j^s \boxed{+} \ldots$ for all the instances allocated on this resource (here only a_i and b_j). Similarly for releasing the resource, $P_{rel} \triangleq a_i^e \boxed{+} b_j^e \boxed{+} \ldots$ Additionally, one must forbid the simultaneous acquisition of a single resource by two concurrent instances. This is done by adding exclusion constraints, pairwise, on each start of allocated instance and each end. In our example this means $(a_i^s \boxed{\#} b_j^s) \wedge (a_i^e \boxed{\#} b_j^e)$. Finally, the allocation of an agent instance on a specific processor gives the information about the execution time (let say a_{ET} for the instances of agent instances a) of the associated code. This is also captured by a constraint representing that the end of an agent instance is equal to (*i.e.,* synchronous with) the start of the agent delayed for an execution time computed according to execution cycle of the processor: $a_i^e = a_i^s \ \$\ a_{ET}\ \mathit{on}\ P_{exec}$. These constraint allow restricting the concurrency of the application according to the parallelism provided by the platform with respect to a specific allocation.

Allocating Places on Memories. The allocation of the places on a memory is encoded by using a constraint similar to the Precedence constraint. In our case, we want a memory with $m0$ data at the start of the system and a capacity $cap \in \mathbb{N}$, in which we can write several tokens nW in a single write and read several tokens nR in a single read. As usual, readings (resp. writing) are captured with a logical clock r (resp. w). This definition has been written in CCSL but the formal definition is not given here due to space limitations.

This memory is such that, reading is never allowed if there are not at least nR tokens available, considering the initial number of tokens ($m0$) and all those that were written and read ($\delta(n)$). Simultaneous read and write are possible if the memory capacity is not reached and considering that tokens are read before new tokens are written (causally in the same logical instant).

5 CONCLUSION

We provided a framework in which architectural constraints of various sorts can be translated into extra constraints, to be applied onto a SDF application model that should be mapped to this architecture (so that solving the constraints indeed guarantees the existence of a mapping). We also argued that SDF descriptions should, to some extend, be expanded so that mapping can be applied to *occurrences* of tasks, different instances being then mapped differently. The range of further transformations applicable to original process network models to ease (and extend the range of admissible) mappings could further be studied.

Our approach could be compared and contrasted to other schedulability techniques. Most consider a very abstract description of architecture (identical multiprocessors for instance), and add real-time scheduling requirements on the application side instead (periodicity, deadlines…). Then for each fixed choice of a class of constraints a given adhoc scheduling algorithm is established as optimal (Rate Monotonic, Earliest Deadline First). Instead, we choose to provide a constraint language powerful as CCSL to express a broad variety of constraints, and to let a general method (reachability analysis and model-checking basically) search for a candidate "best" schedule. This does not avoid the usual *NP-completeness* syndroma hidden behind many scheduling approaches, but works relatively well in practice due to symbolic representation techniques. Schedulability analysis by exhaustive model-checking has been attempted elsewhere (Amnell et al., 2004; Sun et al., 2014), but with assumptions quite different from ours in CCSL.

REFERENCES

Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., and Yi, W. (2004). Times: A tool for schedulability analysis and code generation of real-time systems. In *Formal Modeling and Analysis of Timed Systems*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer Berlin Heidelberg.

André, C. (2009). Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA.

Bamakhrama, M. and Stefanov, T. (2011). Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *ACM Int. Conf. on Embedded software*, pages 195–204. ACM.

Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and de Simone, R. (2003). The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83.

Bilsen, G., Engels, M., Lauwereins, R., and Peperstraete, J. (1995). Cyclo-static data flow. In *Int. Conf. on Acoustics, Speech, and Signal Processing, ICASSP'95*, volume 5, pages 3255–3258.

Buck, J. T. (1993). *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California, Berkeley, CA 94720.

Commoner, F., Holt, A. W., Even, S., and Pnueli, A. (1971). Marked directed graph. *Journal of Computer and System Sciences*, 5:511–523.

de Groote, R., Hölzenspies, P. K. F., Kuper, J., and Broersma, H. (2013). Back to basics: Homogeneous representations of multi-rate synchronous dataflow graphs. In *MEMOCODE*, pages 35–46. IEEE.

Deantoni, J. and Mallet, F. (2012). TimeSquare: Treat your Models with Logical Time. In Carlo A. Furia, S. N., editor, *TOOLS*, volume 7304 of *LNCS*, pages 34–41. Springer.

Goossens, K. and Hansson, A. (2010). The aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *Design Automation Conference (DAC'10)*, pages 306–311. ACM/IEEE.

Grandpierre, T., Lavarenne, C., and Sorel, Y. (1999). Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Int. W. on Hardware/Software Co-Design, CODES'99*, Rome, Italy.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In *Inform. Process. 74: Proc. IFIP Congr. 74*, pages 471–475.

Kalray (2014). Mppa manycore. http://www.kalray.eu/products/mppa-manycore.

Karczmarek, M., Thies, W., and Amarasinghe, S. (2003). Phased scheduling of stream programs. *ACM SIGPLAN Notices*, 38(7):103–112.

Karp, R. M., Miller, R. E., and Winograd, S. (1967). The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590.

Lamport, L. (1974). The parallel execution of do loops. *Commun. ACM*, 17(2):83–93.

Lee, E. A. and Messerschmitt, D. G. (1987a). Static scheduling of synchronous data flow programs for digital signal processing. *IEEE transactions on computers*, C-36(1):24–35.

Lee, E. A. and Messerschmitt, D. G. (1987b). Synchronous data flow. *Proceeding of the IEEE*, 75(9):1235–1245.

Mallet, F. and de Simone, R. (2015). Correctness issues on MARTE/CCSL constraints. *Sci. Comput. Program.*, 106:78–92.

Melpignano, D., Benini, L., Flamand, E., Jego, B., Lepley, T., Haugou, G., Clermidy, F., and Dutoit, D. (2012). Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications. In *DAC'12*, pages 1137–1142.

Milner, R. (1982). *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Petri, C. A. (1962). *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2. Technical Report RADC-TR-65–377, Vol.1, 1966, English translation.

Sriram, S. and Bhattacharyya, S. S. (2012). *Embedded multiprocessors: Scheduling and synchronization*. CRC press.

Stuijk, S. (2007). *Predictable Mapping of Streaming Applications on Multiprocessors*. PhD thesis, Faculty of Electrical Engineering, Eindhoven University of Technology, The Netherlands.

Stuijk, S., Geilen, M., and Basten, T. (2006). Sdf3: Sdf for free. In *ACSD*, volume 6, pages 276–278.

Sun, Y., Soulat, R., Lipari, G., Andr, ., and Fribourg, L. (2014). Parametric schedulability analysis of fixed priority real-time distributed systems. In *FTSCS*, volume 419 of *Communications in Computer and Information Science*, pages 212–228. Springer.

Yu, H., Talpin, J.-P., Besnard, L., Gautier, T., Marchand, H., and Guernic, P. L. (2011). Polychronous controller synthesis from Marte CCSL timing specifications. In *MEMOCODE*, pages 21–30. IEEE.