Explicit Control of Dataflow Graphs with MARTE/CCSL

Jean-Vivien Millo, Emilien Kofman, Julien Deantoni, Frédéric Mallet, Amine Oueslati, Robert de Simone

Explicit Control of Dataflow Graphs: a Requirement for Application Architecture Adequation

Jean-Vivien Millo
Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia-Antipolis France
jvmillo@unice.fr

Émilien Kofman
Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia-Antipolis France
emilien.kofman@unice.fr

Julien Deantoni
Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia-Antipolis France
julien.deantoni@unice.fr

Frédéric Mallet
Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia-Antipolis France
fmallet@unice.fr

Amin Oueslati
INRIA Sophia-Antipolis, 06900 Sophia-Antipolis France
amin.oueslati@unice.fr

Robert de Simone
INRIA Sophia-Antipolis, 06900 Sophia-Antipolis France
robert.de_simone@inria.fr

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. Lastly, the system can be checked for the existence of a schedule satisfying all the constraints using a state space exploration technique.

The approach is validated on a typical embedded system application allocated on a multi-core platform.

EMSOFT 2014 New Delhi, India

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION - BASED SYSTEMS**]: Real-time and embedded systems; F.118 [**COMPUTATION BY ABSTRACT DEVICES**]: Models of Computation

General Terms

Theory, Design, Verification

Keywords

Dataflow, Platform based design, Scheduling

1. INTRODUCTION

Synchronous Data Flow (SDF) [29] graphs are a popular model of choice to support Platform-Based Design approaches (also called Y-Chart flow, Application Architecture Adequation (AAA) methodology, or Model-Based optimized mapping, according to various authors). There are obvious reasons for that: SDF makes explicit the (in)dependences of data-flow 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.

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 [40, 5, 24, 20, 4] 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*)* 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 [44, 42, 45].

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 validates our approaches on a case study and Section 6 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 [21], Commoner/Holt's marked graphs [9], or even Karp et *al.* systems of uniform recurrence equations [25].

There was a renewal of interest in the 1980's for the class of so-called *Dataflow Process Networks (DFPNs)*, starting with SDF[28] and successive variants (boolean[8], cycloStatic[7], predicated dataflow graphs[38, 15]), first integrated inside the Ptolemy environment at UC Berkeley around Edward Lee, then spreading into various academic[34, 40, 19] and commercial contexts[26, 10].

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 [40] provides *SDFG-based MP-SoC design flow* [39] 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).

The Daedalus design flow [5] performs scheduling of CSDF graphs exclusively with time constraints. The architecture is modelled with another specific formalism and the schedule is determined with a combination of the architecture and the analysed CSDF graph.

Streamit experiments different scheduling policies [24] 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 [20] 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 toolchain [4] allows 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 [24], SDF3 with COMPSoC/ Aelite [18], sigmaC with Kalray/ MPPA [2], and PEDF with SThorM [31].

Other works [23] focus on a single step of the overall process. For example, K-Passa[3] [32] computes a schedule in order to maximize throughput while keeping the buffers as small as possible. K-Passa does not consider the constraints introduced when mapping to hardware. ArrayOL [16] focuses on multidimensional DFPN and AADL [37] provides modelling support for the design and analysis of embedded systems.

The TIMES tool [2] 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 con-

[1]Similarly to the transformation of SDF in HSDF

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

[3]www-sop.inria.fr/members/Jean-Vivien.Millo/kpassa

trary, 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.

The automatic parallel compilation community frequently considers a restricted class of programs called nested loops with affine bounds [1, 14]. Treatment of data is finer than in DFPNs, and so combines to a larger extent data parallelism with task parallelism and pipelining. Analysis techniques usually rely on polyhedral geometric representations of data space, which allow to deal with similar issues than in DFPNs. The issue of efficient mapping onto an existing parallel architecture has also been considered [43].

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 is threefold: 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 [29] (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.

Definition 1. *An SDF graph is a structure* $G = \langle N, P, F, W, M_0 \rangle$ *where*

- N *is a set of agents.*
- P *is a set of places.* $N \cap P = \emptyset$.
- $F \subseteq (N \times P) \cup (P \times N)$ *is a set of arcs. If* $n \in N$ *and* $p \in P$, (n,p) *and* (p,n) *are two arcs from* n *to* p *and from* p *to* n.
- $W : F \to \mathbb{N}^*$ *associates a width with each arc.*
- $M : P \to \mathbb{N}$ *gives the number of tokens in each place,* i.e., *the marking.* M_0 *is its initial marking.*
- *Each place has exactly one incoming and one outgoing arc:* $\forall p \in P$, $|\{(n,p) \mid \forall n \in N\}| = |\{(p,n) \mid \forall n \in N\}| = 1$. *The tuple arc/place/arc* $\langle (n,p), p, (p,n') \rangle$ *is called a (data) flow.*

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 [36] or monotonic in the sense of Kahn Process Network [21] or confluent in the sense of CCS [33].

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. Formally, agent a is fireable iff $\forall p \in {}^\bullet a$, $M(p) \geq W((p,a))$.

When an agent is fired, it consumes tokens in every input place and produces tokens in every output places. Formally, when a is fired, $\forall p \in {}^\bullet a$, $M(p) = M(p) - W((p,a))$ and $\forall p' \in a^\bullet$, $M(p') = M(p') + W((a,p'))$.

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[4]. 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.

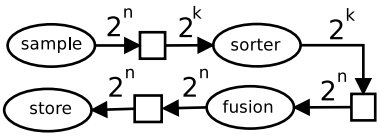

Figure 1: An SDF graph of a parallel sorting algorithm

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 [29], 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 entry 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

[4]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

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 [9]) [11]. 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→place→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).

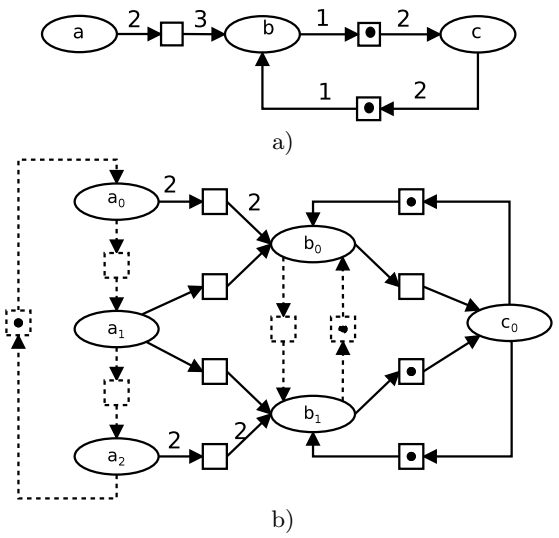

Figure 2: An SDF graph and its corresponding OFG

Converting an SDF graph into OFG.

The following algorithm converts an SDF graph into its OFG.

Algorithm 1: Converting an SDF graph into an OFG

Input: A balanced SDF Graph G
Output: A occurrence flow graph ofg
ofg=convertToHSDF(G)
for all *place* of ofg **do**
  $sibling = \{place' | {}^\bullet place = {}^\bullet place'$ and $place^\bullet = place'^\bullet$ and $place' \neq place\}$
  $ofg.removeAll(sibling)$
  $W({}^\bullet place, place) = W(place, place^\bullet) = |sibling| + 1$
end for
$addCausesBtwInstOfSameAgent(cfg)$
return cfg

The function ($convertToHSDF()$) to convert an SDF graph into an Homogeneous SDF (multi)-graph is given in [38] (p.45). Note that the major drawback of HSDF is the explosion of the number of arcs. Our actual implementation of Algorithm 1 goes directly from SDF to CFG by rehashing the transformation while avoiding the explosion. The function $addCausesBtwInstOfSameAgent()$ consists in adding flows following the first rule ($\forall$ agent a, a_i causes $a_{i+1 mod X(a)}$).

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 [22] 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:

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

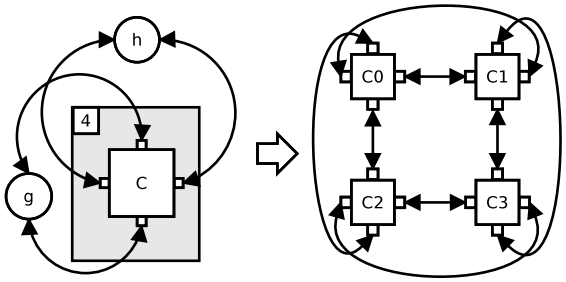

Figure 3: A torus expressed in a parametric way

If two components are not repeated then they can be connected directly. If they belong to the same pattern the components with same indices are connected. If a repeated component is connected to a non repeated component then all the instances of the repeated component are connected to the non-repeated one.

For the experiments, we considered a generic multi-core with one level of cache and a simple repeated pattern 4. We consider however that 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 is).

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 2. *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.*

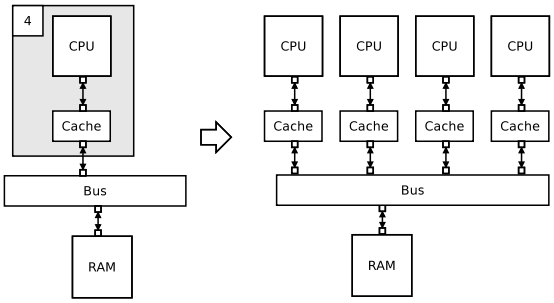

Figure 4: A generic multi-core architecture

$M = M_a(i), M_b(i), ...$ *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:
$M_A = (M_a, M_b, M_c)$
$M_a(i) = i \mod 2 == 1; \qquad M_b(i) = i; \qquad M_c(i) = 2$

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) [3] 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 [12]. 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 [6]). They have also been used outside the synchronous community to capture partial orderings between components in distributed systems [27]. 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.

DEFINITION 3 (LOGICAL CLOCK). *A* logical clock *c is defined as an infinite sequence of ticks:* $(c_n)_{n=1}^{\infty}$.

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.

DEFINITION 4 (CCSL SPECIFICATION). *A* CCSL *specification is a tuple* $Spec = \langle C, Cons \rangle$*, where* C *is a finite set of clocks and* $Cons$ *is a finite set of constraints.*

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).

DEFINITION 5 (SCHEDULE). *A schedule* σ *over set of clocks* C *is a sequence of ticking clocks.* $\sigma : \mathbb{N} \to 2^C$.

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.

DEFINITION 6 (SATISFACTION). *A schedule* σ *satisfies a specification* ($\sigma \models Spec$) *if it satisfies all of its constraints* ($\forall cons \in Cons, \sigma \models cons$).

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: $\forall n \in \mathbb{N}_{>0}, \sigma(n) \neq \emptyset$.

4.1 Library of CCSL constraints

New CCSL constraints can be defined from kernel ones (see [3]) 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 and Exclusion.

DEFINITION 7 (UNION). *Let* a, b *be two logical clocks. A schedule* σ *satisfies the union constraint on* a *and* b *if the following condition holds:* $\sigma \models u \triangleq a \boxed{+} b \iff \big(\forall n \in \mathbb{N}, u \in \sigma(n) \iff (a \in \sigma(n) \vee b \in \sigma(n))\big)$

Note that Union is commutative and associative, we use in next sections an n-ary extension of this binary definition.

DEFINITION 8 (EXCLUSION). *Let* a, b *be two logical clocks. A schedule* σ *satisfies the exclusion constraint on* a *and* b *if the following condition holds:* $\sigma \models a \boxed{\#} b \iff \big(\forall n \in \mathbb{N}, (a \notin \sigma(n) \vee b \notin \sigma(n))\big)$

For stateful constraints, we use the history of clocks for a specific schedule.

DEFINITION 9 (HISTORY). *Given a schedule* σ*, the* history *over a set of clocks* C *is a function* $H_\sigma : C \times \mathbb{N} \to \mathbb{N}$ *defined inductively as follows for all clocks* $c \in C$*:*

$H_\sigma(c, 0) = 0$

$\forall n \in \mathbb{N}, c \notin \sigma(n) \implies H_\sigma(c, n+1) = H_\sigma(c, n)$

$\forall n \in \mathbb{N}, c \in \sigma(n) \implies H_\sigma(c, n+1) = H_\sigma(c, n) + 1$

A simple example of a primitive stateful CCSL clock constraint is Causality. When an event causes another one, the effect cannot occur if the cause has not. In CCSL, causality can be instantaneous.

DEFINITION 10 (CAUSALITY). *Let* a, b *be two logical clocks. A schedule* σ *satisfies the causality constraint on* a *and* b *if the following condition holds:* $\sigma \models a \boxed{\preccurlyeq} b \iff \big(\forall n \in \mathbb{N}, H_\sigma(a, n) \geq H_\sigma(b, n)\big)$

A small extension of Causality includes a notion of temporality and is called Precedence.

DEFINITION 11 (PRECEDENCE). *Let* a, b *be two logical clocks and* $\delta \in \mathbb{Z}$*. A schedule* σ *satisfies the precedence constraint on* a *and* b *if the following condition holds:* $\sigma \models a \boxed{\delta \prec} b \iff \big(\forall n \in \mathbb{N}, H_\sigma(a, n) - H_\sigma(b, n) = -\delta \implies b \notin \sigma(n)\big)$

The primitive CCSL precedence is defined as: $a \boxed{\prec} b \equiv a \boxed{0 \prec} b$. A bounded version of precedence is defined as $a \boxed{\prec_N} b \equiv a \boxed{\prec} b \wedge a \boxed{N \prec} b$.

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.

DEFINITION 12 (DELAYFOR). *Let* $base$, ref *and* res *be three logical clocks and* $N \in \mathbb{N}$*. A schedule* σ *satisfies constraint* DelayFor *if the following condition holds:*

$$\sigma \models res \triangleq base \ \$ \ N \ on \ ref \iff$$
$$\big(\exists n \in \mathbb{N}, res \in \sigma(n) \iff ref \in \sigma(n) \wedge$$
$$\exists m < n \in \mathbb{N}, base \in \sigma(m) \wedge H_\sigma(ref, n) - H_\sigma(ref, m) = N\big)$$

4.2 Encoding the occurrence flow graph 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. For a given agent A, the algorithm 1 tells us that it has to be unfolded into $X(a)$ instances $(a_i)_{i \in \{1..X(a)\}}$. 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: $a_i^e = a_i^s \ \$ \ a_{ET} \ on \ P_{exec}$. 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 for readability we present here directly the resulting semantics

DEFINITION 13 (MEMORY). *Let w, r be two logical clocks and $cap, nW, nR, m0 \in \mathbb{N}$.*
Let $\delta(n)$ be $nW * H_\sigma(w,n) - nR * H_\sigma(r,n)$. A schedule σ satisfies the memory constraint if the following condition holds:
$\sigma \models Memory\ (w, r, cap, nW, nR, m0) \iff \big(\forall n \in \mathbb{N},$
$(\delta(n) + m0 < nR \implies r \notin \sigma(n)) \wedge$
$(\delta(n) + m0 \leq cap - nW + nR) \implies (w \in \sigma(n) \implies r \in \sigma(n)) \wedge$
$(\delta(n) + m0 > cap - nW + nR \implies w \notin \sigma(n))\big)$

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).

Capturing real time constraints.

In addition to the platform constraints, real time constraints are often provided in the requirements of the system. Such constraints can for instance impose a maximum time between two specific points of the system (*e.g.,* end-to-end latency). CCSL can capture such constraints in a similar way as for the duration of agents. This is not presented here but some examples can be found for instance in [17].

4.4 Analysis

After the encoding of all the allocated model constraints, the CCSL specification is a symbolic representation of all the acceptable schedules of the system. This specification has the good property to capture, in an explicit and refinable way, all the constraints from the application, its allocation on the hardware platform together with the real time requirements. The ultimate goal is to analyze this specification to get 'the' adequate schedule according to user-defined criteria like performance, energy consumption. To do so, one can try to make a brute force exploration of all the schedules represented by the specification. In this case, one can check first that the number of acceptable schedules is bounded as proposed in [30]. If bounded, it is possible to create the state space of the specification. The state space is made up with the state of the system in terms of the difference between the history used in each constraint (like for instance in the Precedence definition). From two states of the system, the set of clocks that tick synchronously is then representing the transition. This is the analysis technique used in section 5. Of course constructing the state space can be costly and static analysis techniques for instance based on the clock graph (see [12]) could be developed. Another interesting kind of analysis consists in testing if a specific static schedule (a total order of execution and resource access) is acceptable w.r.t. the specification. In this case one must encode his schedule in CCSL, add it to the original specification (like in [13]) and run a simulation of the resulting specification in TimeSquare. If no deadlock is found then the schedule is acceptable, otherwise, it violates at least one of the constraints. Finally another possibility is to export the CCSL specification in an existing analysis model like in [44, 42, 45].

5. TOOL SUPPORT AND CASE STUDIES

5.1 Tool support

The tool *KPassa-AAA* implements the proposed approach. KPassa-AAA is an Eclipse[5] plugin using the EMF technology. The tool and additional information about the case studies is available online[6]. Figure 5 presents a toy example modelled in our framework.

The application part is the OFG generated automatically from an SDF specification. The architecture is also ex-

[5]http://www.eclipse.org
[6]http://www-sop.inria.fr/aoste/software/kpassa-aaa

panded automatically from a parametric one. The allocation is represented with the dotted lines across the frames.

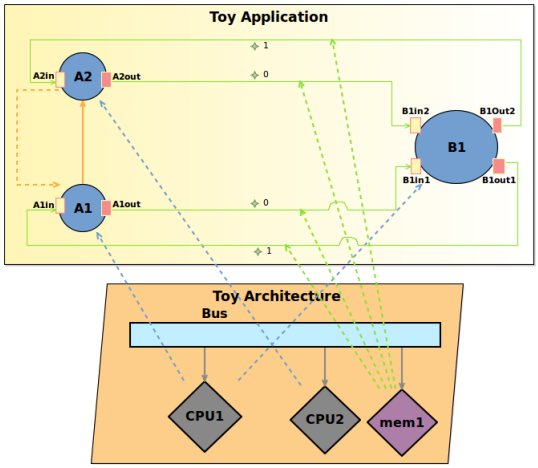

Figure 5: A screenshot of the framework

KPassa-AAA generates automatically the CCSL specification corresponding to the specified system. Note that the elements of the model have attributes (*e.g.,* preemptive or not-preemptive CPU). The generated CCSL specification varies according to the selected values.

5.2 Toy example

The first example is the one presented in Figure 5. Different allocations have been tested for this example. For each, Table 1 reports the size of the state space of the system in term of states, transitions, and elementary cycles. Every cycle corresponds to an admissible schedule. It contains exactly one period of execution. The first line reports the state space without allocation constraint. The lines two and three add CPU binding constraints and the line four and five add memory binding constraints (considering the CPU allocation of line 3).

Constraints	#Vertice	#Edge	#Cycles
None	53	110	N/A
One CPU	32	58	3859
A_1, B_1 on CPU_1 A_2 on CPU_2	32	71	43466
Memory size = 4	28	48	1034
Memory size = 3	22	33	189

Table 1: State space of the toy example: number of vertices, number of edges and number of cycles

The concurrency is progressively reduced. At each step, the model can be exported to an external ad-hoc, more efficient analysis tool but all the steps are made explicit.

5.3 Spectrum analyser

The second example is the simplified spectrum analyser algorithm presented in [35]. The SDF description of the spectrum analyser is presented in Figure 6-a. Note that the input and output weight of the *Adaptative Low-Pass* filter has been artificially changed to limit the concurrency to four agents; the target architecture is composed of four CPUs. The associated OFG is presented in Figure 6-b[7].

The target architecture is the one presented in Figure 4. The execution time of the filters is the same on every processor since they are instances of the same specification. The execution time is 2 for the filters *Peak Detector* and *Zoom Control* and 1 otherwise.

The allocation of the instances of the agents on non- preemptif CPUs is as follows: $M = \{M_{AdaptativeLP}, M_{Decision}\}$ with $M_{AdaptativeLP}(i) = CPU_i$, $M_{Decision}(i) = \text{if}(\ i < 2\ )CPU_1$ else CPU_3. Otherwise, the instances *FFT Zoom* and *Zoom Control* are allocated on CPU_0, *Peak Detector* and *Interpolar* are on CPU_2.

The allocation of the flows on memories is as follows: The flow from $Decision_i$ and $AdaptativeLP_i$ is mapped to $Scrat - chpad_i$ with $i \in \{1, 3\}$. The flow from *Zoom Control* to *FFT Zoom* is mapped to $Scratchpad_0$ and the flow from *Peak Detector* to *Interpolar* is mapped to $Scratchpad_2$. The twelve other flows are mapped to the *RAM*.

The specification is terminated with an end to end latency of 24. The unit of time is logical but is the same as the execution time given above.

The exhaustive state space exploration of this example is cumbersome: more than 4000 states and certainly billions of cycles. However, the resulting CCSL specification remains a ground to perform other analyzes as described in Section 4.4.

6. 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 ad-hoc 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 [2, 41], but with assumptions quite different from ours in CCSL.

7. REFERENCES

[7]The KPassa-AAA representation of this example can be found on the companion web page

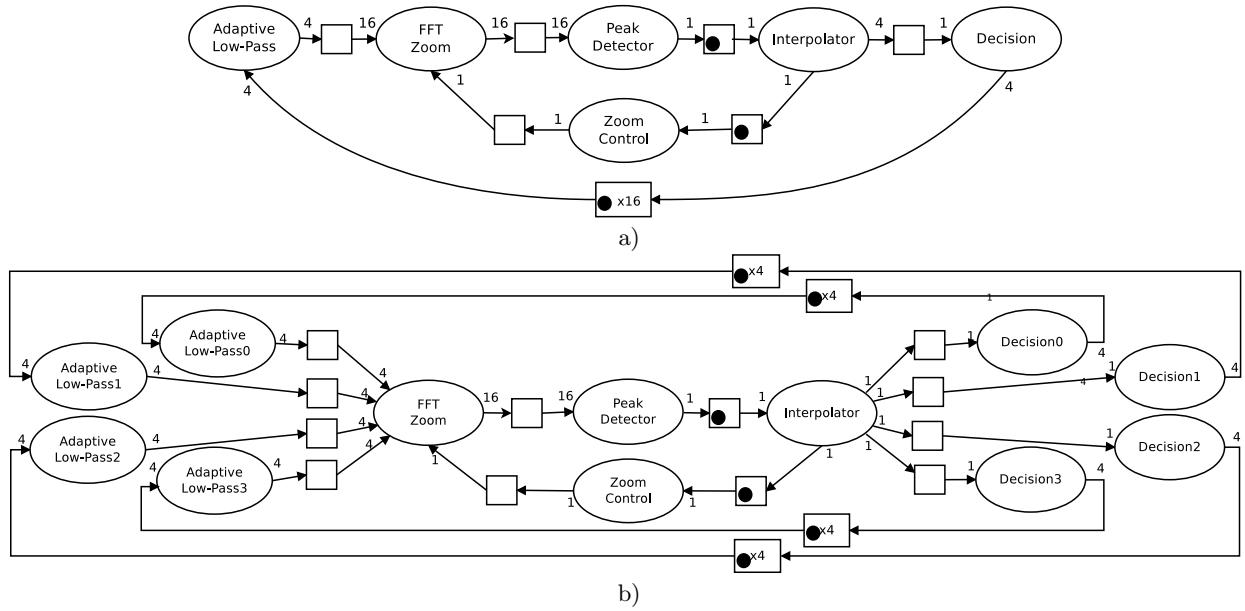

Figure 6: The SDF graph and the corresponding OFG of the spectrum analyser

[1] R. Allen and K. Kennedy. Automatic loop interchange (with retrospective). In *Best of PLDI*, pages 75–90, 1984.

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

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

[4] P. Aubry, P.-E. Beaucamps, F. Blanc, B. Bodin, S. Carpov, L. Cudennec, V. David, P. Dore, P. Dubrulle, B. D. de Dinechin, F. Galea, T. Goubier, M. Harrand, S. Jones, J.-D. Lesage, S. Louise, N. M. Chaisemartin, T. H. Nguyen, X. Raynaud, and R. Sirdey. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. *Procedia Computer Science*, 18(0):1624–1633, 2013. Int. Conf. on Computational Science.

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

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

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

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

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

[10] L. Cudennec and R. Sirdey. Parallelism reduction based on pattern substitution in dataflow oriented programming languages. *Procedia Computer Science*, 9(0):146–155, 2012. Proc. of the Int. Conf. on Computational Science, {ICCS'12}.

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

[12] J. Deantoni and F. Mallet. TimeSquare: Treat your Models with Logical Time. In S. N. Carlo A. Furia, editor, *TOOLS - 50th Int. Conf. on Objects, Models, Components, Patterns - 2012*, volume 7304 of *Lecture Notes in Computer Science - LNCS*, pages 34–41. Springer, May 2012.

[13] J. DeAntoni, F. Mallet, F. Thomas, G. Reydet, J.-P. Babau, C. Mraidha, L. Gauthier, L. Rioux, and N. Sordon. Rt-simex: Retro-analysis of execution traces. In *ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, FSE '10, pages 377–378, New York, NY, USA, 2010. ACM.

[14] P. Feautrier. Some efficient solutions to the affine scheduling problem. part i and ii. *Int. J. of Parallel Programming*, 21(5 and 6):313–347 and 389–420, 1992.

[15] P. Fradet, A. Girault, and P. Poplavkoy. Spdf: A schedulable parametric data-flow moc. In *DATE*, pages 769–774, 2012.

[16] C. Glitia, J. DeAntoni, F. Mallet, J.-V. Millo, P. Boulet, and A. Gamatié. Progressive and explicit refinement of scheduling for multidimensional data-flow applications using UML Marte. *Design Autom. for Emb. Sys.*, 16(2):137–169, 2012.

[17] A. Goknil, J. Deantoni, M.-A. Peraldi-Frati, and F. Mallet. Tool Support for the Analysis of TADL2 Timing Constraints using TimeSquare. In *ICECCS'2013 - 18th Int. Conf. on Engineering of Complex Computer Systems*, Singapore, Singapore, July 2013.

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

[19] M. I. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. PhD thesis, Massachusetts Institute of Technology, 2010.

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

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

[22] Kalray. Mppa manycore, 2014. http://www.kalray.eu/products/mppa-manycore.

[23] M. kamel Benhaoua, A. E. H. Benyamina, and P. Boulet. Heuristics for routing and spiral run-time task mapping in NoC-based heterogeneous MPSOCs. *CoRR*, abs/1312.5764, 2013.

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

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

[26] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: deriving process networks from matlab for embedded signal processing architectures. pages 13–17, 2000.

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

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

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

[30] F. Mallet, J.-V. Millo, and R. De Simone. Safe CCSL Specifications and Marked Graphs. In M. Roncken and J.-P. Talpin, editors, *MEMOCODE - 11th IEEE/ACM Int. Conf. on Formal Methods and Models for Codesign*, pages 157–166, Portland, United States, Oct. 2013. IEEE CS.

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

[32] J.-V. Millo and R. de Simone. Periodic scheduling of marked graphs using balanced binary words. *Theoretical Computer Science*, 458:113–130, November 2012.

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

[34] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(3):542–555, 2008.

[35] T. W. O'Neil, S. F. Khasawneh, M. E. Richter, and R. K. Pullaguntla. Transforming synchronous data-flow graphs to reduce execution time. *Int'l. Journal of Computers and Their Applications*, 18(2):111 – 122, 2011.

[36] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition:, New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Pages: Suppl. 1, English translation.

[37] SAE. *Architecture Analysis and Design Language (AADL)*, September 2012. AS5506b/1, http://www.sae.org.

[38] S. Sriram and S. S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. CRC press, 2012.

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

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

[41] Y. Sun, R. Soulat, G. Lipari, Ã. AndrÃl', and L. Fribourg. Parametric schedulability analysis of fixed priority real-time distributed systems. In C. Artho and P. C. ÃŰlveczky, editors, *Formal Techniques for Safety-Critical Systems*, volume 419 of *Communications in Computer and Information Science*, pages 212–228. Springer International Publishing, 2014.

[42] J. Suryadevara, C. Seceleanu, F. Mallet, and P. Pettersson. Verifying MARTE/CCSL mode behaviors using UPPAAL. In R. Hierons, M. Merayo, and M. Bravetti, editors, *Software Engineering and Formal Methods*, volume 8137 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2013.

[43] M. Wang and F. Bodin. Compiler-directed memory management for heterogeneous {MPSoCs}. *Journal of Systems Architecture*, 57(1):134–145, 2011. -Special Issue On-Chip Parallel And Network-Based Systems.

[44] L. Yin, F. Mallet, and J. Liu. Verification of MARTE/CCSL Time Requirements in Promela/SPIN. In *IEEE ICECCS 2011 - 16th IEEE Int. Conf. on Engineering of Complex Computer Systems*, Las Vegas, United States, Apr. 2011. IEEE.

[45] H. Yu, J.-P. Talpin, L. Besnard, T. Gautier, H. Marchand, and P. L. Guernic. Polychronous controller synthesis from marte ccsl timing specifications. In *MEMOCODE*, pages 21–30. IEEE, 2011.