

## Progressive and Explicit Refinement of Scheduling for Multidimensional Data-Flow Applications using UML MARTE

Calin Glitia · Julien DeAntoni · Frédéric  
Mallet · Jean-Vivien Millo · Pierre  
Boulet · Abdoulaye Gamatié

Received: date / Accepted: date

**Abstract** Process networks and data-flow graphs are used to capture data-dependencies in computation-intensive embedded systems. Their simplicity allows the computation of static schedules that reduce the dynamic overhead and increase predictability. The resulting schedule is a total ordering of actor computations and communications. It can therefore become an over-specification of the initial system when several schedules are valid. This is particularly the case for multidimensional data-flow applications.

We propose a methodology to avoid such an over-specification. We propose to use logical time to capture explicitly all the valid schedules for a given multi-dimensional data-flow model. Then, we show that the proposed approach allows for a progressive and explicit refinement of computation scheduling that also captures constraints imposed by the environment and the execution platform. All this is achieved by using UML MARTE concepts and the resulting models can be considered for simulation and analysis with existing tools for early design validation. The whole approach is validated on a typical application devoted to radar signal processing.

**Keywords** MARTE · clock constraint specification language · embedded system design · logical time · repetitive structure modeling · multidimensional data-flow

---

Calin Glitia, Julien Deantoni, Frédéric Mallet and Jean-Vivien Millo  
Université Nice Sophia Antipolis, F-06102, Nice, France  
INRIA Sophia Antipolis Méditerranée, F-06903, Sophia Antipolis, France  
I3S, UMR 6042 - UNS and CNRS, F-06902, Sophia Antipolis, France  
Tel.: 33 (0)4 92 38 79 66  
E-mail: {Julien.DeAntoni, Frederic.Mallet}@inria.fr

Pierre Boulet and Abdoulaye Gamatié  
LIFL, UMR 8022 - Univ. Lille 1 and CNRS, F-59650 Villeneuve d'Ascq, France  
INRIA Lille - Nord Europe, F-59650 Villeneuve d'Ascq, France  
Univ. Lille Nord de France, F-59650 Villeneuve d'Ascq, France  
Tel.: 33 (0)3 28 77 85 41  
E-mail: Pierre.Boulet@lifl.fr, Abdoulaye.Gamatie@lifl.fr

## 1 Introduction

Computation-intensive embedded systems require the parallel execution of data-intensive computations on several computation units. Process networks [21] and data-flow graphs [24, 23, 20] are adequate to capture the data dependencies of such systems and to compute static schedules that optimize specific criteria such as communication buffer size. Being able to compute static schedules also reduces the scheduling overhead on target platforms, increases predictability, and allows precise performance estimations. Such a schedule must satisfy both the data dependencies and the execution constraints imposed by application functionality, execution platforms and the environment.

**The execution ordering depends on constraints of different natures.** In general, data dependency constraints induce only a partial execution ordering on the way an application deterministically achieves its associated functionality. In other words, whatever constraints are added to those already induced by data dependencies, the application always computes the same function. As the execution ordering constraints implied by the data dependencies must be respected to ensure that the application computes the right function, we refer to the schedule computed from data dependency constraints as the minimal execution ordering. All valid execution orderings are more constrained (i.e. with less parallelism) than this one.

The target execution platform constrains a refinement of the partial schedule inherent to the application functionality. Typically, this application functionality is scheduled and executed according to the topology of its execution platform in terms of processors and memories. A multiprocessor platform with a Network-on-Chip for communications is usually more efficient for computation-intensive embedded systems than a uniprocessor platform using a bus. These two platforms are likely to induce different schedulings of a given application functionality and hence lead to different execution orderings.

Beyond the data dependencies inherent to application functionality and execution platforms, the environment can impact the application schedule. Typically, the availability, i.e. arrival ordering, of the inputs of an application from the environment via some sensors influences the scheduling of application tasks for an optimal execution. Similarly, an environment can impose some constraints on the outputs of an application, e.g. some production rate constraints on computed data to be sent to an actuator.

**Embedded system design concepts: MARTE profile.** The UML profile for Modeling and Analysis of Real-Time and Embedded systems, referred to as MARTE [25], has been recently adopted by the OMG. It extends the Unified Modeling Language (UML) [26] with concepts required to model embedded systems. The *General Component Modeling* (GCM) and *Repetitive Structure Modeling* (RSM) packages offer a support to capture the application functionality. GCM defines all basic concepts such as data flow ports, components and connectors. RSM provides concepts for expressing repetitive structures and regular topological connections. It is essential for the expression of parallelism, in both application modeling and execution platform modeling; and for the allocation

of applications onto execution platforms. In the context of application modeling, Array-OL with delays [20] is the underlying Model of Computation and Communication (MoCC) which expresses data dependencies independently of the scheduling. In [20] the authors argue that Array-OL with delays is well suited to model multidimensional signal processing applications and compare this MoCC to others, mainly derivatives of SDF. One of the interesting characteristics of Array-OL with delays is that it is more abstract than the derivatives of SDF. Indeed, it expresses data dependencies without any implied scheduling, thus leaving the choice of the schedule to later design stages. This could be viewed as a weak point of Array-OL with delays because an Array-OL with delays model is not executable but it is actually very useful in our opinion because it leaves all the options open to adapt the schedule to the environment and to the hardware architecture. Thus Array-OL with delays and MARTE RSM allow to express the minimal execution ordering whereas it is not generally the case with derivatives of SDF.

The *Hardware Resource Modeling* (HRM) package, which specializes the concepts of GCM into hardware devices such as processor, memory or buses allows the modeling of the execution platforms in MARTE. The *Allocation* (Alloc) package allows the modeling of the space-time allocation of application functionality on an execution platform. Both the HRM and Alloc packages can be used with the RSM package to allow a compact modeling of repetitive hardware (e.g. grids of processing elements) and data and computation distributions of a parallel application onto such repetitive hardware.

The models described with the previous packages can be refined with temporal properties specified within the *Time* package. Such properties are typically clock constraints denoting some activation rate constraints about considered components. The concepts of the Time package are often used with the *Clock Constraint Specification Language* (CCSL) [3], which was initially introduced as a non-normative annex of MARTE.

#### **Our contribution.**

Data dependencies are explicitly captured by typical data-flow models. They are often constrained via specific compiler choices to obtain a particular scheduling according to which their corresponding generated code is executed. These scheduling choices, left to a compiler, may not necessarily be well-adapted to scheduling requirements supported by any execution platform. As a result, this does not enable a very flexible choice of execution platforms for a given data dependency specification beyond compiler scheduling constraints. For a better exploitation of compilers in accordance with execution platform constraints, we believe that scheduling constraints should be made explicit by refining given data dependency specifications so that typical execution constraints coming from environment and platform can be clearly captured. So how can we model all possible schedules and allow a designer to select the most appropriate one with respect to the platform and environment induced constraints?

We propose a uniform framework, illustrated in Figure 1, based on MARTE for designing computation-intensive embedded systems with the repetitive

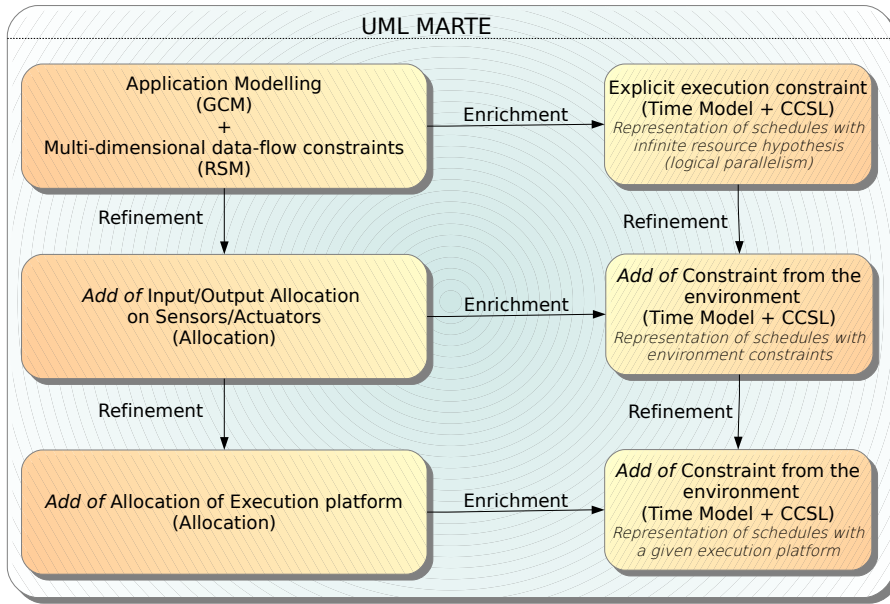


Fig. 1 Outline of the proposed approach

structure modeling concepts. The resulting descriptions are enriched with clock constraints information that explicitly capture information about environment and execution platform properties of systems. The ultimate goal of our framework is to permit an easy high-level design of computation-intensive embedded systems with a possibility to get access to simulation and analysis tools, such as Gaspard2 [14] and TimeSquare<sup>1</sup> for early design validation. The usage of the proposed framework is illustrated on a radar application that aims at detecting from an aircraft the objects moving on the ground. In our opinion, the overall work presented in this paper is one of the first design methodologies about MARTE, which addresses various features of embedded systems by coherently combining a large set of MARTE concepts.

**Outline.** In the following, Section 2 presents some related works. Section 3 gives some background on MARTE to model computation-intensive embedded systems. Section 4 describes our proposition for the use of MARTE to capture, data-dependencies and causal dependencies at application level, physical constraints on execution platforms and environments, and constraints induced by the allocation of applications onto execution platforms. Section 5 illustrates the approach on a moving target indication application used in radars. Finally, Section 6 gives concluding remarks.

<sup>1</sup> <http://timesquare.inria.fr/>

## 2 Related works

The Time package of MARTE promotes the use of logical time as a unifying notion of time to capture both causal dependencies and physical time relationships. Logical time was first introduced by Lamport to represent the execution of distributed systems [22]. It has been extended and used in distributed systems to check the communication and causality path correctness [11]. At the same period, Logical time has also been used in synchronous languages [4,5] for its polychronous and multiform nature, *i.e.*, based on several time references.

In the synchronous programming, the notion of logical clock has proved to be adaptable to any level of description, from very flexible causal time descriptions to very precise scheduling descriptions [7]. A simple and classical use of multiform logical time for specification is “*Task 1 executes twice as often as Task 2*”. The instant at which *Task 2* executes is taken as a time reference to specify the execution of *Task 1*. An event is then expressed relative to another one, that is used as a reference. No reference to physical time is given. Although, if the (physical) duration between two successive executions of *Task1* is given, the instants at which *Task2* must execute can be deduced. Relying only on the logical specification at early stages avoids over-specifications induced by using physical time references while the execution platform is not fully determined, yet. Actually, physical time is a particular case of logical time where a physical clock is taken as reference. In this paper, we show how MARTE can be used to capture explicitly both data-dependencies and execution ordering imposed by the execution platform or the environment. Pure data-dependencies are expressed using classical data-flow languages and the execution ordering is captured through a logical time specification in CCSL.

The Gaspard2 environment [14], which is dedicated to MPSoC provides tools to move from a high level MARTE description to implementation, through successive model transformations. The Gaspard2 modeling relies on MARTE. In particular, it uses the RSM, GCM, Alloc and HRM packages to model regular and massively parallel hardware architectures and applications. To generate automatically implementations from MARTE specifications, a model of a system is transformed into a new hierarchical representation where the top level repetition space represents the time. At the lower level of the hierarchy, the data dependencies are modified such that all the task instances of the system can execute at least once, *i.e.*, enough data are produced for all the system tasks. More recently, the authors proposed to use CCSL to enrich Gaspard2 models with activation rate relationships [13], specified as clock properties. These properties are therefore analyzed according to application mapping choices on a hardware platform, and processor frequencies [1]. This paper goes further by extracting those activation rates from the execution platform, environment and allocation constraints.

In a previous study [19], we have shown how logical time can be used to support for an explicit capture of all valid schedules for a given data-flow model. Our interest was focused on the semantics of Synchronous Data Flow (SDF) [24] and its multidimensional extension: Multidimensional-SDF [23], whereas the

idea of [19] was to replace some of the data-dependencies by logical time constraints. This paper instead considers the logical time as complementary of data dependencies. Indeed, logical clocks being mono-dimensional, the capture of multidimensional data dependencies through logical clocks is cumbersome. This paper aims at proposing to complement multidimensional data dependencies descriptions based on RSM with logical time constraints that capture the temporal dimension.

Beyond the above studies, we can also mention the hybrid modeling paradigms that consists in combining dataflow models and control-oriented features such as finite state machines. Such extensions have been widely studied for both monodimensional and multidimensional dataflow models [9]. Our approach shares a similar goal with these approaches by combining RSM to capture the dataflow aspects of an application while logical clocks of CCSL capture environment and platform constraints.

### 3 Modeling Multidimensional Data-Flow Applications and their Schedule with MARTE

We give here an overview of the Repetitive Structure Modeling and the Time model of the MARTE UML profile. We also discuss in a more precise way the issues addressed in this paper.

Before going into the description of these packages, let us recall briefly what is a profile. This extension mechanism of UML allows to specialize the concepts of UML. These concepts are called “meta-classes” in the UML specification (i.e. the UML “meta-model”) and they can be extended by a “stereotype” that precises their semantics. A stereotype is key-word that can have some parameters called “tagged-values” and that is used in a UML diagram to tag the concepts is applied to. Thus the MARTE UML profile defines lots of stereotypes organized in packages to add the ability to model and analyze real-time embedded systems to UML.

#### 3.1 Repetitive Structure Modeling

As detailed in [20], a MoCC dedicated to multi-dimensional signal processing applications (typical of multi-dimensional data-flow applications) has to allow several constructs:

- Multidimensional data arrays,
- Sub- and over-sampling,
- Data access as sliding windows,
- Cyclic access to these data arrays,
- Delays or some form of memorization.

The Array-OL with delays MoCC allows all these constructs with a limited number of concepts and a focus on a compact representation of the potential parallelism (both data and control parallelisms) of the application. The

control parallelism is expressed by graphs of components and the data parallelism is expressed by multi-dimensional repetitions similar to nested loops with uniform data-dependencies. Both forms of parallelism can be combined hierarchically to express complex applications. The main limitation of Array-OL with delays is that the only data structure available is multi-dimensional arrays of fixed size.

These concepts have been included in MARTE by the way of the General Component Modeling GCM package to model data-flow (UML only considers interaction by services between components and GCM adds the necessary stereotypes to enable data flows between components) and the Repetitive Structure Modeling (RSM) package to model multidimensional arrays and data-parallelism. As a generalization, the usage of the RSM package is not limited to model multi-dimensional data-flow applications but also the parallel hardware and the mapping of the application to the hardware.

RSM allows the definition of *repetitive structures* interconnected by regular connection patterns, via UML extensions focusing on two aspects:

1. the multiplicity (number of potential elements) of elements can be given as a multidimensional array, called *shape*;
2. the topology of complex regular interconnections (exactly which elements of a multidimensional array are linked to elements of another multidimensional array) is given in a compact way.

These very general concepts of shape and interconnection topology apply at three different levels:

1. **at the application level** the *shape* refers to multidimensional data structures and concurrent tasks that capture the potential (*logical*) parallelism within the application. The interconnection reflects the data dependencies between the tasks and the way the data are consumed or produced by those tasks;
2. **at the execution-platform level** these concepts describe the available (*physical*) parallelism: the shape describes the number of resources and the interconnection gives the way they communication links between those resources;
3. **during the allocation** regular patterns capture in a compact way the temporal scheduling and spatial mapping of application elements (*e.g.*, tasks) onto the hardware execution platform resources.

Before further describing these three aspects, we give a general description of the RSM elements.

A MARTE RSM description relies on UML *structured classifiers* and GCM to provide a hierarchical description of the internal structure of classes. It includes internal *parts*, *ports* and *connectors*. Parts can interact together when their ports are linked by a connector. They can also interact with the outside of the structured class when one of its ports is connected to a port of the structured class.

A UML *MultiplicityElement* gives a range (upper and lower value) to restrict the cardinality of a mono-dimensional set of Elements. Stereotype «Shaped» of RSM extends this metaclass to enable the specification of multidimensional arrays of model elements. Its *shape* property defines the number of dimensions and the size of each dimension as a comma-separated sequence of integers (e.g., {3, 2} is a bi-dimensional shape). Although stereotype «Shaped» can be applied to any element that extends *MultiplicityElement*, RSM recommends to restrict its use to parts and ports. This stereotype is used to specify the *repetition space* of a part or a *multidimensional array* associated with a port. When applied to a port, each point of a multidimensional array corresponds to one connection end point. To tailor precisely the connection topology, RSM proposes, via the abstract stereotype «LinkTopology», a mechanism to extend UML connectors and support the description of a regular connection pattern with a single connector. This pattern identifies as connection end points sub-arrays of points inside a multidimensional array. The considered patterns are multidimensional arrays themselves and are described by a shape.

### 3.1.1 The tiling concept

RSM allows the description of repetitive structures, *i.e.*, a UML structured classifier that contains the repetition of a specific part where each instance of the repeated part operates with identical sub-arrays (patterns) of the repetitive structure arrays. The shape associated with this part defines its repetition space.

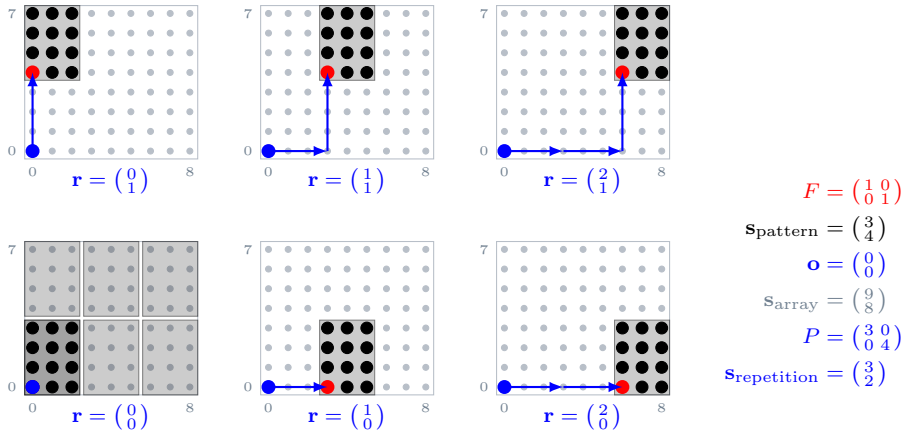
Within a repetitive structure, when ports of a repeated part are connected to ports of the repetitive structure, a *tiler* defines the relationship between an array and a collection of identical sub-arrays (one per instance of the repeated part). It is used to define the mathematical relation that specifies how the pattern paves the array for each instance of the repeated part. The tiling process is described by a tiler having the following properties:

- A *fitting* matrix  $F$ , which represents the distribution of the elements of one pattern within the array. A “fitted” pattern is called a *tile* (the column vectors represent the regular spacing between the elements of a pattern in the array relatively to a reference point).
- An *origin* vector  $\mathbf{o}$ , which represents the reference point of the *reference pattern* (for the *reference repetition*<sup>2</sup>).
- A *paving* matrix  $P$ , which describes how the tiles are used to pave the array (the column vectors represent the regular spacing between the reference points of the tiles relatively to the origin).

We can summarize the pattern construction with one formula. For a given coordinate  $\mathbf{r}$  inside a repetition space, *i.e.*,  $\mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{rep}}$  (the repetition space is unique for a repetition task and all its inputs and outputs) and a given

<sup>2</sup> The instance of the repeated part whose coordinate into the repetition space is 0 on each dimension.





**Fig. 2** A 2D pattern tiling exactly a 2D array

coordinate  $\mathbf{i}$ ,  $\mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}$  in the pattern, the corresponding element in the array has the coordinates:

$$\mathbf{o} + (P F) \cdot \begin{pmatrix} \mathbf{r} \\ \mathbf{i} \end{pmatrix} \bmod \mathbf{s}_{\text{array}}, \quad (1)$$

where  $\mathbf{s}_{\text{array}}$  is the shape of the array,  $\mathbf{s}_{\text{pattern}}$  is the shape of the pattern,  $\mathbf{s}_{\text{rep}}$  is the shape of the repetition space.

The tiling construction allows the specification of a large range of regular paving constructions, that varies from the simple tiling by blocks as illustrated in Figure 2, to complex paving constructions: non-parallel with the axes, strides, overlapping or modulo. The «Tiler» stereotype denotes a tiling operation. It specializes «LinkTopology» by connecting a part of the repetitive structure to a port of the repeated part.

### 3.1.2 Further useful concepts

UML connectors specify one-to-one links between multidimensional arrays with identical shapes. The stereotype «Reshape» is an extension that introduces uniform links between multidimensional arrays with different shapes. By tiling the arrays with an identical pattern, it links together the tiles of the source and destination arrays. The reshaped topology can be seen as a representation of a repetition with no behavior, it defines a repetition space and the shape of the pattern (the same for all connection ends). A tiler has to be associated with each connection end and it describes how the multidimensional shape associated with this end is tiled by the repetition of patterns of the reshape.

The stereotype «InterRepetition» denotes a connector that connects ports of the same repeated part and expresses links between uniformly-spaced instances of the repeated part. It allows modeling topologies in which instances

of the same repetition are interconnected, such as a grid or a cube topology. The shapes of the connected ports must be identical and property *repetition-ShapeDependence* defines the translation vector within the repetition space between linked instances. For non-modulo inter-repetition dependencies (property *modulo*), some translations are computed outside the repetition space. A connector «DefaultLink» connected to one end of the inter repetition topology defines a link whenever the translation of the inter-repetition is undefined. This allows the specification of default values.

### 3.1.3 RSM modeling of repetitive applications and execution platforms

The previous generic concepts of RSM have different interpretations according to their use in a specification of a repetitive application or a repetitive hardware architecture. The next paragraphs elaborate on these differences.

**Repetitive applications.** When modeling repetitive applications, UML classes abstract *tasks*. Tasks are application-related entities of parallelism seen as mathematical functions reading data on their input ports and writing data on their output ports. Ports are multidimensional data arrays characterized by their *shape* («Shaped») and *direction* («FlowPort»).

The difficulty and the variety of signal processing applications does not come from the elementary functions (often available as library functions), but from the way these functions access their input and output data as parts of multidimensional arrays. The repetitive constructions focus on expressing the inherent regularity of such applications. A class with no internal structure is seen as an elementary task (a black box) and it can be deployed on a library function for instance. The topological links define the data dependence graph, while the repetitive structures express how a single sub-task is repeated and how each instance of the repeated task operates with sub-arrays of the inputs and outputs. Making the repetitive structures independent and therefore parallel by construction (equivalent to the parallelization of nested-loops [10]) is key to express data parallelism. The repetitive application model captures the potential logical parallelism of an application.

Figure 3 shows the visual representation of a simple RSM repetitive structure for application modeling. It is an example of matrix multiplication. The port *A* reads an  $l \times m$  matrix<sup>3</sup> and the port *B* reads an  $m \times n$  matrix. The output is the matrix  $A \times B$  of size  $l \times n$ . The tiler after the port *A* slices the matrix in  $l$  vectors of size  $m$ . Each vector corresponds to a line of the matrix. The tiler after the port *B* slices the matrix in  $n$  vectors of size  $m$ . Each vector corresponds to a column of the matrix. One can see the differences on the fitting and paving vectors between these two tilers. The elementary task takes as inputs two vectors of size  $m$ . The elementary task has a repetition space of  $(l, n)$ <sup>4</sup> which means that it has to be repeated  $l \times n$  times to complete

<sup>3</sup> The *shape* of a port gives the size of the matrix going through the port.

<sup>4</sup> here again, the keyword *shape* is used. When it is attached to an elementary task, *shape* denotes the repetition space.

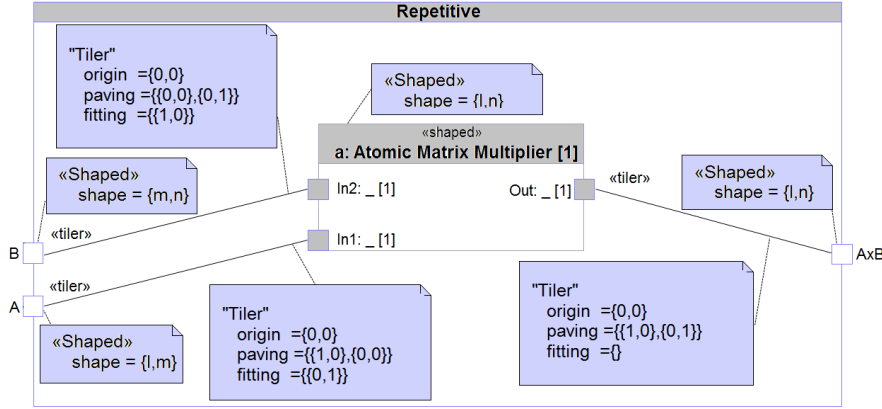


Fig. 3 An  $l \times m$ -matrix is multiplied to an  $m \times n$ -matrix and thus gives an  $l \times n$ -matrix.

the operation.  $l$  times on to the first dimension and  $n$  times on the second dimension. The elementary task computes the sum of the pairwise products of the input vectors values. Each execution of the elementary task produces one scalar value that has to be placed correctly in the output  $l \times n$ -matrix. The output tiler do so.

An RSM application conforms to the Array-OL with delays (Array Oriented Language) Model of Computation [20]. To make Array-OL with delays applications statically schedulable, rules are imposed on the specification, namely regarding the single assignment and the complete production of array elements. As a consequence of the single assignment formalism (no data element is ever written twice while it can be read several times), the spatial and temporal dimensions are treated equally in the arrays. In particular, time is spread over one (or several) dimension(s) of each of the arrays: *Each single assignment data structure defines the entire range of values that are conveyed on the associated port during execution, while the repetition spaces define the entire set of instances of the repeated task that will be instantiated at execution.*

An important point is that the projection of the multidimensional structures in time and space is separated from the functional specification. This allows building functional component libraries for reuse and gives a valuable support to carry out some architecture exploration with the least possible restrictions.

When employed for specifying applications, an inter-repetition dependence expresses uniform dependences between repeated instances of the same repeated part. Consequently, it introduces additional constraints and potentially restricts the completely parallel execution of independent instances of a repetition.

The authors showed that any repetition involved in a loop *fusion* transformation is split into two hierarchical repetitions where the inferior level of hier-

archy represents blocks (*i.e.*, identical subset) of the initial repetition space. A given inter-repetition dependence on the initial repetition is transformed into:

1. An inter-repetition dependence on the inferior hierarchy level if initial dependences are all between instances inside the same block.
2. An inter-repetition dependence on the superior hierarchy level if initial dependences are all between instances found in different blocks.
3. Two hierarchically connected inter-repetition dependences on the two hierarchy levels, defining respectively dependences inside the blocks and between different blocks.

When propagating the data-flow ordering, an inter-repetition dependence inside the blocks constrains the parallelism of the instances inside each block. A dependence between blocks introduces new scheduling constraints:

- If the dependence vector follows the direction of execution dictated by the data-flow ordering, the inter-repetition dependence just introduces new execution dependencies between the blocks, according to the dependence vector.
- If the dependence vector conflicts with the execution dictated by the data-flow ordering, the inter-repetition dependence must be moved inside the blocks, by increasing the block size through a paving transformation.

**Repetitive execution platforms.** When combined with the Hardware Resource Modeling (HRM) package of MARTE, the same RSM constructions can be used to model repetitive architectures. It is especially useful for architectures with a large number of identical components (like meshes of processors). Compact representations both simplify the modeling stage but also explicitly unify components having identical properties. Repetitive execution platforms capture the physical parallelism of the platform.

Stereotype «*distribute*» is proposed in RSM to specialize the MARTE allocation and is employed for designing, similar to the reshape mechanism, a repetitive allocation of multidimensional application structures (repetitions or data) onto repetitive architecture units. Repetitive Allocation Modeling with MARTE is discussed in [6]. The allocation of functional application elements onto the available resources (the execution platform) results in a spatial distribution and potentially reduces the logical parallelism defined in the application. In the context of RSM, «*distribute*» can be used to specify spatial projection of the multidimensional structures and partial time projection; *i.e.*, two data elements allocated to the same memory element cannot be read and written concurrently; however, their scheduling is not constrained but implicitly chosen at scheduling time.

### 3.1.4 Scheduling repetitive applications

An Array-OL with delays application expresses the maximum *logical parallelism* through the use of the repetitive structures and the UML hierarchical

decomposition. It describes the data dependencies between the elements of the arrays. As a direct consequence, it results a strict partial execution ordering between the tasks accessing these arrays. Any schedule that respects this ordering computes the same output values from the same inputs. The space-time mapping decision is separated from the functional specification and must respect the strict partial execution ordering imposed by the specification. It is a design intention that, by expressing the minimal execution ordering, lots of decisions can and have to be taken when mapping a MARTE RSM specification onto an execution platform.

The refactoring of the application for the space-time mapping boils down to giving a data-flow ordering (in time) for each data structure. Doing so further constrains the partial order defined by the data dependencies. For data-flow languages as the synchronous data-flow family, the ordering in time in which data are processed is inherent to the semantics of the language; a complete ordering for each data structure of SDF or a partial (by dimension) ordering for the data structure of MDSDF. When the data-flow ordering is not entirely defined within the specification, this choice is often done at compilation time and is encoded in the (offline) scheduler. Note that the scheduler usually also makes a choice on the ordering of the task instances (and not only on the data structures).

While this approach is very interesting at its first stage by exhibiting logical and physical parallelism, we found various drawbacks. First, the introduction of time information comes with a change in the structure of the application so that it can be difficult to apprehend its impact on the application. Second, the modification of the data dependencies over-constrains the system and in some cases drastically reduces the potential logical parallelism. Third, such an approach hides in the scheduling constraints the ordering of data imposed by the environment. Fourth, the information that specifies and explicits the links and constraints involved by: the space, the time, the environment and the platform are scattered (*i.e.*, almost lost) in the transformation, making it difficult to reason about potential optimizations of the allocation.

From the previous paragraphs, one can understand the need for making explicit scheduling choices in system specifications defined with RSM. Such a refinement can be done by considering additional scheduling constraints (from environments and execution platforms) expressed with *clocks* as described in the MARTE time sub-profile.

### 3.2 Time in MARTE

In MARTE, a *clock* is a totally ordered set of *instants*. A *time structure* is a set of clocks  $C$  and a set of relations on instants.  $I$  denotes the union of all instants of all clocks within a given time structure. We consider two kinds of relations: *causal* and *temporal* ones. The basic causal relation over  $I$  is causality/dependency,  $i \in I, j \in I, i \preceq j$  means  $i$  causes  $j$  or  $j$  depends on  $i$ , *i.e.*, if  $j$  occurs then  $i$  also occurs. The three basic temporal relations over

$I$  are *precedence* ( $\prec$ ), *coincidence* ( $\equiv$ ), and *exclusion* ( $\#$ ). For any instants  $i$  and  $j$  in a time structure,  $i \prec j$  means that the only acceptable execution traces are those where  $i$  occurs strictly before (precedes)  $j$ .  $i \equiv j$  imposes instants  $i$  and  $j$  to be coincident, *i.e.*, they must always occur at the same execution step, both of them or none of them.  $i \# j$  forbids the coincidence of the two instants, *i.e.*, they cannot occur at the same execution step. Note that, some consistency rules must be enforced between causal and temporal relations.  $i \prec j$  can be refined either as  $i \prec j$  or  $i \equiv j$ , but  $j$  can never precede  $i$ . Furthermore, we do not assume a global notion of time. Temporality is given by the *precedence* binary relation, which is partial, asymmetric (*i.e.*, antisymmetric and irreflexive) and transitive. The *coincidence* binary relation is an equivalence relation on instants, *i.e.*, reflexive, symmetric and transitive.

In this paper, we consider discrete sets of instants only, so that the instants of a clock can be indexed by natural numbers. For a clock  $c$ ,  $c[k]$  denotes its  $k^{\text{th}}$  instant.

Specifying a full time structure using only instant relations is not realistic since clocks are usually infinite sets of instants. Thus, an enumerative specification of instant relations is forbidden. The Clock Constraint Specification Language (CCSL) defines a set of time patterns between clocks that apply to infinitely many instant relations. As an example, consider clock relation *precedence* (denoted  $\prec$ ).  $c_1 \prec c_2$ , read ‘ $c_1$  precedes  $c_2$ ’, specifies that the  $k^{\text{th}}$  instant of clock  $c_1$  *precedes* the  $k^{\text{th}}$  instant of clock  $c_2$ , for all  $k$ . More formally:  $c_1 \prec c_2$  means  $\forall k \in \mathbb{N}^*, c_1[k] \prec c_2[k]$ .  $(c_1 \text{ by } m) \prec (c_2 \text{ by } n)$  is a grouping extension that means  $\forall k \in \mathbb{N}^*, c_1[m*k] \prec c_2[(n-1)*k+1]$ . Similarly,  $c_1 \sqsubset c_2$  ( $c_1$  is a sub clock of  $c_2$ ) means that for all  $k$ , the instant  $c_1[k]$  of  $c_1$  coincides with exactly one instant of  $c_2$ . More formally:  $c_1 \sqsubset c_2$  means  $\forall k \in \mathbb{N}^*, \exists n \in \mathbb{N}^* \text{ s.t. } c_1[k] \equiv c_2[n]$ . The relation  $\sqsubset$  is order-preserving. All the coincidence-based relations are based on *isSubclockOf*. When both  $c_1 \sqsubset c_2$  and  $c_2 \sqsubset c_1$  then we say that  $c_1$  and  $c_2$  are synchronous ( $c_1 \equiv c_2$ ).  $c_1 \prec c_2$  represents causality relationships, *i.e.*,  $\forall k \in \mathbb{N}^* \text{ s.t. } c_1[k] \equiv c_2[k] \vee c_1[k] \prec c_2[k]$ .

A CCSL specification consists of clock declarations and conjunctions of *clock relations* between *clock expressions*. A clock expression defines a set of new clocks from existing ones, most expressions deterministically define one single clock. An example of clock expression is *delay* (denoted  $\$$ ).  $c \$ n$  specifies that a new clock is created and is the exact image of  $c$ , delayed for  $n$  instants of  $d$ .  $c \$_c n$  is simply written  $c \$ n$ .

By combining primitive relations and expressions, we derive a very useful clock relation that denotes a bounded precedence.  $c_1 \prec_n c_2$  is equivalent to the conjunction of  $c_1 \prec c_2$  and  $c_2 \prec (c_1 \$ n)$ . The special case, when  $n$  is equal to 1 is called *alternation* and is denoted  $c_1 \sim c_2$  (reads  $c_1$  alternates with  $c_2$ ).

Another useful CCSL expression is given by operator *union* (denoted  $+$ ).  $u \equiv c_1 + c_2$  defines a new clock  $u$  so that  $u$  is the fastest clock that is both a subclock of  $c_1$  and  $c_2$ :  $\forall k \in \mathbb{N}^*, \exists i \in \mathbb{N}^* \text{ s.t. } u[k] \equiv c_1[i] \vee u[k] \equiv c_2[i]$ .

The UML Profile for MARTE proposes several specific stereotypes in the Time chapter to capture CCSL specifications. Figure 4 briefly describes the three ones that are used in the paper. Boxes with the annotation «metaclass» denote the UML concepts on which our profile relies, so-called metaclasses. Boxes with «stereotype» are the concepts introduced by MARTE, *i.e.*, the stereotypes. Arrow with a filled head represent extensions, whereas normal arrows indicate properties of the introduced stereotypes. Clock extends UML Events to spot those events that can be used as time bases to express temporal or logical properties. ClockConstraint extends UML Constraints to make an explicit reference to the constrained clocks. TimedProcessing extends Action to make explicit their start and finishing events. When those events are clocks, then a ClockConstraint can constrain the underlying action to start or finish its execution as defined in a CCSL specification.

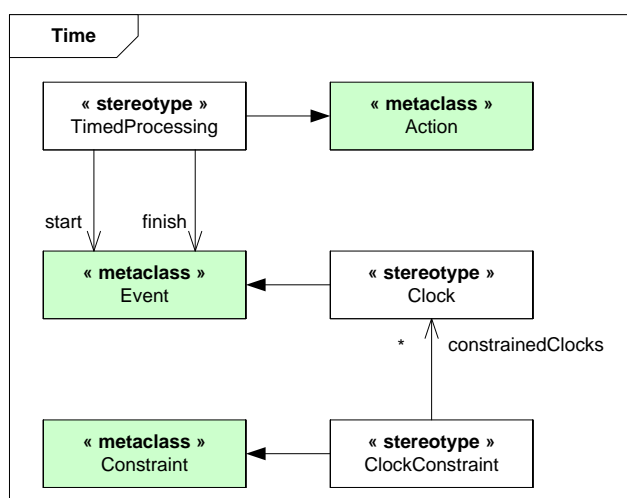


Fig. 4 Excerpt of the MARTE Time profile

Figure 5 shows an example where those stereotypes are applied. The two actions *Inversion* and *Filter* are stereotyped «TimedProcessing». Their respective start and finish events are *Inversion.start*, *Inversion.finish*, *Filter.start* and *Filter.finish*. Those events are clocks even though the stereotype is not actually displayed on that figure. Then a clock constraint specifies that *Inversion* must finish before and every time *Filter* starts.

The operational semantics of CCSL constraints [3] has been formally defined to allow for executing such specifications. TimeSquare<sup>5</sup> is an Eclipse-based environment dedicated to the analysis of CCSL models. It implements the operational semantics of CCSL constraints and eases the use of MARTE stereotypes. Amongst other features it also supports the animation of Papyrus

<sup>5</sup> <http://timesquare.inria.fr>

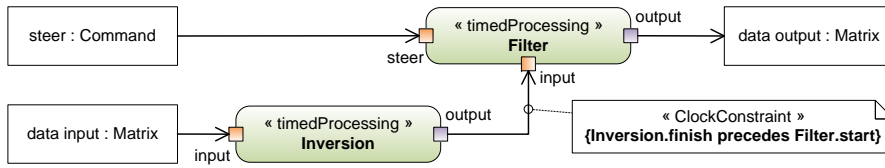



Fig. 5 Applying MARTE time constraints

UML diagrams according to a given CCSL specification. Papyrus is an academic, open-source, UML modeling tool provided by the CEA<sup>6</sup>. Figure 6 shows an execution trace produced by TimeSquare focusing on the four clocks involved in modeling the inversion and the filter. The dashed-arrows represent precedence relations between instants. Amongst others, the figure shows that the finish of *Inversion* precedes the start of *Filter*: *Inversion.finish*  *Filter.start*. The gray boxes are called ghosts. They represent instants at which a clock could have ticked but has not, *i.e.*, there is no constraint that either explicitly impose the clock to tick or prevent it from ticking.

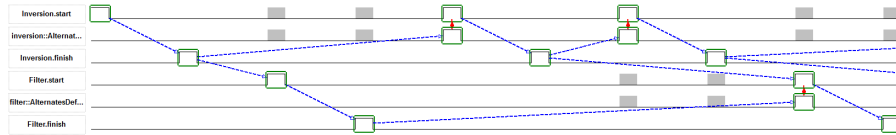


Fig. 6 Execution trace produced with TimeSquare

Note that this simple constraint allows streaming, which is not the by-default semantics of UML activities. Indeed, the second occurrence of *Filter.start* occurs after the third occurrence of *Inversion.finish*. This was not imposed by the specification (there is no dashed arrow) but this was not forbidden either. The two ghosts between the first and the second occurrence of *Filter.start* show that this second occurrence could have come earlier, but under no condition before the second occurrence of *Inversion.finish*.

#### 4 Modeling for Analyzing Applications in their Environment

In this section, we first discuss an alternative way to express an RSM-based model via an abstraction in order to mitigate its complexity in terms of parallelism (Section 4.1). Then, we deal with the explicit encoding of both the space-time mapping choices and environment constraints (Section 4.2). These constraints can be injected in a specification on arbitrary data structures and we propose a method to automatically propagate them as additional execution constraints (Section 4.3). We represent the allocation of an application on a

<sup>6</sup> <http://www.papyrusuml.org>



hardware platform by additional constraints, which do not reduce the possible scheduling solutions (Section 4.4).

#### 4.1 Capturing data dependences

An RSM-based model of an application expresses the data-dependences necessary to the modeled computation and thus the full potential parallelism permitted by this computation. This parallelism is expressed only for the components that have an implementation. The elementary components are seen as atomic (or sequential) computations. It is thus not possible to design a parallel library of components with hidden applications. This limits the reusability of components in the context of codesign where the effective use of the potential parallelism of a component is a decision that could be taken after the design of the component library.

We propose here a rich interface (as introduced by Alfaro and Henzinger in [2]) that exposes an abstraction of the potential parallelism of a component while hiding its implementation. The idea is to abstract a component as a repetition using the RSM concepts. Thus a designer can use such an abstraction in lieu of a component exposing its implementation for the purpose of deciding how to concretely use its internal parallelism.

The abstraction of a component  $\mathcal{C}$  is built using a bottom-up process starting from the elementary components used as building blocks of  $\mathcal{C}$  up to the top level composition of  $\mathcal{C}$ . At each level, the process uses refactoring transformations from the set of transformations described in [16,17].

At the lowest hierarchical level, the abstraction of an elementary component is a degenerate data-parallel repetition of one time the component itself. Indeed, there is no parallelism available, the execution is atomic.

There are two kinds of compositions to build composite components: the data-parallel repetition and the directed acyclic graph (DAG) composition. Let us now detail how we build our abstraction with these two compositions, starting with the data-parallel repetition.

The problem of the data-parallel repetition is to define how to build the abstraction of a data-parallel repetition of an abstraction of an internal repeated component. As this internal repeated component is abstracted itself as a data-parallel repetition, the problem can be viewed as how to abstract as a data-parallel repetition two nested data-parallel repetitions. This problem is exactly solved without any loss of parallelism by the “flatten” transformation.

The problem of the DAG composition is to define how to build the abstraction of a DAG composition of abstractions of internal components. As these are data-parallel repetitions, the “one-level” transformation is used to create a new hierarchy level where the top-level component is a data-parallel repetition of a DAG of the original internal components. The top-level repetition represents a factorization of the parallelism expressed by the repetitions of the internal components similar to what would be obtained by loop fusions. Actually, the “one-level” transformation is a succession of “fusion” and “flat-

ten” transformations two components at a time on the whole graph. The final abstraction is obtained by keeping only this top-level repetition as a way to express part of the potential parallelism of the original DAG composition. This process may lose some potential parallelism but keeps the parallelism that can be expressed as nested loops with uniform dependences.

For detailed examples of these transformations, the reader is invited to read the papers where they have been presented [16,17] and for the full details the PhD theses of Julien Soula [27], Philippe Dumont [8] and Calin Glitia [15].

For the rest of this paper the abstraction of the internal parallelism of a component is indistinguishable from an ordinary component exposing its implementation.

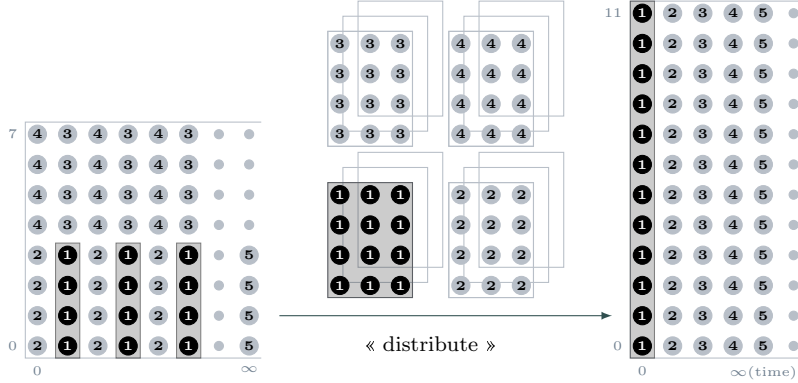
## 4.2 Space-time ordering

A data structure is represented in RSM by a port associated with a multi-dimensional shape specifying the number of elements on each dimension. To remind the reader, each single assignment data structure defines the entire range of values conveyed on the annotated port. At the implementation level, these values do not coexist in the same time range and not all the values need to be stored persistently in memory. Because all the dimensions are treated equally and time can be spread throughout multiple dimensions, the question is: *How can we define a data-flow order in time for a multidimensional data structure?*

We propose to define a special kind of data structure, called *data-flow shape*, where the time is isolated as the last dimension of the multidimensional shape. Because it is ordered in time, it is specified by applying stereotype «Clock» on a shape. Ordering an existing shape is achieved by allocating it onto a data-flow shape. This allocation must use the *distribute* concept of RSM to specify the exact correspondence between data elements from the two shapes.

Figure 7 shows a distribution example. How a data-flow shape propagates its ordering in time in the specification is the topic of the following section. This figure shows the distribution of a data array of shape  $(\infty)$  (on the left-hand side) onto a flow of shape  $(\infty)$  (on the right-hand side). To do the distribution, the data are virtually arranged within tiles of  $(\frac{4}{3})$  (shown in the middle of the figure). The numbers from 1 to 8 indicates how the tiles are built. The four first tiles are built from the six first columns of the data array. Each of these tiles make one single column of the output. «*distribute*» has four properties. The *patternShape* gives the shape of the intermediate tile  $(\frac{4}{3})$ . The *repetitionSpace* here would be  $\{2, 2, *\}$  meaning that bunches of 4 tiles  $(\frac{2}{2})$  are needed infinitely often in the intermediate representation. The *fromTiler* property defines how the elements are taken from the input shape to the intermediate tile. On this example, it would be *fromTiler* =  $\{origin = \{0, 1\}, fitting = \{\{1, 0\}, \{0, 2\}\}, paving = \{\{0, -1\}, \{4, 0\}, \{0, 6\}\}\}$ . Finally, the *toTiler* property defines how the elements are taken from the intermediate

tile and filled into the output shape. On this example, it would be  $toTiler = \{origin = \{0, 0\}, fitting = \{\{1, 0\}, \{4, 0\}\}, paving = \{\{0, 1\}, \{0, 2\}, \{0, 4\}\}\}$ . Section 3.1.1 explains the tiling concept in details. A practical example on the use of `distribute` is given in the following section (Fig. 15).



**Fig. 7** The distribution of a data array of shape  $\begin{pmatrix} 8 \\ \infty \end{pmatrix}$  onto a data-flow shape of  $\begin{pmatrix} 12 \\ \infty \end{pmatrix}$ .

### 4.3 Capturing execution constraints

We detail how the MARTE/RSM application can be enriched with explicit execution constraints. This section starts with the intra task constraints (*i.e.*, how we can constrain a single (repeated) task); then, we detail the inter task constraints that occur when various tasks are linked in a consumer/producer manner. Then, we take into consideration the time ordering induced by the allocation of a port onto a sensor (or actuator) port. Finally, the last subsection details the constraints that come from the allocation of tasks onto exclusive access resources (*e.g.*, processors, memory, bus).

#### 4.3.1 Intra-task dependency specification

In RSM, the structural units are (repeated) tasks. The behavior of such tasks represents either a single instance or a set of instances depending on the repetition space of the task. The execution conditions of this behavior is given by using stereotype «`TimedProcessing`» . It is used to create a start and a finish clock, which can be constrained. The minimal execution constraint used is the same than the one introduced at the end of section 3.2:  $task.start \prec task.finish$ .

Because in a first step, we consider only the logical parallelism, the data from the environment are always available and a task that consumes data from the environment has a maximal logical parallelism. Consequently, The «`TimedProcessing`» behavior represents either a single instance or a set of *concurrent* instances. We chose to represent the task instances in an activity

diagram (see Fig. 8). On this diagram, only the control flows are represented for simplification and clarity.

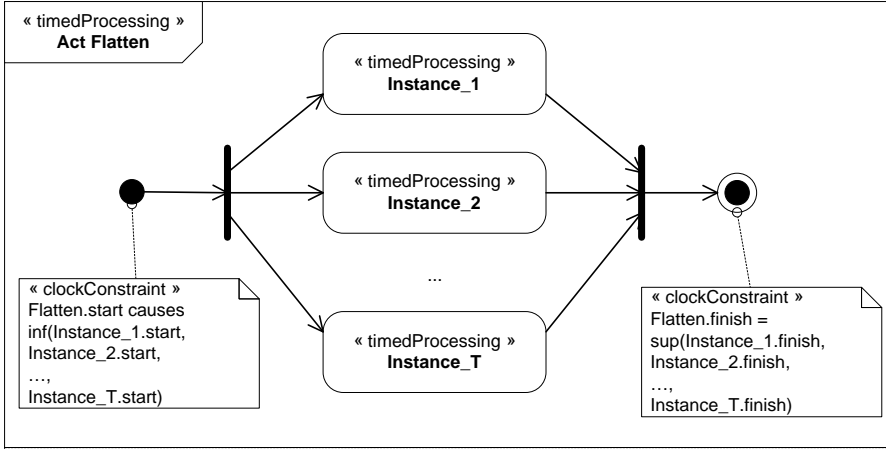


Fig. 8 Explicit parallelism

Each instance is stereotyped «TimedProcessing» to encode the potential parallelism. The instances can start at the same time or after the start of its parent behavior *Flatten*. *Flatten* finishes as soon as the latest of its contained instances finishes. CCSL operator *inf* ( $\wedge$ ) computes the slowest of the clocks faster than  $n$  given clocks. The  $k^{th}$  instant of  $c_1 \wedge c_2$  occurs simultaneously with the  $k^{th}$  instant of either  $c_1$  or  $c_2$ , whichever occurs first. In a dual way, *sup* ( $\vee$ ) is the fastest of the slower clocks. Consequently, two additional constraints make the link between the start/finish events of the compound *Flatten* and the start/finish events of each instance:

- Each of the  $T$  instances can start executing only after the start event of *Flatten*, expressed compactly in CCSL by :

$$Flatten.start \boxed{\preceq} Instance_1.start \wedge \dots \wedge Instance_T.start$$

- The finish event of *Flatten* coincides with the latest finish event of the  $T$  instances:

$$Flatten.finish \boxed{\equiv} Instance_1.finish \vee \dots \vee Instance_T.finish$$

#### 4.3.2 Inter-tasks dependency specification

By using the method presented in Section 3.1.4, the execution constraints between each producer/consumer are computed. The method result is that  $N_T$  executions of the producer are needed (to create enough data) for  $M_T$  executions of the consumer:

$$(Producer.finish \textit{by} N_T) \boxed{\prec} consumer.start$$

Note that parameter  $M_T$  is not used in this constraint but is used to constrain the number of parallel task instances used in the intra task parallelism. Consequently,  $M_T$  reflects the number of instances ( $T$ ) used in the previous subsection.

We have shown how logical time and CCSL can be used to capture the functional semantics of an RSM application, by encoding the execution constraints between computational tasks and instances. It specifies the set of correct schedules and can be used to simulate the model (*i.e.*, to compute one valid schedule if any). These schedules respect the data dependencies but are not aware of the execution platform constraints. It is as if the application would run on an ideal execution platform with infinite resources (execution, storage), no communication costs and where any operation can execute in zero time. When deploying an application on a real execution platform, execution constraints are introduced in the system. First of all we present how the deployment information is translated into additional execution constraints (specified also in CCSL), before introducing physical time concepts associated to the hardware platform.

#### 4.4 Encoding execution platform constraints

The application model captures the potential parallelism and assumes infinite resources (see Section 3.1.3). The execution platform model describes the actual resources and therefore the physical available parallelism. The allocation process maps application model elements onto execution platform elements thus reducing the set of correct schedules. This section discusses our proposition to model explicitly the allocation constraints using MARTE allocation and time subprofiles. It is important to be able to specify the allocation constraints so that the specification of the correct schedule stay consistent with the model. Moreover, it enables the detection of possible conflicts caused by the limited amount of resources. Limitations may come from the number of physical resources available or from non-functional properties, *e.g.*, a reduced time budget (deadline). Subsection 4.4.1 gives an example to capture the limitation of physical resources, whereas subsection 4.4.2 elaborates on constraints that must be applied when there is a limited time budget. The two kinds of constraints can be described independently, the composed specification is a conjunction of all the constraints.

##### 4.4.1 Constraints to capture limited hardware resources

This section focuses on the modeling of exclusive accesses to shared resources. We first assume the use of a mutual exclusion mechanism (see Eq. 2), then we show that this constraint can be relaxed if the resource allows several concurrent accesses (Eq. 3).

An example of such a resource is a (mono-core) processor but it is also the case for some communication buses or shared memories. For each exclusive resource  $Res$ , we define two clocks:  $Res_{acquire}$  and  $Res_{release}$ .  $Res_{acquire}$  represents the successive entering into the critical section, *i.e.*, the resource is acquired.  $Res_{release}$  models the instants at which the critical resource is released, *i.e.*, leaving the critical section.

Now let us consider the  $n$  tasks  $Ti$  ( $i \in [0; n]$ ) that are allocated onto this resource. For each task  $Ti$ , we consider two clocks  $Ti_{start}$  and  $Ti_{finish}$ . Clock  $Ti_{start}$  ticks whenever the task  $Ti$  starts and therefore when  $Ti$  acquires the shared resource  $Res$ . Clock  $Ti_{finish}$  ticks whenever  $Ti$  finishes and releases the shared resource<sup>7</sup>.

This informal description of exclusive resource access by concurrent tasks can be modeled in CCSL by the two following relations.

$$\begin{aligned} Res_{acquire} &\equiv T1_{start} + T2_{start} + \dots + Tn_{start} \\ Res_{release} &\equiv T1_{finish} + T2_{finish} + \dots + Tn_{finish} \end{aligned}$$

Because the access to the resource is exclusive, the resource cannot be acquired a second time if it has not been released first. This is captured by an alternation as follows:

$$Res_{acquire} \sim Res_{release}. \quad (2)$$

This latter constraint can be easily relaxed to model resources with several degrees of re-entrance. Indeed, it would just require to replace the alternation constraint (Eq. 2) by a bounded precedence (see Eq. 3).

$$Res_{acquire} \prec_n Res_{release} \quad (3)$$

Finally, the tasks cannot execute at the same time. This is specified in CCSL as follows:

$$\forall j \in [0; n], \forall k \in [0; n], j \neq k, Tj_{start} \# Tk_{start}.$$

Figure 9 shows an illustration with two tasks sharing the same processor.

This section has introduced a simple exclusion mechanism assuming a non-preemptive scheduler. However, more complex mechanisms, like the ones required in preemptive scheduling can also be encoded in CCSL [12].

#### 4.4.2 Constraints to capture limited time budgets

When the time limitations are inherent to an application itself, «TimedProcessing» discussed in section 3.2 can be applied to express time constraints on actions or behaviors. To express all time limitations it may be required to refine the application by making explicit the communication accesses. Figure 10 presents a refined view of the application, with a new action called *Transport* that represents the communication itself. When communications become explicit then

<sup>7</sup> Let us note that we do not model here the failed attempts to capture the resource but only the successful ones.

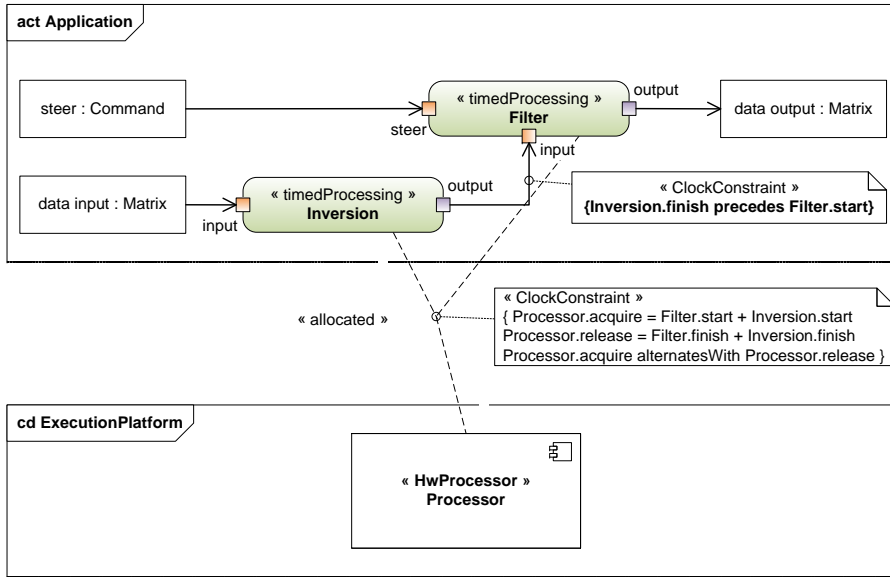


Fig. 9 Two tasks sharing the same processor

«TimedProcessing» is used to express constraints on the duration of the communication or relationships between the start and the end of the communication.

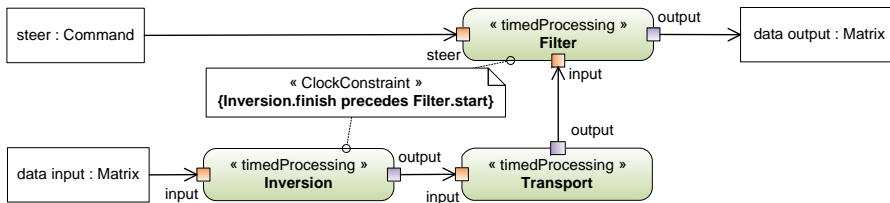


Fig. 10 Refined application with explicit communication accesses.

Let us note that for this model to be a refinement of the Figure 5 action *Transport* must conform to the initial constraint:

$$Inversion_{finish} \prec Filter_{start}$$

In particular, this requires to have lossless communications but does not require to have an upper bound for the communication duration.

When the time limitations come from the execution platform then allocation constraints are used in two ways:

1. spatial distribution of application tasks onto execution platform elements;

2. temporal scheduling of application tasks relative to the physical constraints imposed by the execution platform.

Temporal scheduling requires the association of task rates to the processor frequency on which it is allocated. As discussed before, task rates are expressed through logical time expressions relative to each other. Whereas, the processor frequency is imposed by a crystal oscillator and relates to physical durations. CCSL clocks can either be logical or physical. Since logical clocks have been discussed a lot, we elaborate now on mechanisms provided by CCSL to build physical clocks, so-called chronometric clocks.

The MARTE library defines a dense clock called *idealClock* that represents the perfect physical time. CCSL operator **discretizedBy** allows for discretizing dense clocks. Note that discretization is not necessarily perfect and two clocks discretized at the same rate need not be synchronous. The next two CCSL relations illustrate the use of this operator.

$$oscillator \equiv idealClock \text{ discretizedBy } 0.000001 \quad (4)$$

$$T_{finish} \prec (T_{start} \text{ \$ } (oscillator) 120) \quad (5)$$

Eq. 4 builds a new clock *oscillator* that represents a 1MHz oscillator. Eq. 5 expresses that the duration of task *T* is less than 120  $\mu$ s. T can either be a computation (e.g., Filter) of a communication (e.g., Transport).

Every part of the system can now be refined with physical time durations and still be simulated at the model level to ensure there still exist a correct schedule. Note that it is also possible to check the correctness of the CCSL specification with some non-functional requirements (deadlines, etc); see [12] for details.

## 5 Validation on a Radar Application

### 5.1 The STAP radar application: an informal description

STAP [18] is a Moving Target Indication (MTI) application (Fig. 11), whose goal is to detect from an aircraft the objects that move on the ground, and especially move slowly among all the other generally still reflecting surfaces under the radar beam (ground clutter). This is done by receiving the echo from the ground of a periodic sequence of radar pulses (bursts). Radar processing permits to estimate both the position of a target through the delay between transmission of a signal (pulse) and reception of its echo, and its speed through the Doppler effect that affects echoes of several identical pulses sent periodically: the speed of the target results in a (small) variation of its distance from the radar, which is only visible as a phase shift on the radar signal. In this basic approach, Doppler processing consists in a bench of filters (e.g. a Fast Fourier Transform) each tuned towards a particular phase shift between successive echoes. This kind of Doppler processing is in some situations sufficient to separate reflecting objects on the basis of their speed. When the beam is



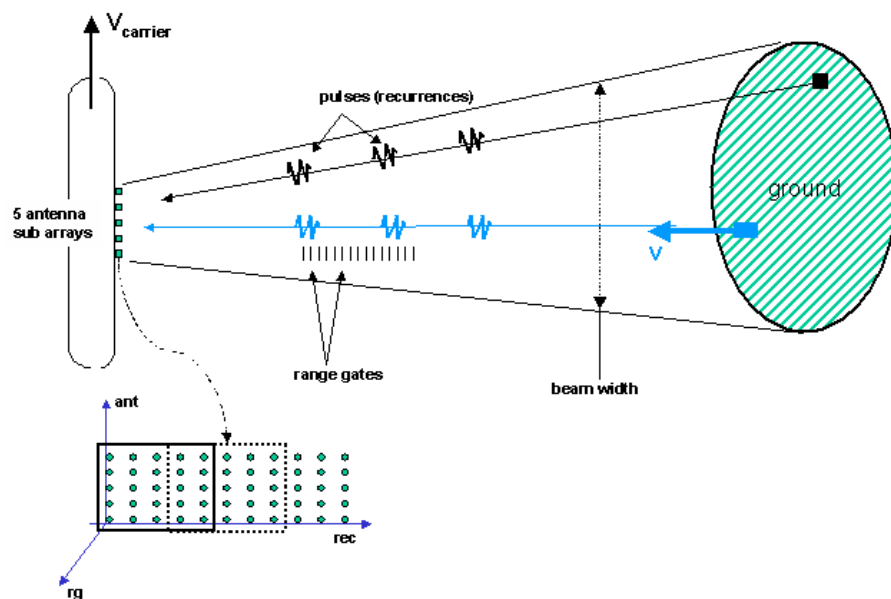


Fig. 11 STAP

directed towards the ground, the largest part of the echoed energy is supposed to come from the still objects that compose the ground (called clutter), while the moving objects of interest send weak but phase-shifted echoes. However, the radar beam is not perfectly sharp and has a width of a few degrees, which results in giving to some still objects on the ground at the borders of the beam a relative speed with respect to the aircraft (due to the aircraft's own speed) and creating undesired interferences over the moving targets echoes: this creates an ambiguity between intrinsic speeds and azimuths of targets. Adaptive filtering techniques, where fixed filters are replaced by filters that are computed at run-time from the received signal itself, help to minimize at best the effects of an unwanted clutter signal due to the movement of the aircraft: in this MTI case, the Space Time Adaptive Processing (STAP) is used.

In this method, a set of filters is computed at every burst, by solving linear systems whose right hand side terms are reference vectors of theoretical phase patterns expected on antenna sub-arrays at several consecutive pulses, each of which corresponding to a particular (velocity, angle) hypothesis of the target relatively to the aircraft. This is shown in Figure 11 where 2D patterns on dimensions antenna and pulse (rec) are considered to compute filters that remove the natural ambiguity between velocity and azimuth.

## 5.2 The STAP model in MARTE RSM

The STAP application was modeled in Papyrus UML using the MARTE Profile (Fig. 12)<sup>8</sup>. Starting from the top level, the application is successively depicted

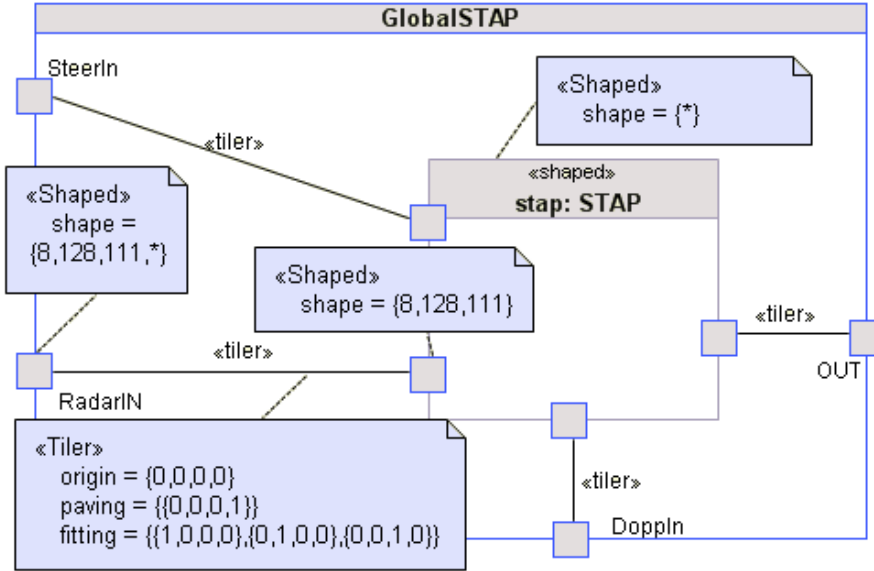


Fig. 12 STAP: coarse-grain data-flow level

using a compound or repetitive decomposition. The compound is made of tasks that can be deployed on library functions. The task can be further decomposed if more parallelism must be shown or exploited (Fig. 14). The infinite dimension of the arrays processed by *GlobalSTAP* represents the abstraction of time. Figure 12 describes the data-flow level of the application including the infinite repetition (over time) of a single STAP data treatment.

Just one sample tiler is shown, expressing how the infinite radar signals, represented by an array of shape  $\{8, 128, 111, *\}$  is decomposed into an infinity  $\{*\}$  of patterns with a shape  $\{8, 128, 111\}$ . The paving matrix  $\{\{0, 0, 0, 1\}\}$  expresses the correspondence between the infinite repetition and the last dimension of the array. The fitting matrix  $\{\{1, 0, 0, 0\}, \{0, 1, 0, 0\}, \{0, 0, 1, 0\}\}$  expresses the correspondence between the pattern dimensions and the first three dimensions of the array.

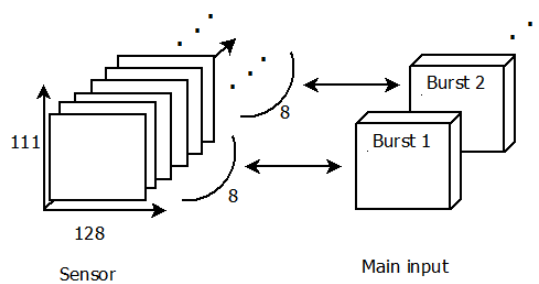
Here the *stap* element needs to be further decomposed into simpler tasks until we reach a level where elementary tasks are allocated to the model of the execution platform. Figure 14 illustrates the compound decomposition of the repeated task *stap* into successive repetitive filters, with array sizes and

<sup>8</sup> Throughout this last section, figures are screen captures of actual UML models.

repetition spaces shown on the figure. The tilers that express the pattern/tiles construction for each repetition are not made visible.<sup>9</sup> At the filter level, each repeated task has an elementary functionality which can be deployed on library functions: *e.g.*, matrix inversion, average computations. Elementary tasks are shown with a dark background whereas compound tasks are shown with a white background.

### 5.3 Space time ordering

As described in Section 4.2, we first have to make explicit the time ordering for the availability of the input data. We make the assumption that the main input (*RadarIn*) with a shape  $\{8, 128, 111, *\}$  is distributed on a sensor that provides an array of data with a shape  $\{128, 111, *\}$  (see Fig. 13).



**Fig. 13** The distribution of the main input on the sensor

The eight antennas provide their signals in sequence periodically. Thus, the 8 first arrays altogether form the first burst of data of the main input. The distribution is depicted in Figure 13. The equivalent RSM modeling using the distribute stereotype is depicted in Figure 15.

One should also note that the consequence of this specific space time ordering is that the task *Pulse* computes its entire output arrays before the next tasks can run. However, the other tasks can be pipelined and thus the impact on the memory is minimal. Table 1 gives the memory space required during the execution.

### 5.4 Capturing data dependencies

Following our methodology described in Section 4.3, we have computed the following execution dependencies.

<sup>9</sup> A complete model is available at [http://www-sop.inria.fr/aoste/dev/time\\_square/ccsl-rsm](http://www-sop.inria.fr/aoste/dev/time_square/ccsl-rsm).

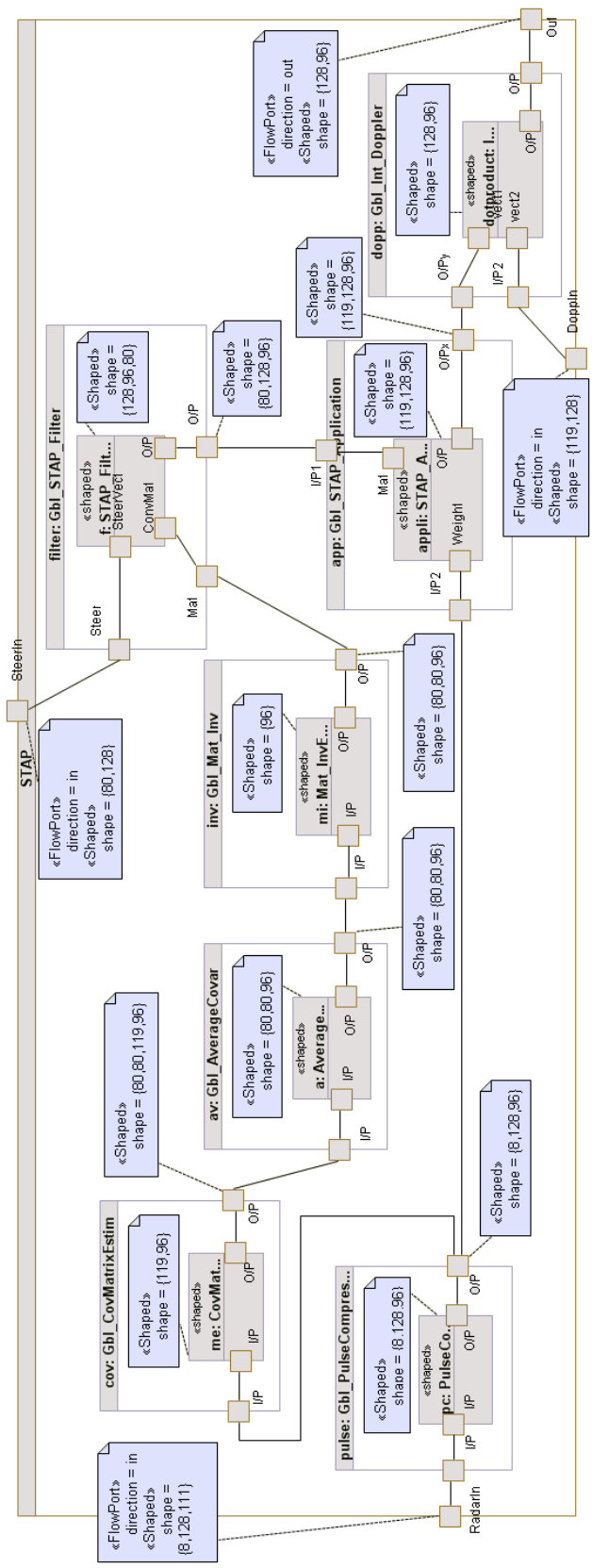


Fig. 14 STAP decomposition

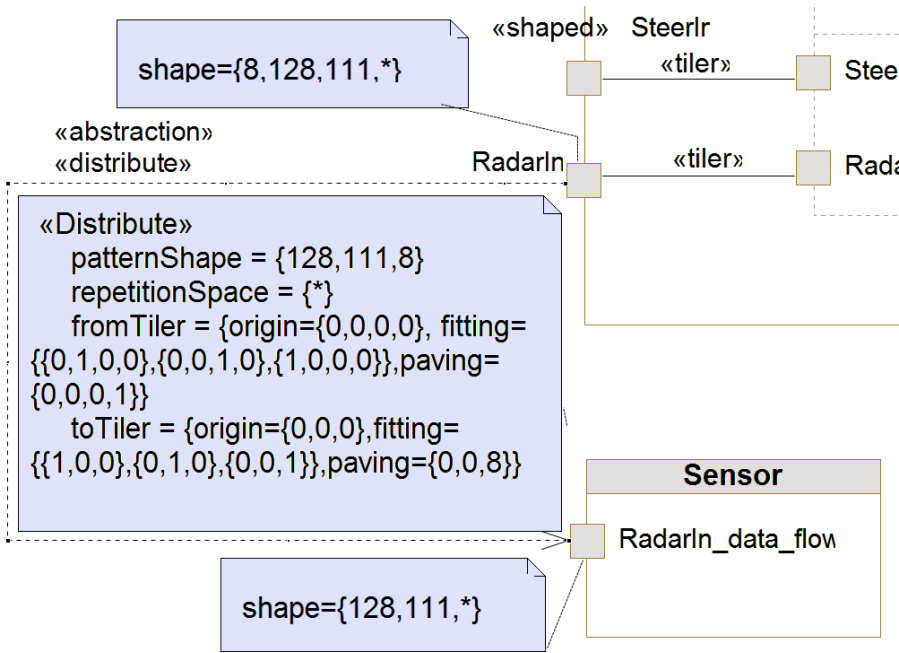


Fig. 15 The distribution of the main input on the sensor: RSM model

Interconnection	Size of the array
<i>Sensor</i> → <i>Pulse</i>	128 * 111
<i>Pulse</i> → <i>CovMatrix</i> and <i>STAP_application</i>	8 * 128 * 96
<i>CovMatrix</i> → <i>Average</i>	80 * 80 * 119
<i>Average</i> → <i>Inversion</i>	80 * 80
<i>Inversion</i> → <i>Filter</i>	80 * 80
<i>SteerIn</i> → <i>Filter</i>	80 * 128
<i>Filter</i> → <i>STAP_application</i>	80 * 128
<i>STAP_application</i> → <i>Doppler</i>	119 * 128
<i>DoppIn</i> → <i>Doppler</i>	119 * 128
<i>Doppler</i> → <i>OUT</i>	128

Table 1 Memory requirement of the STAP application

*Intra-task dependency specification* A first trivial constraint for each task specifies that each task must start before it finishes:  $task_{start} \preceq task_{finish}$  where  $task$  takes values in  $\{PulseCompression, CovMatrix, Average, Inversion, Filter, STAP\_application, Doppler\}$ .

*Inter-task dependency specification* Figure 16 gives the dependency constraints extracted from Figure 14.

*Intra-task parallelism specification* : Each task can be parallelized to compute patterns simultaneously. All its instances can be executed in parallel. The

following two CCSL constraints must be applied to every task.

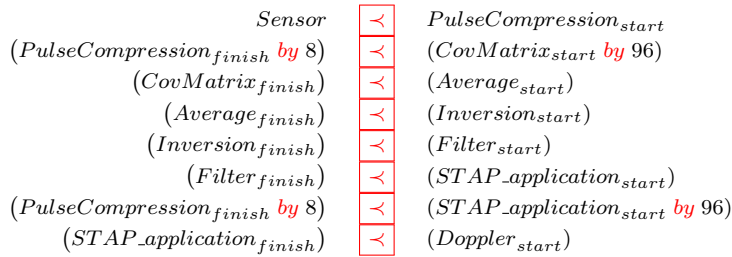
$$\begin{aligned} task_{start} & \begin{array}{|c|} \hline \prec \\ \hline \end{array} task_{i1}_{start} \wedge \dots \wedge task_{iX}_{start} \\ task_{finish} & \begin{array}{|c|} \hline \equiv \\ \hline \end{array} task_{i1}_{finish} \vee \dots \vee task_{iX}_{finish} \end{aligned}$$

where  $task$  takes values in  $\{PulseCompression, CovMatrix, Average, Inversion, Filter, STAP\_application, Doppler\}$ .

Table 2 gives the number of parallel instances (denoted  $X$  in the following) for each task. The last column gives the number of repetitions of each instance required to complete a burst of data of the global input.

Task	Number of parallel instances ( $X$ )	Repetition
<i>PulseCompression</i>	$128 * 96 = 12288$	8
<i>CovMatrix</i>	119	96
<i>Average</i>	$80 * 80 = 6400$	96
<i>Inversion</i>	1	96
<i>Filter</i>	$80 * 128 = 10240$	96
<i>STAP_application</i>	$119 * 128 = 15232$	96
<i>Doppler</i>	128	96

**Table 2** Number of parallel instances ( $X$ ) for every task



**Fig. 16** Inter-task dependency specification

The STAP application does not contain any inter repetition dependencies.

## 5.5 Hardware allocation of the STAP application

The application has been deployed on an illustrative multi core architecture. The architecture is composed of an array of cores with a regular communication topology such as a network-on-chip. We assume that the communication bandwidth on the network is non-blocking.

From the data dependencies, we can see that task *PulseCompression* has to be completed before the next tasks start. However, the six other tasks can be pipelined efficiently. Thus six cores has been affected to the application.

Task *PulseCompression* is distributed on the six cores. Each core deals with 2048 (= 12288/6) instances. Then, each of the six other tasks have been deployed on one core each. *CovMatrix* on *core*<sub>1</sub>, *Average* on *core*<sub>2</sub> and so on.

This deployment attempts to optimize the parallel usage of the cores. However, any other optimization criteria could have been selected and captured. To capture this allocation specification, the following CCSL constraints must be given:

The access to *core*<sub>1</sub> is shared between the 2048 first instances of task *PulseCompression* and all the 119 instances of task *CovMatrix*. This is modeled using the mechanism described in Section 4.4.1. All these instances are executed on the same core in a mutually exclusive way, one after the other. We do care about their relative ordering, but we do not want all of them to be executed simultaneously.

$$\begin{aligned} core_{1\_acquire} &\equiv CovMatrix.i1_{start} + \dots + CovMatrix.i119_{start} + \\ &PulseCompression.i1_{start} + \dots + PulseCompression.i2048_{start} \\ core_{1\_release} &\equiv CovMatrix.i1_{finish} + \dots + CovMatrix.i119_{finish} + \\ &PulseCompression.i1_{finish} + \dots + PulseCompression.i2048_{finish} \end{aligned}$$

Similarly, the access to the other cores is shared between 2048 instances of task *PulseCompression* and all the instances of the task allocated on that core. Moreover, each of the six cores are exclusive resources:

$\forall c \in [1, 6], core\_c\_acquire \approx core\_c\_release$  and

$$\forall j, k, j \in [0; n], k \in [0; n], j \neq k, Tj_{start} \# Tk_{start}.$$

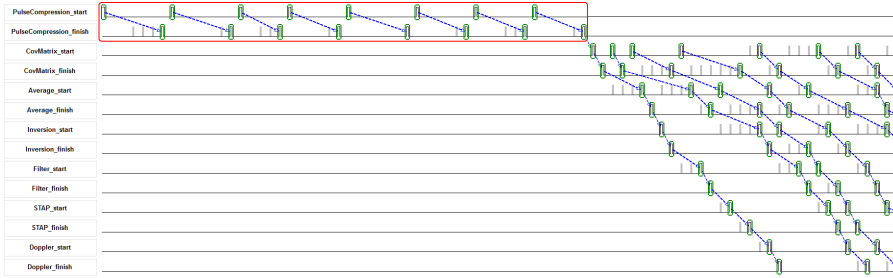
where  $n$  is the number of tasks on *core* <sub>$c$</sub>  and  $Tj, Tk$  are instances allocated to *core* <sub>$c$</sub> .

## 5.6 Simulation in TimeSquare

The CCSL clocks and constraints extracted from the STAP application have been entered in the tool Timesquare<sup>10</sup> and simulated. TimeSquare executes the operational semantics of CCSL constraints step by steps. At each step, it computes the set of all the possible solutions that satisfies the CCSL specification. If there is no possible solution, the specification is rejected meaning that some constraints are contradictory. If there is only one solution, then the specification is deterministic, the solution is applied and consequently all the clocks enabled in the solution are fired. If there are several solutions, then a simulation policy is applied to pick one solution amongst the valid ones. In all the simulations performed in this section, we have always chosen the *balanced random* simulation policy, that picks randomly one solution amongst the acceptable ones. Other simulation policies could have been chosen.

<sup>10</sup> <http://timesquare.inria.fr>

Figure 17 presents one of the possible execution traces of the STAP. This execution highlights data dependencies imposed by RSM stereotypes. Ghosts (gray boxes) show instants at which there were some non-determinism that was resolved by making the clocks not ticking. As explained before, dashed arrows show constraints imposed by the CCSL specification. The big (red) rectangle<sup>11</sup> shows that task *PulseCompression* must execute eight times to process the whole arrays upfront before any other task can execute. Then, the six other tasks (*CovMatrix*, *Average*, *Inversion*, *Filter*, *STAP*, *Doppler*) can execute 96 times in a pipelined fashion. This pattern of execution computes exactly one burst of data. The pattern repeats infinitely often over the time dimension.



**Fig. 17** STAP data dependencies

Figure 18 focuses on the pipeline itself and on the usage of the shared cores. It shows another possible execution trace and use the ASAP (As soon as possible) simulation policy. In the first phase, all the six cores execute the 12288 instances of *PulseCompression*. Then each of the remaining six tasks is executed on a different core. Once the pipeline is filled, then the 96 execution can execute at full speed with a 100% usage of the cores. At the end, the pipeline is flushed and the pattern starts again.

The STAP specification does not explicitly impose to wait for the 96<sup>th</sup> execution of *Doppler* before starting the pattern again. This has been imposed by adding a back-pressure constraint. Without this constraint, some instances of *PulseCompression* could have been interleaved with the pipelined tasks. However, this would require having buffers large enough to save several instances (at least two) of the full array. This back-pressure constraint makes it possible to process the STAP with the full array stored only once. There cannot be a bigger reduction of the memory usage anyway since task *CovMatrix* requires having access to the full array before starting its execution.

Figure 19 shows a refinement of what is happening between each couple of *PulseCompression\_start* and *PulseCompression\_finish*. The 12288 instances are spread into six groups of 2048, each of which is allocated on a different core. Each group can therefore execute concurrently, all the groups

<sup>11</sup> This red rectangle has been superimposed for the discussion. Timesquare can only display groups of ticks of the same clock (in green).



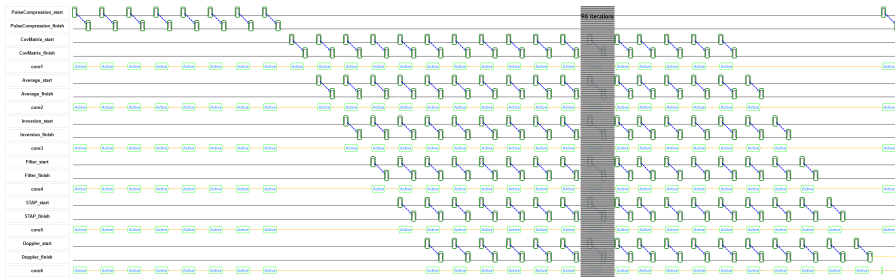


Fig. 18 Pipelined execution of STAP

must finish one execution before task *PulseCompression* can be considered as finished. This was captured using CCSL *inf* and *sup* operators. It is depicted by the vertical (red) plain lines, which mark coincident instants. Each occurrence of *PulseCompression\_start* coincides with the earliest of the six groups, whereas, each occurrence of *PulseCompression\_finish* coincides with the latest one. Since nothing constrains the duration of each instance, the duration can be arbitrarily high. We could refine what is happening between

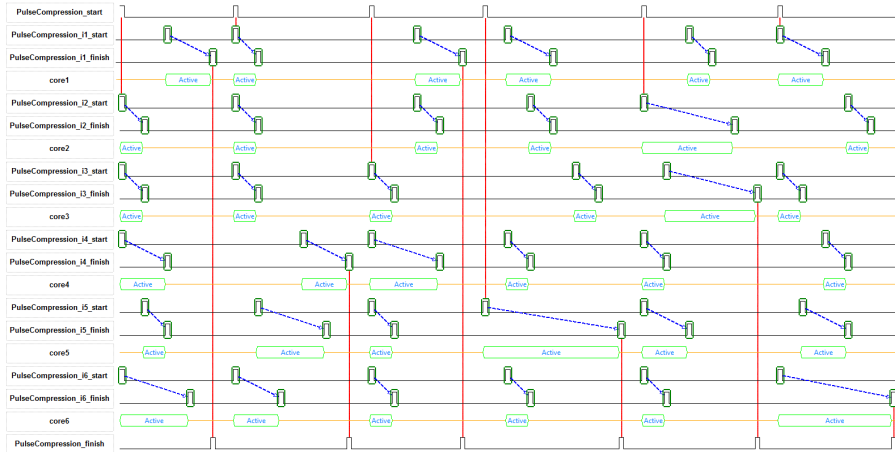


Fig. 19 Instances of PulseCompression executing concurrently on six cores

*PulseCompression\_i1\_start* and *PulseCompression\_i1\_finish* to see the actual execution of the 2048 instances that execute on *core1* (with a shared access). However, that would clearly point at the limitation of pure graphical representations and stress the need for automatic trace exploration techniques that have not been discussed here but are also available in Timesquare.

The size of the considered example poses a clear problem of scalability. The problem is not so much to solve the Boolean constraints but rather to capture the thousands of constraints needed to model all the instance executions. The abstraction presented in Section 4.1 can be used to abstract away some fine

grain parallelism at the price of a reduction of scheduling possibilities. Another way to deal with this large number of constraints is studied in some ongoing extensions of CCSL that could be applied at the meta-model level rather than at the model level. With such a facility it should be possible to specify that a constraint should apply uniformly to the 12288 instances of one particular model element, without having to enumerate all the constraints by hand. Another key issue concerns visualization. Graphical representations even though useful to understand what happens are of little help when it comes to a systematic analysis of specifications. TimeSquare offers support for systematic analyses not discussed here.

## 6 Conclusions

RSM has been proposed in the annex of the MARTE profile as a means to specify logical parallelism of multi-dimensional data-flow applications. The MARTE specification gives several notations to address various aspects of embedded systems but does not provide any methodology or even real size example on how it can be exploited. This paper describes a family of applications, *i.e.*, computation-intensive embedded systems, and show how different concepts of MARTE can be used to capture complementary parts. Indeed, the proposed approach combines the use of the following concepts: Generic Component Modeling, Hardware Resource Modeling, Allocation, Time, Repetitive Structure Modeling.

By using the MARTE time model, we first proposed a way to introduce explicitly the ordering in time rather than hiding it in a compilation phase. Second, we enhanced the RSM model with logical clocks whose evolutions are constrained by CCSL to represent the correct schedules with regards to data dependencies. It gives a formal, explicit and executable semantics to the model. Based on this, we have shown how it is possible to add CCSL constraints that represent allocation choices of task instances on a hardware platform. These constraints restrict the logical parallelism by the physical parallelism of the platform. The combined use of RSM and CCSL has been illustrated on a non-trivial example.

Future work should consider a larger part of the design flow presented in this paper, from requirements to analysis, so as to cover an even larger part of MARTE specification.

## References

1. Abdallah, A., Gamatié, A., Dekeyser, J.L.: Correct and energy-efficient design of socs: The h.264 encoder case study. In: System on Chip (SoC), 2010 International Symposium on, pp. 115–120 (2010)
2. Alfaro, L., Henzinger, T.A.: Interface theories for Component-Based design. In: T.A. Henzinger, C.M. Kirsch (eds.) *Embedded Software, Lecture Notes in Computer Science*, vol. 2211, pp. 148–165. Springer Berlin Heidelberg, Berlin, Heidelberg (2001). URL <http://www.springerlink.com/content/0jqhuw40j1rbk8c7/>

3. André, C.: Syntax and semantics of the Clock Constraint Specification Language (CCSL). Research Report 6925, INRIA (2009). URL <http://hal.inria.fr/inria-00384077/>
4. Benveniste, A., Le Guernic, P., Jacquemot, C.: Synchronous programming with events and relations: the SIGNAL language and its semantics. *Sci. Comput. Program.* **16**(2), 103–149 (1991)
5. Berry, G.: The foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner* pp. 425–454 (2000)
6. Boulet, P., Marquet, P., Piel, E., Taillard, J.: Repetitive Allocation Modeling with MARTE. In: *Forum on specification and design languages (FDL'07)*. Barcelona, Spain (2007)
7. Boussinot, F., De Simone, R.: The ESTEREL language. *Proceedings of the IEEE* **79**(9), 1293–1304 (2002)
8. Dumont, P.: Spécification multidimensionnelle pour le traitement du signal systématique. Ph.D. thesis, , (2005)
9. Falk, J., Keinert, J., Haubelt, C., Teich, J., Zebelein, C.: Integrated modeling using finite state machines and dataflow graphs. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) *Handbook of Signal Processing Systems*, pp. 1041–1075. Springer US (2010). URL [http://dx.doi.org/10.1007/978-1-4419-6345-1\\_36](http://dx.doi.org/10.1007/978-1-4419-6345-1_36)
10. Feautrier, P.: Automatic parallelization in the polytope model. In: G.R. Perrin, A. Darté (eds.) *The Data Parallel Programming Model, Lecture Notes in Computer Science*, vol. 1132, pp. 79–103. Springer Berlin / Heidelberg (1996)
11. Fidge, C.: Logical time in distributed computing systems. *Computer* **24**(8), 28–33 (2002)
12. Peraldi Frati, M.A., Deantoni, J.: Scheduling Multi Clock Real Time Systems: From Requirements to Implementation. In: *IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing*, p. 50; 57. IEEE computer society, Newport Beach États-Unis (2011). Newport Beach
13. Gamatié, A.: Specification of data intensive applications with data dependency and abstract clocks. In: B. Furht, A. Escalante (eds.) *Handbook of Data Intensive Computing*, pp. 323–348. Springer New York (2011). URL [http://dx.doi.org/10.1007/978-1-4614-1415-5\\_12](http://dx.doi.org/10.1007/978-1-4614-1415-5_12)
14. Gamatié, A., Beux, S.L., Piel, É., Atitallah, R.B., Etien, A., Marquet, P., Dekeyser, J.L.: A model-driven design framework for massively parallel embedded systems. *ACM Trans. Embedded Comput. Syst.* **10**(4), 39 (2011)
15. Glitia, C.: Optimisation des applications de traitement systématique intensives sur systems-on-chip. Ph.D. thesis, , Université Lille 1, Sciences et Technologies (2009)
16. Glitia, C., Boulet, P.: High level loop transformations for multidimensional signal processing embedded applications. In: *SAMOS 2008 Workshop*. Samos, Greece (2008)
17. Glitia, C., Boulet, P.: Interaction between inter-repetition dependences and high-level transformations in array-ol. In: *Conference on Design and Architectures for Signal and Image Processing 2009*. Sophia Antipolis, France (2009)
18. Glitia, C., Boulet, P., Lenormand, E., Barreteau, M.: Repetitive model refactoring strategy for the design space exploration of intensive signal processing applications. *Journal of Systems Architecture* **57**(9), 815–829 (2011). DOI 10.1016/j.sysarc.2010.12.002. URL <http://www.sciencedirect.com/science/article/pii/S1383762110001645>
19. Glitia, C., DeAntoni, J., Mallet, F.: Logical time at work: capturing data dependencies and platform constraints. *International Forum for Design Languages (FDL 2010)* (2010)
20. Glitia, C., Dumont, P., Boulet, P.: Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Systems and Signal Processing* **21**(2), 105–131 (2010)
21. Kahn, G.: The semantics of a simple language for parallel programming. *Information Processing* pp. 471–475 (1974)
22. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* **21**(7), 558–565 (1978)
23. Lee, E.A.: Multidimensional streams rooted in dataflow. In: *Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pp. 295–306 (1993)
24. Lee, E.A., Messerschmitt, D.G.: Synchronous Data Flow. *Proc. of the IEEE* **75**(9), 1235–1245 (1987)

- 
25. OMG: UML Profile for MARTE, v1.0. Object Management Group (2009). Document number: formal/09-11-02
  26. OMG: UML Superstructure, v2.2. Object Management Group (2009). Formal/2009-02-02
  27. Soula, J.: Principe de compilation d'un langage de traitement de signal. Ph.D. thesis, , (2001)