Model Transformations from a Data Parallel Formalism towards Synchronous Languages

Huafeng Yu, Abdoulaye Gamatié, Eric Rutten, Jean-Luc Dekeyser

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Model Transformations from a Data Parallel Formalism towards Synchronous Languages

Huafeng Yu — Abdoulaye Gamatié — Éric Rutten — Jean-Luc Dekeyser

N° 6291

September 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6291--FR+ENG

Model Transformations from a Data Parallel Formalism towards Synchronous Languages

Huafeng Yu* , Abdoulaye Gamatié† , Éric Rutten‡ , Jean-Luc Dekeyser§

Thème COM — Systèmes communicants
Projets DaRT, Pop Art

Rapport de recherche n° 6291 — September 2007 — 43 pages

Abstract: The increasing complexity of embedded system designs calls for high-level specification formalisms and for automated transformations towards lower-level descriptions. In this report, a metamodel and a transformation chain are defined from a high-level modeling framework, Gaspard, for data-parallel systems towards a formalism of synchronous equations. These equations are translated in synchronous data-flow languages, such as Lustre, Lucid synchrone and Signal, which provide designers with formal techniques and tools for validation. In order to benefit from the methodological advantages of re-usability and platform-independence, a Model-Driven Engineering approach is applied.

Key-words: Intensive signal processing, data-parallel, Gaspard, Array-OL, synchronous approach, MDE, model transformations

* huafeng.yu@lifl.fr
† abdoulaye.gamatie@lifl.fr
‡ eric.rutten@inria.fr
§ jean-luc.dekeyser@lifl.fr

Unité de recherche INRIA Futurs
Parc Club Orsay Université, ZAC des Vignes,
4, rue Jacques Monod, 91893 ORSAY Cedex (France)
Téléphone : +33 1 72 92 59 00 —Télécopie : +33 1 60 19 66 08

Transformation de modèles à partir d'un formalisme à parallélisme de données vers des langages synchrones

Résumé : La complexité croissante de la conception des systèmes embarqués entraîne un besoin de formalismes de spécifications de haut niveau ainsi que des transformations automatisées vers des descriptions de plus bas niveau. Dans ce rapport, un métamodèle et une chaîne de transformation sont définis à partir d'un cadre de modélisation de haut niveau, GASPARD, pour des systèmes avec du parallélisme de données, vers un formalisme d'équations synchrones. Ces équations sont ensuite traduites dans les languages synchrones flot de données, tels que LUSTRE, LUCID SYNCHRONE et SIGNAL, qui offrent aux concepteurs des techniques et outils de validation. Afin de bénéficier des avantages méthodologiques de la réutilisabilité et de l'indépendance vis-à-vis de toute plate-forme, une approche d'Ingénierie Dirigée par les Modèles (IDM) est appliquée.

Mots-clés : Traitement intensif du signal, parallélisme de données, GASPARD, ARRAY-OL, approche synchrone, IDM, transformations de modèles

Contents

1 Context and motivation

1.1 MDE and data-parallel applications

Data-parallel applications, such as mobile multimedia processing, high-definition TV and radar/sonar signal processing, play an increasingly important role in embedded systems. Parallel massive data processing is a key feature in these applications. Unlike general parallel applications which focus on the code parallelization and their communications, data-parallel applications pay more attention to data distribution and their access. The data manipulated in these applications are generally in multidimensional data structure, such as multidimensional arrays. And regularly repeated computations manipulate associated small data blocks, which are called *tiles* and can be also multidimensional, from the input data. But these applications become increasingly more complex following the trend of integration of more and more various functionalities into one single system and/or into one single chip. As a result, their design, implementation and validation turn out to be dramatically more complicated and difficult. Furthermore, the increasing system complexity leads to the productivity problem that becomes a great constraint for the developments. Hence, more efficient design methods, including modeling, implementation and validation methods are highly needed.

Nowadays, among intensive research activities to address such problems, Model-Driven Engineering (MDE) [34, 28] based methods should be mentioned. Model is the key concept in MDE, which is greatly influenced by the concept of abstraction. It enables to represent the system with an accepted level of abstraction, i.e., all unnecessary details of the system are removed for the sake of simplicity of modeling, analysis, etc., whereas the obtained models is usable. Another key point of MDE is to enable the transformations between models. These model transformations are not only from high-abstraction level into lower ones, but from models of one domain into models of other domains. One of the best known MDE initiative is Model-Driven Architecture (MDA) [27], which is proposed by OMG (Object Management Group) [30]. According to MDA, two types of model are distinguished following the abstraction level: Platform-Independent Models (PIM) and Platform-Specific Models (PSM). The former generally represents the system functional requirements, and the PSM always involves implementation concerns. For example, a PSM can be an executable model itself, or be used to generate certain domain-specific source code, etc.. Transformation specifications are also proposed by OMG to bridge these two types of model. UML is always taken as the visual modeling language in MDE. It is supported by many visual modeling environments and tools now. MDE brings several advantages: well-defined modeling specifications lead to rapid design as well as concise and clear documentation; and their automated transformations offer the opportunity to simplify the code generation in consideration of their correctness; finally, the re-usability and modularity of their models and Intellectual Properties (IPs) make the development of these applications more efficient and rapid.

1.2 The GASPARD methodology for data-parallel computing

GASPARD [17] is a MDE based development environment and methodology for the design of data-parallel applications. It proposes concepts, which feature high level data-parallel concepts, data flow and control flow mixing, hierarchical and repetitive application and architecture models, etc. The inherent data-parallel description of GASPARD was adopted by MARTE (Modeling and Analysis of Real-Time and Embedded systems) [33], which is an OMG standard proposal for the modeling and analysis of real-time embedded systems. One of the important feature of GASPARD is its software/hardware co-modeling and model transformations. More precisely, it enables to model *software applications*, *hardware architectures*, their *association* and IP *deployment* through predefined metamodel in a unique modeling environment. This modeling stays in a high abstraction level and is usually platform independent. GASPARD enables as well transformations from these models to lower level models, which are involved in the execution, synthesis and validation issues.

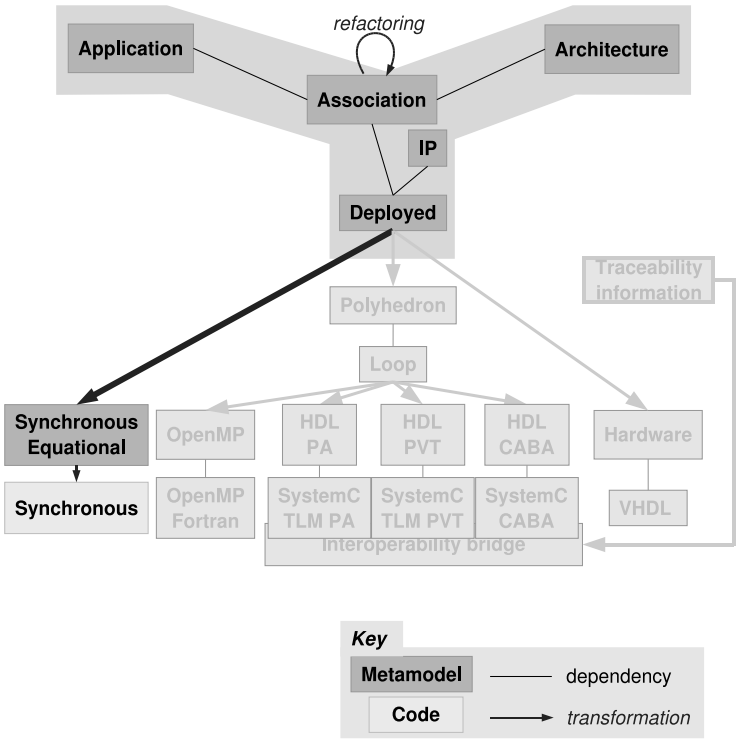

Figure 1: Y-chart according to GASPARD.

This metamodel is partially based on the concept of the Y-chart (see Fig. 1 and [17]). Models for application and hardware architecture are defined separately. Then, application models can be mapped on architecture models. The obtained models are linked to software or hardware IPs during the deployment phase. All these models are platform-independent, i.e., they are not associated with an execution, simulation, validation or synthesis technology. Model transformations are performed from deployed models to specific languages (synchronous languages and others, which are not detailed in this paper and are shadowed in Fig. 1, such as HPF, SystemC and VHDL). These characteristics of Gaspard make it easier to reduce the complexity of system design.

In the following, we briefly present the main features of the Gaspard metamodel:

- **Application** focuses on the description of data parallelism and data dependencies between application components. These components and dependencies completely describe the functional behavior. Application components mainly manipulate multi-dimensional arrays, with a possible infinite dimension to represent time. Data-parallel constructs, such as "tiler" (see [6]) has been included in this metamodel, which offers the opportunity of expressing the regular multidimensional data access. This metamodel defines as well task repetition in correspondence to data-parallelism. The tasks are either atomic computations (elementary task) on arrays or composed tasks (hierarchical task).

- **Architecture** specifies the hardware architecture at a high abstraction level. It enables to dimension hardware resources. A mechanism similar to the one used in application enables to specify repetitive architecture in a compact way as the increasing popularity of these kinds of regular parallel computation units in hardware.

- **Association.** allows one to express how the application is projected on the hardware architecture, i.e., which hardware component executes which functionality. One particularity of this metamodel is to consider the mapping as well as the parallelism both in the application and architecture.

- **Deployment** (represented by the box tagged as "Deployed" in Fig. 1) enables to chose a specific target platform for code generation from Gaspard models. This is achieved by importing IPs in which no implementation details are provided except their interfaces, information about compilation and their inter-communications.

1.3 Motivation: connecting Gaspard to validation tools

In embedded system design, the top-down approach, which goes from the high abstraction level to low implementation levels, is well adopted. Furthermore, the reliability of the system is always a significant issue, which requires the correctness of the design and implementation. On one side, the automatic transformation from the system specification to implementation details reduces the error occurrences caused by the manual coding. On the other side, the

design correctness verification at the original high abstraction level is as well necessary as the high level verification helps to reduce the system design complexity and to avoid to handle too many implementation details at the beginning development stage.

GASPARD models are dedicated to the co-modeling of data-parallel applications at high abstraction level with the help of the UML visual modeling language. However UML suffers from the lack of formal semantics, which is necessary for the formal verification and validation of the system design. Thus, a map from these models on formal methods is needed. Synchronous languages are well known for their formal aspects and their richness in terms of tools for verification, analysis and code distribution. Therefore, the connection of these two technologies is encouraged because it offers the opportunity to benefit from the capabilities of GASPARD in the specification of data-parallelism and also from the power of formal aspects of synchronous languages.

This report presents how MDE transformations contribute to bridge the two technologies through crossing the different abstraction levels from GASPARD to synchronous languages (LUSTRE [13] is considered here for illustration).

1.4 Outline

In section 2, a brief introduction to the background and our proposed approach is given. The background includes the data-parallelism, particularly GASPARD co-modeling environment, and synchronous languages, particularly LUSTRE. It is followed by the proposed synchronous equational metamodel in section 3. This metamodel is dedicated to the specification of data-parallelism for synchronous languages. Model transformations and some implementation examples are then detailed in section 4. These transformations are regrouped into a transformation chain and some examples of matrix and image processing are showed. After, some general validation discussions and implemented examples (section 5) are presented. Related works are discovered in the section 6. Finally the conclusions and some perspectives are drawn in the last section.

2 Background and proposed approach

2.1 Data-parallel application design: GASPARD

GASPARD has a core formalism for specifying data-parallel applications, called ARRAY-OL, can be found in [6]. It provides a metamodel for the modeling of these applications according to MDE, and also a UML profile for the visualize the design of these applications.

2.1.1 An overview of GASPARD

This paper only addresses software application modeling and its deployment. A summarized grammar on software applicationis given according to the core formalism and the metamodel, which is easy to understand in spite of the complexity of the overall metamodel.

$$
\begin{array}{lcll}
Task & ::= & < Interface; Body > & (r1) \\
Interface & ::= & < in, out : \{Port\} > & (r2) \\
Port & ::= & < type; size > & (r3) \\
Body & ::= & Task^h \mid Task^r \mid Task^e & (r4) \\
Task^e & ::= & < some\ function\ call > & (r5) \\
Task^r & ::= & < t_i : \{Tiler\}; (\mathbf{r}, Task); t_o : \{Tiler\} > & (r6) \\
Tiler & ::= & < F; \mathbf{o}; P > & (r7) \\
Task^h & ::= & < \{Task\}; \{(Task, array, Task)\} > & (r8)
\end{array}
$$

All GASPARD tasks share common features (rule $(r1)$): an *interface* (rule $(r2)$ where `{}` denotes a set) that specifies input/output ports (typed by `in` or `out` in rule $(r2)$ and defined in rule $(r3)$) from which each task respectively receives and produces multidimensional arrays; and a *body* (rule $(r4)$), which depends on the type of task as follows:

- *Elementary task* (rule $(r5)$). The body corresponds to an atomic computation block. Typically, it consists of a function or an IP.

- *Repetitive task* (rule $(r6)$). It expresses the data-parallelism in a task. The *instances* of the associated repeated task are assumed to be independent and schedulable following any order, even in parallel. The attribute $\mathbf{r}$ (in the rule $(r6)$) denotes the *repetition space*, which indicates the number of repetitions. It is defined itself as a multidimensional array with a shape. Each dimension of this repetition space can be seen as a parallel loop and the shape of the repetition space gives the bounds of the loop indices of the nested parallel loops [6]. In addition, each task instance consumes and produces sub-arrays with the same shape. These sub-arrays are referred to as *patterns*. The way patterns are constructed is defined via *tilers* (rule $(r7)$), which are associated with each array. A tiler extracts (resp. stores) patterns from (resp. in) an array based on certain information it contains: F: a *fitting* matrix (how array elements fill patterns); $\mathbf{o}$: the *origin* of the *reference pattern*; and P: a *paving* matrix (how patterns cover arrays).

- *Hierarchical task* (rule $(r8)$). It is represented by a hierarchical acyclic graph in which each node consists of a task, and edges are labeled by arrays exchanged between task nodes.

An *application* is a hierarchical task in which the top-level of the hierarchy is composed of a single task, which plays a similar role to the "main" in a C program.

2.1.2 The GASPARD metamodel

The GASPARD application metamodel is defined according to above basic concepts. The whole software application is modeled as an *ApplicationModel*, in which *ApplicationComponent*s model hierarchical tasks. Instances of other *ApplicationComponent*, called *ApplicationComponentInstance*, can be composed in it. These instances have *PortInstance*s.

*Connector*s are used to connect *PortInstance*s and/or *Port*s. Internal structures, such as *Elementary*, *Compound* and *Repetitive*, are defined in an *ApplicationComponent* according to its inside component instances.

- *Elementary* points out that the *ApplicationComponent* is an elementary task, which is a black box in GASPARD.
- *Repetitive* indicates that the *ApplicationComponent* is a repetitive task. The *Connector*s which connect *ApplicationComponent*'s ports and *PortInstance*s of its internal instance are *Tiler*s.
- *Compound* corresponds to a hierarchical task and expresses task parallelism. All the *ComponentInstance*s inside this component run in parallel.

2.1.3 A simple example

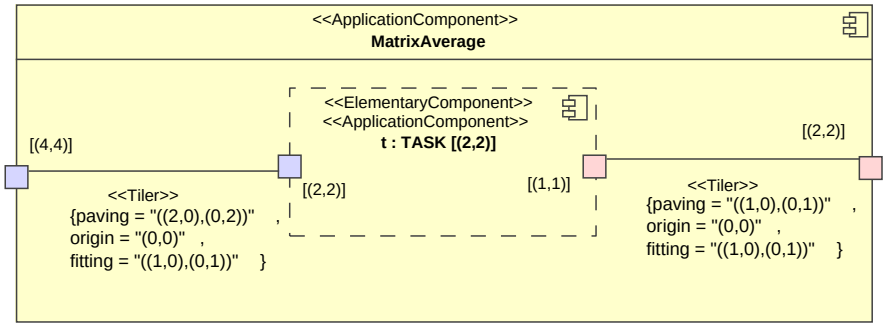

Figure 2: An example of matrix average

An example using GASPARD UML profile is given here (Fig. 2). The application illustrates the computation of the average of a (4∗4) matrix. The average function here indicates the computation of an average of the four elements stored in a (2∗2) matrix. The resulting average is then stored in a (1∗1) array. In Fig. 2, the *ApplicationComponent*, called *MatrixAverage*, is the main application, in which it contains an *ApplicationComponentInstance*, called *t*. *t* is an instance of another *ApplicationComponent*, called *TASK*, which is defined in the application elsewhere. *t* has *(2,2)* as its repetition space, which indicates it has four instances that execute in parallel. Ports are inputs and outputs of *ApplicationComponents*. Each *Port* has its own type and shape. In Fig. 2, only the shape of the ports are specified. The connectors between ports are *Tilers*, which contain tiling information. In this example,

the input array with a shape (4,4) is tiled into four (2,2) sub arrays, each of which is then taken as the input by a repetition of t. Each of these repetitions of t then produces an array of (1,1), from which an output array of application with shape (2,2) is then constructed.

2.2 The synchronous approach

The synchronous approach [3] proposes formal concepts that favor the trusted design of reactive embedded systems. Its basic assumption is that computation and communication are instantaneous, referred to as *synchrony hypothesis*. Concurrency is also available and well defined. Several remarkable characteristics of the specifications are reactivity, determinism, synchrony, and etc.. There are different styles of synchronous languages: data-flow languages, such as LUSTRE, LUCID SYNCHRONE and SIGNAL; imperative languages, such as ESTEREL. These languages have strong mathematical foundations that help to verify the design correctness. Hence, they are adopted in the critical system design. Among these languages, only data-flow languages are adopted in this report as they are suitable for data-flow-oriented applications.

In this report, LUSTRE is taken as the example (see a segment of LUSTRE code in Fig. 3) for the introduction of some basic concepts in synchronous languages. A *node* is a basic functionality unit in LUSTRE. Each node gives the same results with the same inputs because of its determinism. Nodes have modular declarations that enable their reuse. Each node has an *interface* (input at line (l1) and output at (l2)), local definition (l3), and *equation*s (line (l5) and (l6)). Variables are called *signal*s in LUSTRE. Equations are signal assignments. Furthermore only unique assignments are allowed for signals. In these equations, there are possibly node invocations (l5) that are declared outside this node. Obviously, in LUSTRE, modularity and hierarchy are inbuilt. The composition of these equations, denoted by ";", means their parallel execution w.r.t. data-dependencies. The node has the same meaning independently of the equation order.

```
node node_name (A1:int^4)              (l1)
 returns(A3:int^4);                    (l2)
 var A2:int^4;                         (l3)
 let                                   (l4)
    A2 = a_function(A1);               (l5)
    A3 = A1+A2;                        (l6)
 tel                                   (l7)
```

Figure 3: An example of LUSTRE code.

Synchronous languages provide, on one hand, inbuilt compilers to check the determinism and reactivity of the synchronous programs; on the other hand, associated tool-sets for the verification of the program correctness. As a result, these languages have been successfully used in several critical domains (e.g. avionics, automotive, nuclear power plants).

2.3 The proposed approach

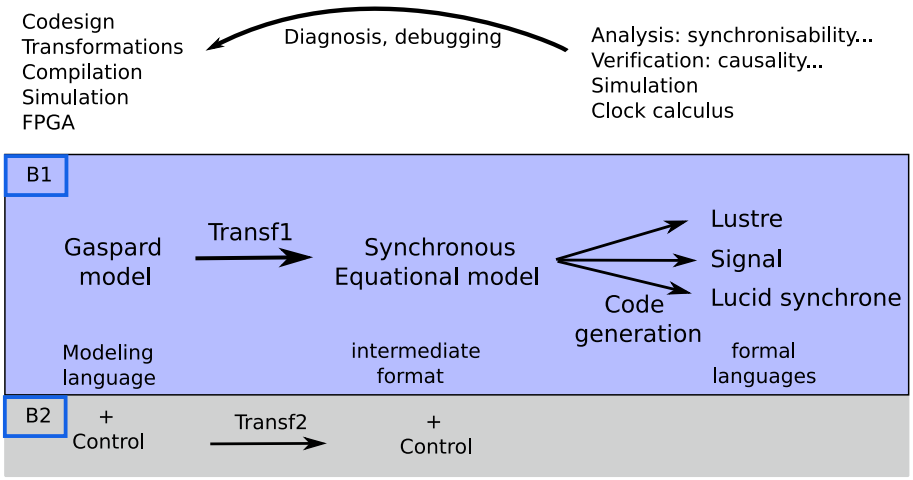

Figure 4: A global view of the proposed approach.

Fig. 4 illustrates the global view of our proposed approach. The involved transformations are located in box B1, which start from the GASPARD model. MDE transformations (*Transf1* in the figure) are then carried out on this GASPARD model into the synchronous model. The obtained model serves to generate the synchronous language code (*Code generation* in the figure).

With the help of the code and tools provided by synchronous languages, the application validation and design correctness verification are possible (the part on the top of box B1 in Fig. 4). For example, The generated code in LUSTRE, SIGNAL, or LUCID SYNCHRONE, can be used for various purposes: synchronizability analysis, causality verification, simulation, etc.. The results of the analysis and verification contribute to uncover the corresponding problems that are present in the original designs. The corrected designs can be transformed to other lower level languages for the purpose of simulation, performance evaluation, compilation, etc..

An ongoing work, which appears in box B2, is the integration of control in term of automata in GASPARD and current synchronous modeling. This will be discussed in section 5.1 and 7.

3 A synchronous equational metamodel

Synchronous data-flow languages have several common aspects. As a result, we proposed a synchronous modeling [12] that is common to these languages and enables the code generation for these languages altogether. It aims to be generic enough to target the synchronous

data-flow languages mentioned earlier and to be adequate to express data-parallel applications. So, it is not intended to have exactly the same expressivity as these languages. However it does not suffer from the complexity and particularity of target languages. The resulting models bridge the gap between data-parallel applications and data-flow languages. Their parallel compositions preserve the parallelism, and their modularity and re-usability ensure hierarchical compositions of original GASPARD models. Moreover it enables potential improvements, for instance, the integration of application control inspired by [20]. A synchronous equational metamodel is proposed in the following according to this modeling and is based on the previous basic concepts (in the subsection 2.2).

3.1 Signal

In the proposed metamodel, all input, output or local variables are called `Signals` (see Fig. 5). Each `Signal` is associated with a `SignalDeclaration`, which declares the name and type of the `Signal`. It is associated with at least one `SignalUsage`. The latter represents one operation on `Signal`. If the `Signal` is an array, a `SignalUsage` can be an operation on a pattern of this array. Hence, if the array has several patterns, the `Signal` is associated to the same number of `SignalUsage` correspondingly. Each of these `SignalUsages` has an `IndexValueSet`, which is a set of `IndexValue` of the associated `Signal`. An `IndexValue` indicates one array coordinate, which is composed of a list of integer numbers. A `SignalUsage` is associated with at least one `Argument` of equations, which indicates their inputs/outputs.

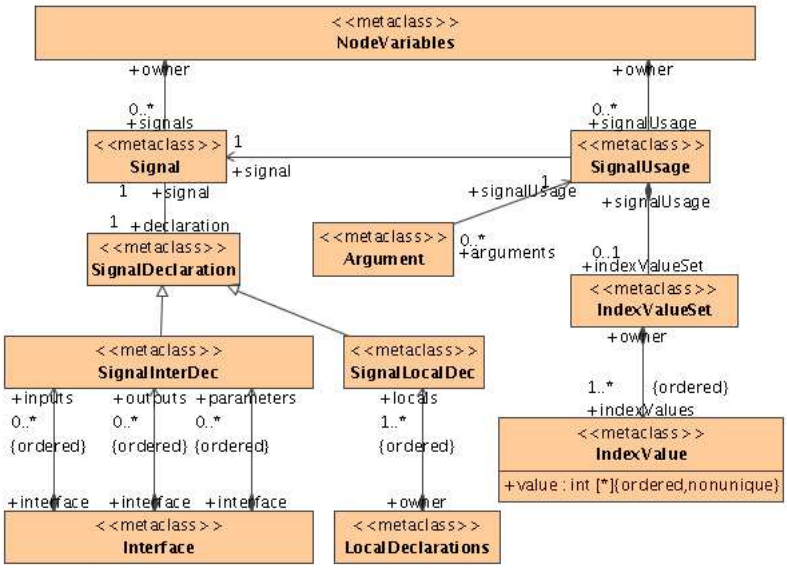

Figure 5: Extract of the synchronous metamodel: `Signal`.

3.2 Equation

`Equations` are functional units (see Fig. 6). They indicate relations between their inputs and outputs, which are called `Arguments` here. An `Equation` has an `EquationRightPart` and at most one `EquationLeftPart`. `EquationLeftPart` has ordered `Arguments` as `Equation` outputs. `EquationRightPart` is either an `ArrayAssignment` or an `Invocation`. `ArrayAssignment` has ordered `Arguments`, and indicates that the `Equation` is an array assignment, whereas an `Invocation` is a call to another `Node` (see 3.3). In an `Invocation`, `FunctionIdentifier` indicates the called function, and `Arguments` indicate parameters to be passed to the function.

3.3 Node

Synchronous functionalities are modeled as `Nodes` (see Fig. 6). A `Node` has no more than one `Interface`, `LocalDeclaration`, `NodeVariables`, an `EquationSystem` and some `Implementations` and `CodeFiles`. `NodeVariables` is the container of `Signals` and `SignalUsages`. Each input/output `Signal` is associated with a `SignalDeclaration`, which belongs to the `Interface`, while local `Signals`' `SignalDeclarations` belong to `LocalDeclaration`. `EquationSystem` is the node body that fulfills the functionality through a composition of at least one synchronous `Equations`.

3.4 Module

All `Nodes` are finally grouped in a `Module`, which represents the whole application. It contains one `Node` as the main `Node` of the application. Each `Node` is either defined in the `Module` or linked to an external function through IP deployment. `Nodes` that are not defined in the `Module` should be deployed. The equivalent of these nodes are GASPARD elementary tasks. An `Implementation` associated with a `Node` contains the information of the external function. Parameters of external function are represented by `PortImplementations`. Their order are defined in the `Implementation` so that parameters are passed correctly to the application. An `Implementation` is associated with at least one `CodeFile`, which represents the implementation of the external function.

4 Model transformations

Only GASPARD models with the infinite dimension at the highest hierarchy can be transformed into synchronous models. The infinite dimension is translated by a logical time in the reactive style of synchronous languages. So in synchronous models, signals have no more infinite dimensions. The multidimensional arrays are translated into array-type signals. Parallelism in GASPARD can be easily modeled in synchronous models with the help of the composition operator defined in synchronous languages.

Transformations of GASPARD model into synchronous specifications (typically, LUSTRE programs) consist of two steps (Fig. 7): firstly, a transformation of GASPARD model into

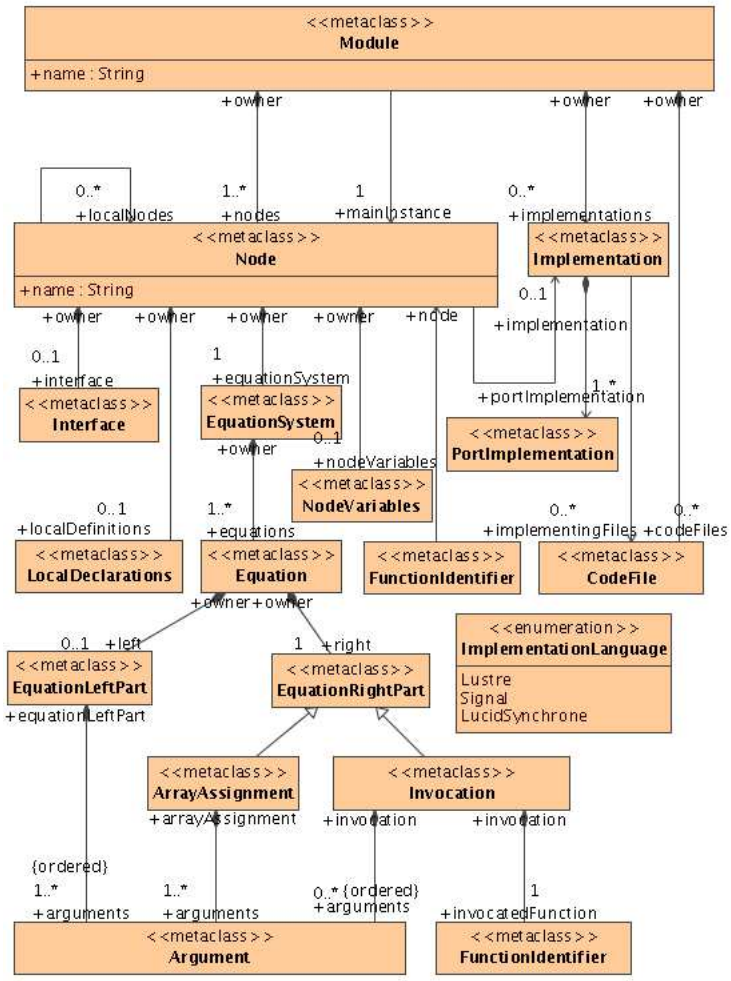

Figure 6: Extract of the synchronous metamodel: `Node`, `Equation`.

synchronous model; and then, the generation of synchronous code from the synchronous model obtained from the first step.

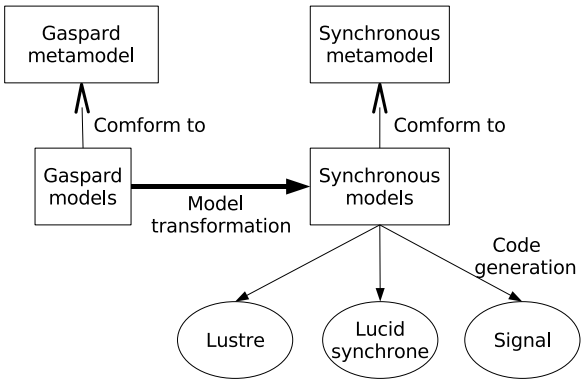

Figure 7: MDE based model transformation

4.1 From GASPARD model to synchronous equational model

Some basic transformations between two these models are first given. `Components` and `ComponentInstances` are transformed into `Nodes` and `Equations` respectively. `Ports`, `PortInstances` and `DefaultLink` connectors in a `Component` are transformed into `Signals`, whereas `Tiler` connectors are transformed into `Equations` as well as `Nodes`.

4.1.1 Transformation rules

The whole transformation can be represented through a tree structure (see Fig. 8). The unique initial (root) rule is *GModel2SModel*. It transforms a whole deployed GASPARD application into a synchronous module. This rule then calls its sub-rules: *GApplication2SNode*, *GTiler2SNode*, *GACI2SNode*, *GAConsumer2SNode*, *GAProducer2SNode*, *GCodeFile2SCodeFile*, *GASImpl2SImpl*,etc. *GApplication2SNode* has also three sub-rules: *GElementary2SEquationSystem*, *GCompound2SEquationSystem*, *GRepetitive2SEquationSystem*. Note that not all rules in the transformation are given in this report. In the following, only rules presented in the Fig. 8 are described. Among them, *GTiler2SNode* and *GApplication2SNode* are a little more detailed. The other rules are constructed in the same way.

- *GTiler2SNode* (see Fig. 9 in which each element is numbered). It is a rule for the transformation of `Tiler` connectors into synchronous input or output tiler `Node`. An input tiler `Node` is taken as an example for the construction of a synchronous node. First of all, the `Node` (numbered 1) is created and is associated with its `Module`.

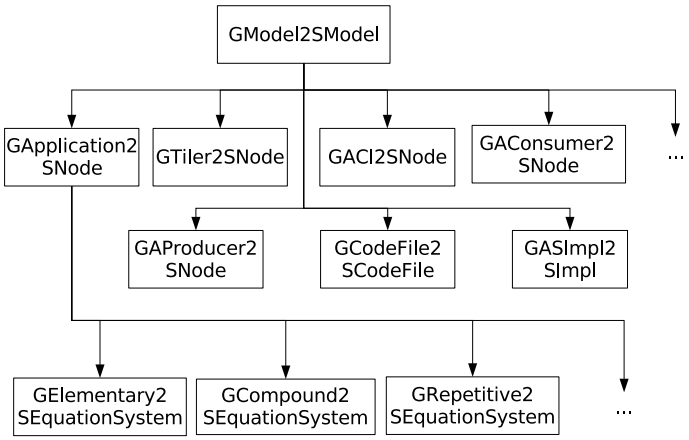

Figure 8: Hierarchy of the transformation rules

The `Port` and `PortInstance` connected by this tiler are then transformed into input and output `Signals` respectively. One `Port` corresponds to one input signal, and one `PortInstance` corresponds to several output signals, whose quantity, n, is calculated from the repetition space defined in its connected `ComponentInstance`. The input signal is associated with n `SignalUsages` (4) and an output signal are associated with a `SignalUsages` (8). `Interface` (2) is then created and associated with `SignalDeclarations` (3, 9) that are associated with signals. Note that there are no `LocalDeclarations` in this node. Next, an `EquationSystem` contains n `Equations` (5). In each `Equation`, the `EquationLeftPart` has an `Argument` (6) which is associated with a `SignalUsage` of an input signal. `EquationRightPart` is directly an `ArrayAssignment`. Its `Argument` (7) is associated with a `SignalUsage` (8) of a corresponding output.

- *GACI2SNode.* It transforms the unique main `ApplicationComponentInstances` into a synchronous `Node`. It is the main instance of the application.

- *GAConsumer2SNode.* It transforms an `ArrayConsumer` to a `Node`. `ArrayConsumer` is a special concept in GASPARD, which is used to model the acquirement of array from the environment, such as radar, sonar. The generated `Node` models the same action.

- *GAProducer2SNode.* It transforms an `ArrayProducer` to a `Node`. `ArrayProducer` is also a special concept in GASPARD, which is used to model the consummation of arrays, such as display screen. The generated `Node` models the same action.

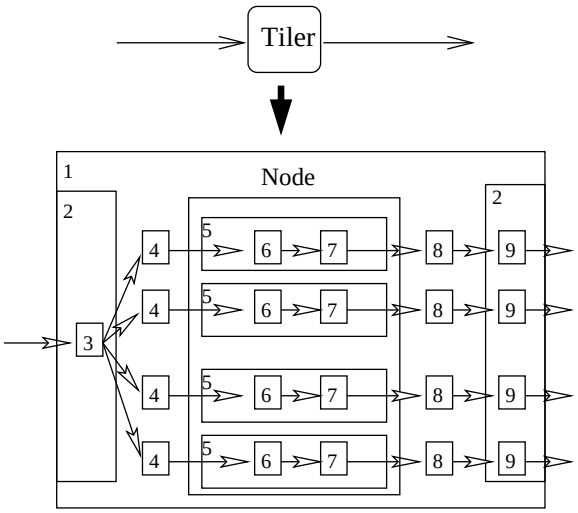

Figure 9: Transformation of the tiler.

- *GCodeFile2SCodeFile*. `CodeFile` represents the physical file of the related IP. This transformation simply copy fields in a GASPARD `CodeFile` to fields in a synchronous `CodeFile`.

- *GASImpl2SImpl*. It transforms an GASPARD `AbstractSoftwareImplementation` to a synchronous `Implementation`. According to the `AbstractSoftwareImplementation`, the transformation find the corresponding `SoftwareImplementation` and then copy its information to the synchronous `Implementation`.

- *GApplication2SNode*. It transforms application components into `Nodes`. However, all the elements in these `Nodes` are generated by its three sub-rules, which transform internal structures in the `Component` into an `EquationSystem`. This rule contain several sub-rules listed in the following:

  - *GRepetitive2SEquationSystem* (see Fig. 10). In this rule, an `EquationSystem` is first created. And then three types of `Equation` are created: input tiler `Equations`, repeated task `Equation` and output tiler `Equations`. `Tiler` connectors are transformed into input/output tiler `Equations`, which are invocations to `Nodes` generated by *GTiler2SNode*, and the internal `ComponentInstance` is transformed into repeated task `Equation`. A relevant repeated task `Node` is then created, in which n equations invoke the task node corresponding to the component that declares the internal component instance.

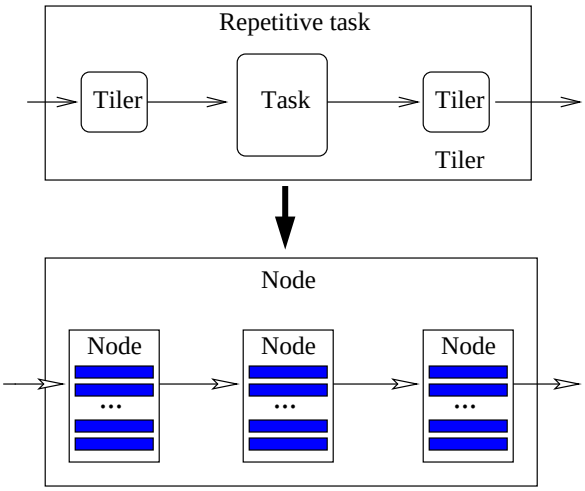

Figure 10: Transformation of the repetitive task

Note that hierarchical composition in GASPARD models is preserved in synchronous models by node invocations.

– *GCompound2SEquationSystem.* Each internal `ComponentInstance` is transformed into an equation. Connectors between these `ComponentInstances` are transformed into local `Signals`.

– *GElementary2SEquationSystem.* No `Equation` is created because its owner `Node` is implemented externally and `Deployment` models are used to import its external declarations. However an `Interface` is created according to the component's ports.

4.1.2 Illustration of a rule model

These transformation rules are always difficult to describe with natural languages. The complete explanation is very tedious and takes long time for understanding, whereas a curtailed one is always ambiguous for lack of details. It is also impossible to show the implemented code for demonstration. Hence a new language is needed for the efficient description of these transformations.

TRML [11] is a TRansformation Modeling Language, which offers a graphical representation of model transformations through:

- its UML profile,
- its portable notation which is independent from any existing transformation engines,

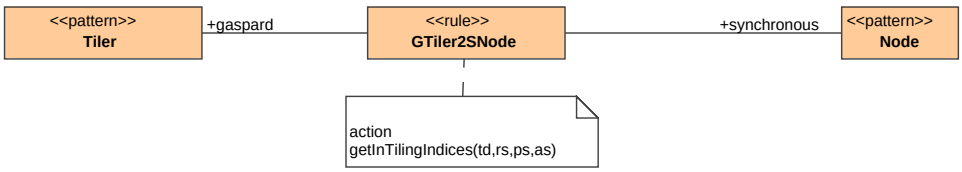

Figure 11: TRML patterns and rule

- its mechanisms to divide transformation into rules.

Furthermore a metamodel is provided for the modeling. Note that model elements are in bold and associations between them are in italics in the following explanation. In a TRML rule (Fig. 11), a transformation is divided into three parts: input `pattern` (`Tiler` in Fig. 11), `rule` (`GTiler2SNode`), and output `pattern` (`Node`). The input pattern indicates the set of model elements to be transformed, which are based on the input model concepts (indicated by *gaspard*, the association between `GTiler2SNode` and `Tiler`). Similarly, the output pattern indicates the set of model elements to be generated, which are based on output model concepts (*synchronous*). `Rule` takes the input pattern and transforms it into output pattern. Some external functions used in the transformation are showed in the `note` (annotation boxes with `action`). Note that TRML allowes the modeling of bidirectional transformations, but here, only one direction from GASPARD into synchronous is illustrated.

A typical transformation is illustrated with the help of TRML. The transformation of a GASPARD `Tiler` into a `Node` (Fig. 12), called `GTiler2SNode`, is detailed in the order of input pattern, `rule`, and output pattern. The root element of the `Input pattern` is a GASPARD `Tiler`. In the transformation, however, more model elements execpt `Tiler` are needed. These elements can then be found through the associations connected to this Tiler. The `TilingDescription` stores the tiler's `F, O, P` information, which can be found through the *tiling*. A tiler is connected to, on one hand, a `Port` through the *Source*, which indicates the input array of the application component; on the other hand, a `PortInstance` through the *Target*, which indicates the input pattern of the repetitions of the internal component. The port is connected to a `Shape` so as to indicate the array shape. The port instance is connected to `ComponentInstance` through the *componentInstance*, from which the shape of its port can be found. The port instance is also connected to `Part` by the `part`, from which the shape of repetition space can be found. Finally, from the association of `owner` of the tiler, the `ApplicationPart` and then `ApplicationModel` can be found.

`Rule` is the bridge between input and output patterns. The black box functions used by the transformation are specified in an annotation box that is linked to this rule, for instance, `getTilingIndices(td, rs, ps, as)` in the annotation box tagged with "action". This is a function implemented in Java that calculates the array indices from the tiling information. The arguments of the function come from the input pattern.

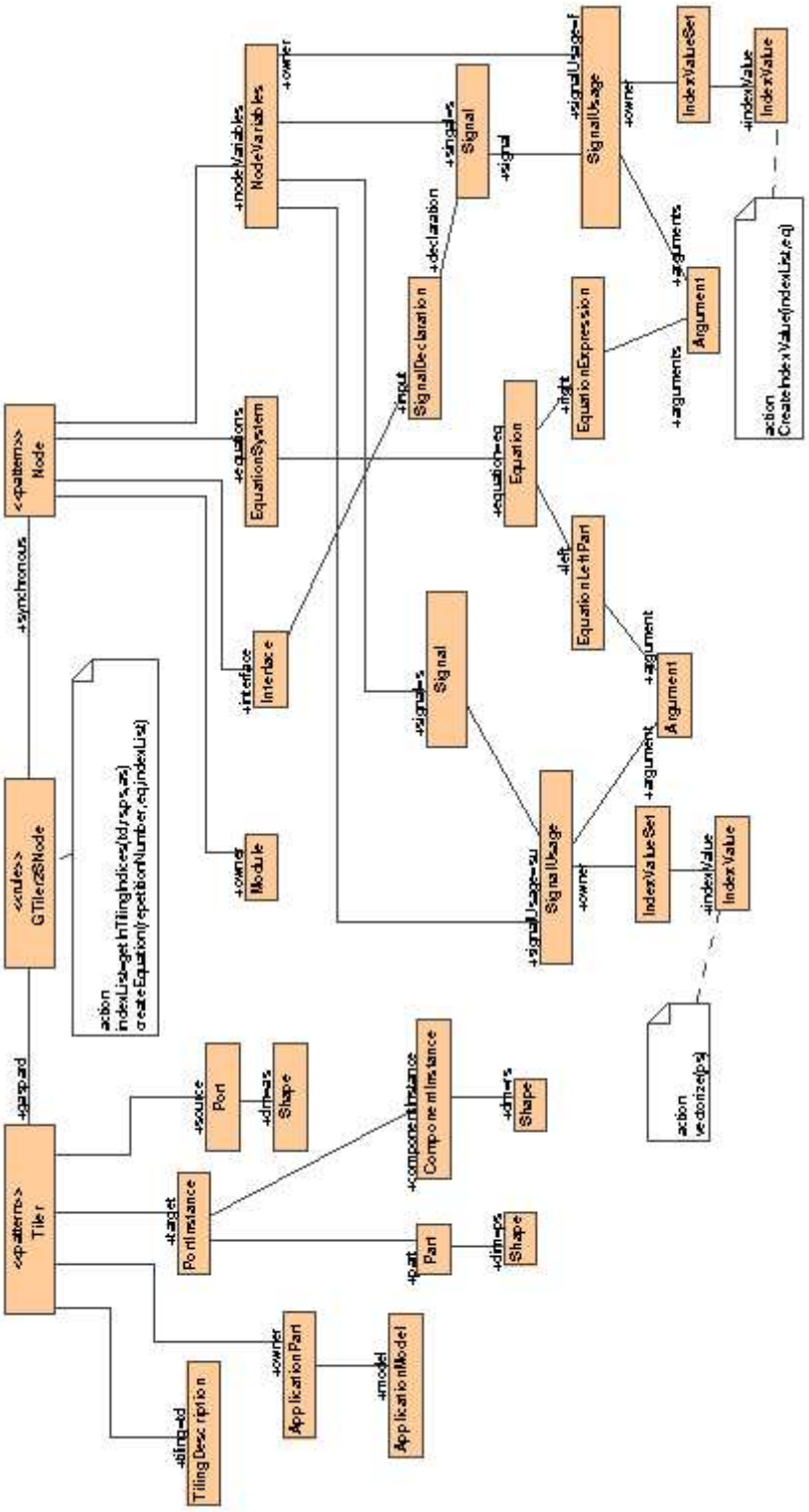

Figure 12: TRML representation of a tiler transformation

The root element of the `Output pattern` is a `Node` of the synchronous model. The node is associated with its owner, called `Module`, by the association *owner*. The `Module` can be found through the GASPARD `ApplicationModel` from the input pattern. All the elements required in the node are then associated to the node. For instance, `Interface`, `EquationSystem` and `NodeVariables` are associated to the node by *interface*, *equations* and *nodeVariables* respectively. Similarly, other elements are associated to `Interface`, `EquationSystem` and `NodeVariables` and so on. Some black box functions can be used during the creations of the model elements and their associations, such as `CreatIndexValue()`.

The transformation illustrated in the Fig. 12 is not a complete transformation because of the lack of expressivity of imperative aspects in TRML, for instance, the iterated creations of `Equation`, `Signal`, and `IndexValue` and the associations of the signals to their index values. Despite of this disadvantage, this graphical transformation language greatly helps to understand syntactic and certain semantic transformations between input/output models.

4.1.3 Implementation of a transformation chain

GASPARD models are specified in the graphical environment MAGICDRAW, and are exported as ECLIPSE Modeling Framework (EMF) [9] models. EMF is a modeling framework and code generation facility. In the following transformation phase, these models are transformed into EMF GASPARD models. These two previous transformations will not be detailed here. Then the EMF GASPARD model is transformed into the EMF synchronous equational model, which is finally used to generate synchronous language code (e.g. LUSTRE code). An automated model transformation chain (Fig. 13) is then defined through the concatenation of these transformations from MAGICDRAW UML models to data-flow languages.

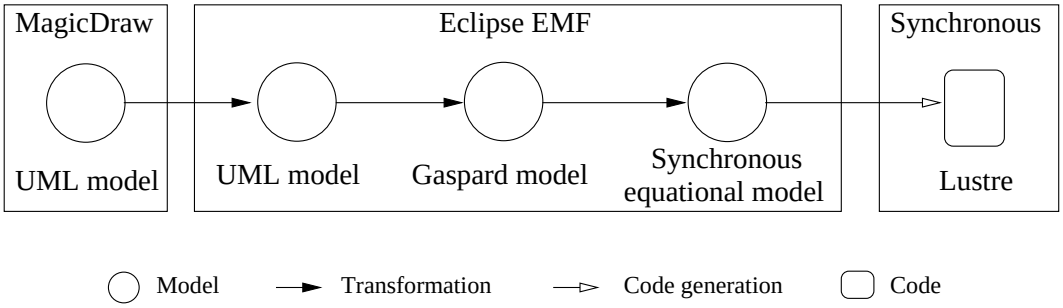

Figure 13: The detailed transformation chain.

These transformations were implemented with the help of specifications, standards and transformation languages. Some of them are briefly presented in this paper. MOF QVT [29] is the OMG standard on model query and transformation, which is respected in transformations presented here. Several other transformation languages and tools, such as ATL [15] and KERMETA [18] already exist. ATL is a model transformation language (a mixed style

of declarative and imperative constructions) designed w.r.t. QVT. KERMETA is a metaprogramming environment based on an object-oriented Domain Specific Language. But these two languages lack of extension capability specially when some external functions are needed to be integrated into the transformation. EMFT (Eclipse Modeling Framework Technology) project was initiated to develop new technologies that extend or complement EMF. Its query component offers capabilities to specify and execute queries against EMF model elements and their contents. The MOMOTE tool (MOdel to MOdel Transformation Engine), which is based on the EMFT QUERY and is integrated into GASPARD, is a JAVA framework that allows to perform model to model transformations. It is composed of an API and an engine. It takes input models that conform to some metamodels and produces output models that conform to other metamodels. A transformation by MOMOTE is composed of rules that may call sub-rules. These rules are integrated into an ECLIPSE plugin. In general, one plugin corresponds to one transformation. During model transformations, these plugins are automatically invoked one by one.

4.2 Synchronous code generation from the equational model

The implemented code generation (Fig. 14) from synchronous models is based on EMF JET (Java Emitter Templates) [10]. It is a generic template engine for the purpose of code generation. The JET templates are specified by using a JSP-like (JavaServer Pages) syntax and are used to generate JAVA implementation classes. Finally, these classes can be invoked to generate source code, such as SQL, XML, Java source code and LUSTRE (also SIGNAL and LUCID SYNCHRONE) in our case. MOCODE (MOdels to CODe Engine) is another GASPARD integrated tool, which works with JET for the code generation. MOCODE offers an API that reads the input models, and also an engine that recursively takes elements from input models and executes a corresponding JET JAVA implementation class on them.

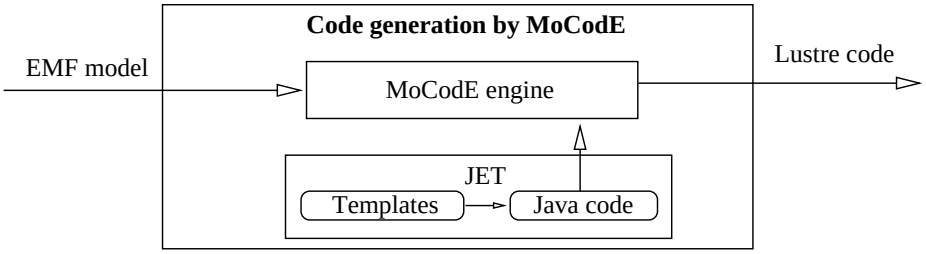

Figure 14: Generation of synchronous code from synchronous models.

4.3 Application examples

Some examples of matrix and image processing have been implemented through the proposed approach and transformation chain exposed in this paper.

4.3.1 Matrix average

Examples of matrix processing, which averages the patterns from inputs, are intuitive, but they are typical to show the transformation and the application domain. The specification of the chosen example in GASPARD and its explanation can be found in Fig. 2 in section 2. The deployment of the matrix average IP is illustrated in Fig. 15. This deployment indicates where to find the LUSTRE code that implements the matrix average computing. The physical LUSTRE code is represented by the `CodeFile`, and it is associated to the elementary task by the component `AbstractSoftwareImplementation`, which is composed of at least `SoftwareImplementations`. This means one elementary task may have several different implementations (in different languages or through different algorithms). The `SoftwareImplementation` contains the deployment information, for example, the elementary function name, the language of its implementation. Other deployment information, such as ports, etc., can be found in the associations (`portImplementedBy`, `implementedBy`) between `AbstractSoftwareImplementation` and `ElementaryTask`.

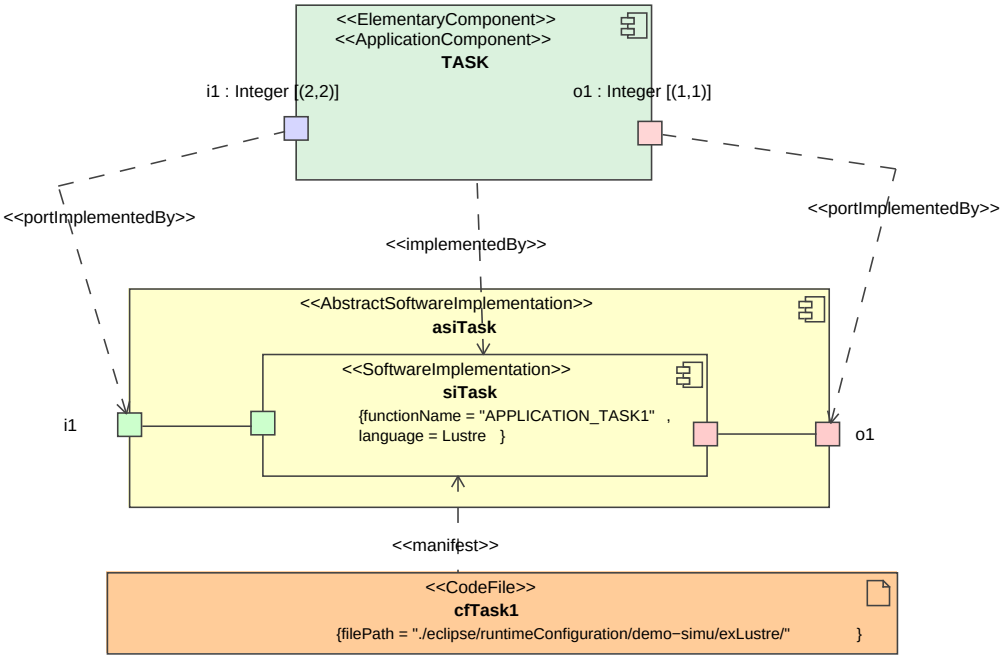

Figure 15: The deployment of the matrix average IP

The example and a video of the transformation chain is located in [16]. The code generated from the transformation is showed in Fig. 16. In this extract of the complete code,

`int^4^4` defines a integer array type with a shape (4,4). In the next, all the generated nodes are explained briefly .

- `APPLICATION_TASK1` is the imported LUSTRE function, which implements matrix average.
- `TASK_REPETITION` is the node in which the repetitions of `APPLICATION_TASK1` are built. Each repetition takes a pattern as input and another pattern as output.
- `TILER_INPUT` is the input tiler, which takes `IA` as input array, and produces four patterns: `TP0`, `TP1`, `TP2`, `TP3`. Note that all the index values are calculated by a black box function in the transformation.
- `TILER_OUTPUT` is the output tiler, which takes four patterns: `TP0`, `TP1`, `TP2`, `TP3`, and build an output array `OA`.
- `APPLICATION_MATRIX` is the main application, which invokes `TILER_INPUT_MATRIX`, `TILER_OUTPUT_MATRIX` and `TASK_REPETITION`.

4.3.2 Image downscaling

Besides the first simple example, which is easy to understand and illustrate, we have other more complicated examples, such as downscaler. In this report, a downscaler from a standard definition TV to a mobile phone screen display will be illustrated (Fig. 17).

In this example, a flow of (640∗480)-array will be taken as input (represented by the port with a shape [(640∗480)]), and a flow of (320,240)-array is then produced. In this application, the filter component, which contains horizontal and vertical filter, is repeated 80∗60 times at each instance. And for each repetition of this component, it takes a (8,8)-array as input and produces a (4,4)-array as output. The filter component can be decomposed into two filters, `Horizontal filter` and `Vertical filter`. Horizontal filter contains `Hfilter`, which is repeated 8 times. And each of the repetition takes a (8)-array (resp. (4)-array) as input (resp. output). This is a filter that works only on the first dimension of the array. Vertical filter contains `Vfilter`, which is repeated 4 times. And each of the repetition takes a (8)-array (resp. (4)-array) as input (resp. output). This is a filter that works only on the second dimension of the array.

The corresponding generated code can be found in Annex A. But not all code is illustrated because of the big size of the generated code.

```
node APPLICATION_TASK1 (I1:int^2^2)
 returns  (O1:int^1^1);
 let
  O1[0,0] =
    (I1[0,0]+I1[0,1]+I1[1,0]+I1[1,1])/4;
 tel

node TASK_REPETITION
 (PI0_0:int^2^2; PI1_0:int^2^2;
  PI2_0:int^2^2; PI3_0:int^2^2)
 returns
 (PO0_1:int^1^1; PO1_1:int^1^1;
  PO2_1:int^1^1; PO3_1:int^1^1);
 let
   PO0_1 = APPLICATION_TASK1(PI0_0);
   PO1_1 = APPLICATION_TASK1(PI1_0);
   PO2_1 = APPLICATION_TASK1(PI2_0);
   PO3_1 = APPLICATION_TASK1(PI3_0);
 tel

node TILER_INPUT
 (IA:int^4^4)
 returns (TP0:int^2^2; TP1:int^2^2;
          TP2:int^2^2; TP3:int^2^2);
 let
 (TP0[0,0],TP0[1,0],TP0[0,1],TP0[1,1])
   =(IA[0,0],IA[1,0],IA[0,1],IA[1,1]);
 (TP1[0,0],TP1[1,0],TP1[0,1],TP1[1,1])
   =(IA[2,0],IA[3,0],IA[2,1],IA[3,1]);
-- (continue in the next column)
 (TP2[0,0],TP2[1,0],TP2[0,1],TP2[1,1])
   =(IA[0,2],IA[1,2],IA[0,3],IA[1,3]);
 (TP3[0,0],TP3[1,0],TP3[0,1],TP3[1,1])
   =(IA[2,2],IA[3,2],IA[2,3],IA[3,3]);
 tel
node TILER_OUTPUT
 (TP0:int^1^1;TP1:int^1^1;
  TP2:int^1^1;TP3:int^1^1)
 returns  (OA:int^2^2);
 let
   OA[0,0] = TP0[0,0];
   OA[1,0] = TP1[0,0];
   OA[0,1] = TP2[0,0];
   OA[1,1] = TP3[0,0];
 tel

node APPLICATION_MATRIX (IA0:int^4^4)
 returns (OA1:int^2^2);
 var IP0:int^2^2; IP1:int^2^2;
     IP2:int^2^2; IP3:int^2^2;
     OP0:int^1^1; OP1:int^1^1;
     OP2:int^1^1; OP3:int^1^1;
 let
  (IP0,IP1,IP2,IP3) =
    TILER_INPUT(IA0);
  OA1 =
   TILER_OUTPUT(OP0,OP1,OP2,OP3);
  (OP0,OP1,OP2,OP3) =
    TASK_REPETITION(IP0,IP1,IP2,IP3);
 tel
```

Figure 16: Generated Lustre code.

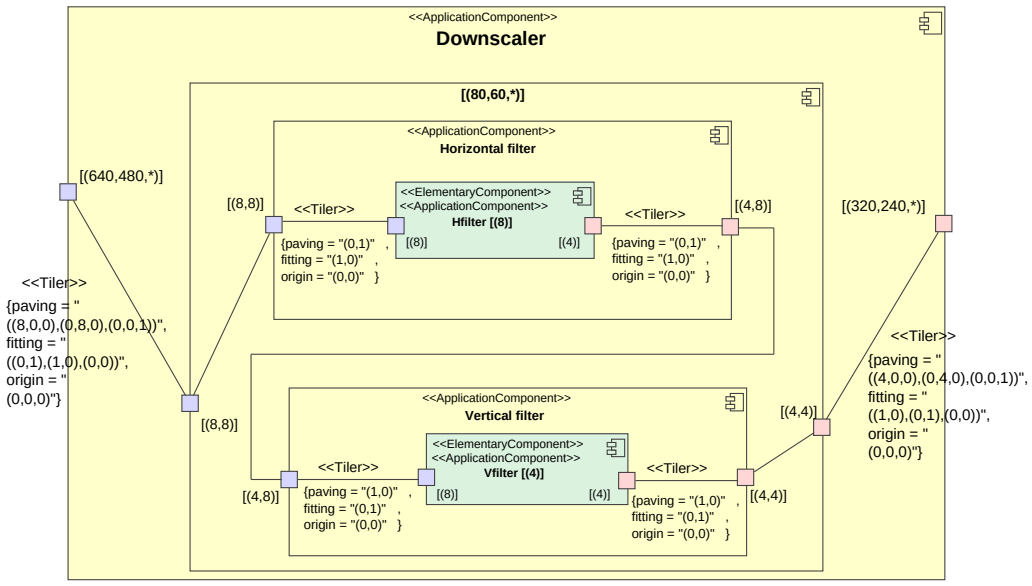

Figure 17: The downscaler

5 Design validation based on synchronous languages

5.1 General issues

As mentioned before, the intention of the connection between the two previous technologies, namely GASPARD and synchronous languages, is to benefit from the validation tools provided by the synchronous approach for the correct design of data-parallel applications.

The verification and analysis offered by synchronous languages can be divided into two sub-domains: functional and non-functional. Functional validation concerns only application (w.r.t. hardware architecture and environment etc.) issues. Safe signal (such as array) assignment and data-dependency analysis are considered as pure functional validation. However, control (such as automata, control clock etc.) verification and simulation have both functional and non-functional aspects. Functional control and simulation are the focus of our work, but the non-functional ones are also worth being studied.

- **Safe array assignment** is carried out on GASPARD array values. The core language of GASPARD imposes several constraints on its arrays, such as single assignment. Single assignment indicates that no data element is ever written twice but it can be read several times. This constraint can be easily checked by compilers of LUSTRE and SIGNAL. Another concern is the partial initialization of GASPARD arrays that may cause problems when the non-initialized array elements are used later. LUSTRE and

SIGNAL provide different way to address such an issue. LUSTRE imposes complete initialization that the compiler rejects the programs with partially initialized arrays, whereas SIGNAL fills the non-initialized array elements with default values. Obviously, default values may change the execution results when they are involved in the array computing.

- **Data-dependency analysis** includes the causality analysis in synchronous languages. Causality is caused by the self dependence under the instantaneous semantics of synchronous languages. In GASPARD, self dependence may cause deadlock. Two approaches of causality detection are distinguished according to the different mechanisms defined in LUSTRE and in SIGNAL. In LUSTRE, specifications are analyzed by the compiler syntactically, and those that have potential causality issues are rejected by the compiler. In SIGNAL, the compiler considers as well a clock analysis to determine the causality, which provides a finer analysis than that of LUSTRE. This analysis helps to check the existence of deadlocks specified in GASPARD. The deadlocks are not always obvious to be avoided because the specified hierarchy and/or components probably conceal the potential real data dependency.

- **Simulation** enables both functional and non-functional verification, performance analysis, etc.. Functional simulation allows the verification of the application correctness through its execution. Both LUSTRE and SIGNAL provide simulators. Typical examples of the simulation for GASPARD are image processing, such as rotation, filtering, etc.. Together with an image display tool (an ongoing work dedicated to image processing in GASPARD environment), the processing results can be shown directly. Certain non-functional simulation is also studied in SIGNAL language, such as the performance evaluation for temporal validation [19]. Temporal information is associated to the SIGNAL program, from which an approximation of the program execution time can be calculated.

- **Clock** analysis is another important validation issue. Although non-functional clock does not represent hardware architecture and the execution environment directly, it is logically related to it. Clock relations specified in the system to some extent reflect the relations that exist between the components in hardware architecture and its environment. These relations can be specified in synchronous languages. LUSTRE compiler can check the coherence between these clocks. Whereas in SIGNAL, a more complex clock system is built by its compiler on which the verification is carried out. The latter is called SIGNAL *clock calculus*. With the specified clock relations, some problems, such as clock synchronisability, can be checked. Other non-functional characteristics, such as serialized model (w.r.t. the parallel model), can also be implemented. Details of the clock related analysis can be found in [12].

- **Control** introduced in GASPARD offers the opportunity to enable the flexibility of the application/architecture design. However this flexibility may lead to the undepend-

ability of the system, such as safety, liveness, schedulability, etc.. Software test may be a solution to verify the system dependability, but they are time-consuming and can not give a complete guarantee. And their test results relies greatly on the chosen scenario/strategy. Model checking is considered as one of key solutions to this issue as the evolution of this technology significantly reduces the difficulty of its usage and the time that it costs.

Control can be specified in several forms, such as automata and clock etc., in synchronous languages. In particular, automata based functional control is our first choice because of not only its clear definition of syntax and semantics, but its adoption in synchronous languages. In spite of STATECHARTS [14], ARGOS [23] and SYNCCHARTS [2], etc., we are concentrated in Mode-automata, which we can find in MATOU [24], LUCID SYNCHRONE and SIGNAL etc. Both LUSTRE and SIGNAL provide model-checking tools, which are called LESAR [32] and SIGALI [4] respectively. Another interesting technology is the controller synthesis [25, 26]. It is based on the same principle as model checking, except that it is intended to add control to the system execution to exclude potential bad executions.

As mentioned above, the synchronous languages provide the possibilities to verify GASPARD designs without real implementations on SoC. This verification can be classified into two categories: GASPARD related verification and application related verification. GASPARD related verification aims at finding errors in the user's original design that do not conform to the GASPARD specifications. Hence, the single assignment and the causality analysis belong to this category. However there exist some others designs that comply with GASPARD specifications without providing correct or expected results, i.e., they do not conform to application specifications, such as safety issues and non-functional requirements. Simulation, model checking and clock analysis then belong to this category.

In the next, some examples of verification concerning deadlock detection and simulation are presented.

5.2 Deadlock detection

Currently, the check of absence of dependency cycles in GASPARD specifications, which is still not available in any GASPARD tools, can be carried out with the help of synchronous language compilers. We have experimented causality analysis with different LUSTRE programs that are generated automatically from GASPARD models in order to verify the absence of dependency cycles which lead to deadlocks in the specifications.

A simple analysis. GASPARD imposes deadlock free by construction. A counterexample is then given in Fig. 18. In this example, the `Task` is a hierarchical component, in which two sub-tasks, `t1:Task1` and `t2:Task2`, execute in parallel. In this hierarchical level, a deadlock, t1.o2 → t2.i2 → t2.o2 → t1.i2 → t1.o2 (→ means "is needed by", and t1.o2 signifies *port* o2 of *task* t1) can be found easily. Obviously, this application is not allowed in GASPARD because it only adopts deadlock-free specifications.

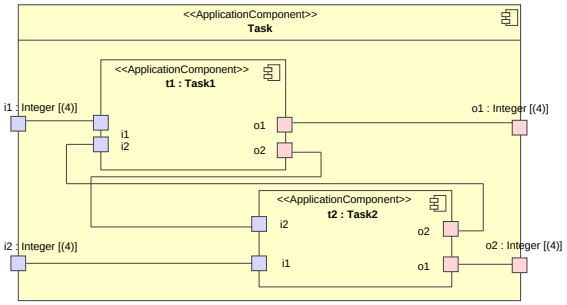

Figure 18: Causality analysis

A finer analysis. The previous case is however too strict for some applications where this type of deadlock is not a real one, i.e., the coarse granularity of analysis hides some real dependencies between components. An example of these applications is illustrated in Fig. 19. *Task1* and *Task2* in the previous application are illustrated in a finer grain manner. If a new analysis is carried out on this example, then the deadlock previously found is not preserved because the dependencies of t1.i2 → t1.o2 and t2.i2 → t2.o2 do not exit any more.

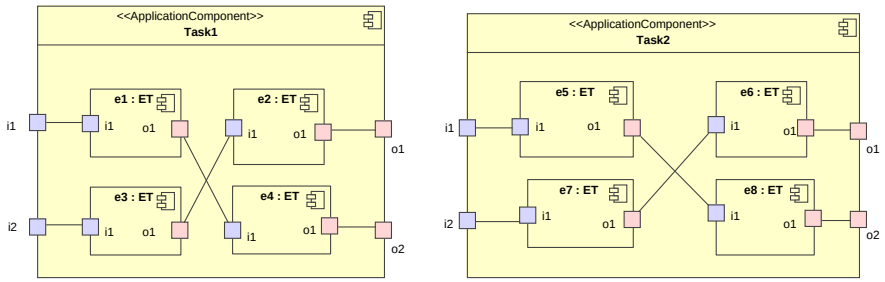

Figure 19: Causality analysis

5.3 Simulation

In order to verify the functional correctness of applications, the functional simulators are then used. The simulation through the graphical simulator SIMEC [22], which is distributed in the LUSTRE environment, is illustrated here. The previously mentioned GASPARD application of the matrix average (Fig. 2) illustrates the simulation (see Fig. 20).

This figure can be divided into 3 parts: The left one at the top of the figure (box B1) represents the input data of the simulation, which is array of the shape (4,4). So it has 16 elements, whose names start from `INPUT_ARRAY_0_0_0` to `INPUT_ARRAY_0_3_3` (their index

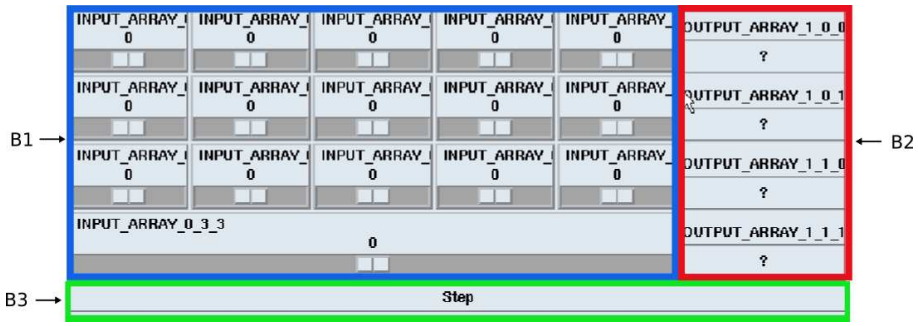

Figure 20: Simulation of the matrix average

values are concealed because of the small window size). The right part at the top of the figure (box B2) represents the output array with the shape of (2,2), whose name start from `OUTPUT_ARRAY_1_0_0` to `OUTPUT_ARRAY_1_1_1`. The button at the bottom of the figure (box B3), called *Step*, enables the advancement of the simulation step by step.

6 Related works

There exists another synchronous metamodel, SIGNALMETA [7], which does not have the same idea as the one presented here. It represents a language oriented modeling specifically dedicated to the SIGNAL language. This metamodel completely defines all programming concepts of SIGNAL. It has been specified in the *Generic Modeling Environment* (GME), developed at Vanderbilt University. Users can visualize the application modeling with the help of this model in GME from which code can be generated. But the model presented in this report act as an intermediate model transformed automatically from the original GASPARD model. Then this model is used to generate synchronous code according to user's choice. A second difference is that our model does not have the same expressivity as that of SIGNALMETA. SIGNALMETA has the same expressivity as the SIGNAL language, but out model has less expressivity that is adequate to data-parallel applications. For instance, the model do not have complex clock mechanism. A model transformation from our model into SIGNALMETA model is possible.

Our work aims at the control oriented formal verification at an application level, whereas several other studies on the verification of data parallelism through formal methods [31] have been carried out. These studies focused on some fundamental properties of data parallel paradigm, and they usually adopted some small core languages with the basic primitives of data parallel languages. The first one [21] was based on the functional language, which studies the integration of data parallelism with declarative language. Through the defined functional language, semantics of some parallel data types and the operations on them were studied, and some laws for the transformations of programs and data were proved, etc. The

second study [5] was based on the assertional approach. A new language L, which has the common control structures of some data parallel languages, was defined, and a proof system was built on it. The weakest preconditions calculus was then defined and the associated definability property was discussed.

Other works on the projection of GASPARD on formal computation models have been studied, for instance, the simulation of GASPARD specifications in PTOLEMY II [8] and also the projection of GASPARD applications into Kahn process network for the distributed execution [1]. These studies show the practicability of the transformation from GASPARD data-parallelism specifications to other formal models, and also help to some extent to reduce the complexity of the original parallel specifications. But these works did not aim at formal validation issues nor was implemented by using a MDE approach compared to the work presented in this paper.

7 Conclusions and perspectives

In this paper, we proposed a synchronous metamodel and presented model transformations from data-intensive applications specified in GASPARD into synchronous languages, particularly the LUSTRE language, through a MDE approach. The implemented code in JAVA and transformation rules adds up to about five thousands lines of code in ECLIPSE. Code generators for other languages, such as SIGNAL and LUCID SYNCHRONE are still under developing.

While the previously presented transformations and validations are directly carried out without any modifications of original GASPARD models, some others can also be envisaged. In general, it requires the introduction of particular concepts in GASPARD, such as clocks, their relations and "indicators for parallel or sequential mapping". Synchronizability analysis and serialized models will then be available correspondingly. Hence a special version of GASPARD with these concepts will be studied in order to automatically generate the well-suited synchronous code.

Another future work concerns the integration of control notion inspired by [20] in GASPARD and the transformation of such notions in synchronous languages. This will benefit the flexibility of GASPARD. The model checking on controlled applications will be carried out then for the verification . The controller synthesis is another similar work that enables the correct control of the application.

Finally, the way all these analysis results can be exploited by GASPARD users is a challenging perspective from a practical point of view.

References

[1] A. Amar, P. Boulet, and P. Dumont. Projection of the array-ol specification language onto the kahn process network computation model. In *Proceedings of the International*

Symposium on Parallel Architectures, Algorithms, and Networks, Las Vegas, Nevada, USA, December 2005.

[2] Charles André. Computing SyncCharts Reactions. *Electronic Notes in Theoretical Computer Science*, 88:3–19, October 2004. http://www.sciencedirect.com; doi:10.1016/j.entcs.2003.05.007.

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

[4] L. Besnard, H. Marchand, and E. Rutten. The SIGALI Tool Box Environment. Workshop on Discrete Event Systems, WODES'06, July 2006.

[5] Luc Bougé, David Cachera, Yann Le Guyadec, Gil Utard, and Bernard Virot. Formal validation of data parallel programs: Introducting the assertional approach. In *The Data Parallel Programming Model*, pages 252–281, 1996.

[6] P. Boulet. Array-OL revisited, multidimensional intensive signal processing specification. Research Report RR-6113, INRIA, http://hal.inria.fr/inria-00128840/en/, February 2007.

[7] C. Brunette, J.-P. Talpin, L. Besnard, and T. Gautier. Modeling multi-clocked dataflow programs using the Generic Modeling Environment. In *Synchronous Languages, Applications, and Programming*. Elsevier, March 2006.

[8] P. Dumont and P. Boulet. Another multidimensional synchronous dataflow: Simulating ARRAY-OL in PTOLEMY II. Technical Report 5516, INRIA, France, March 2005. available at `www.inria.fr/rrrt/rr-5516.html`.

[9] Eclipse. Eclipse Modeling Framework (EMF). http://www.eclipse.org/emf.

[10] Eclipse. EMFT JET. http://www.eclipse.org/emft/projects/jet.

[11] A. Etien, C. Dumoulin, , and E. Renaux. Towards a Unified Notation to Represent Model Transformation. Research Report 6187, INRIA, 05 2007.

[12] A. Gamatié, E. Rutten, H. Yu, P. Boulet, and J.-L. Dekeyser. Synchronous Modeling of Data Intensive Applications. Research Report RR-5876, INRIA, http://hal.inria.fr/inria-00001216/en, 2006.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9), September 1991.

[14] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[15] INRIA Atlas Project. ATL. http://modelware.inria.fr/rubrique12.html.

[16] INRIA DaRT Project. Demos: GASPARD2 to LUSTRE. http://www2.lifl.fr/west/DaRTShortPresentations.

[17] INRIA DaRT Project. GASPARD. http://www.lifl.fr/west/gaspard/.

[18] INRIA Triskell Project. KERMETA. http://www.kermeta.org/.

[19] A. Kountouris and P. Le Guernic. Profiling of SIGNAL programs and its application in the timing evaluation of design implementations. In *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*, pages 6/1–6/9, Bristol, UK, February 1996. HP Labs.

[20] O. Labbani, J.-L. Dekeyser, P. Boulet, and E. Rutten. *Advances in Design and Specification Languages for SoCs, Selected contributions from FDL'06*, chapter UML2 Profile for Modeling Controlled Data Parallel Applications. Springer, 2007.

[21] Björn Lisper. Data parallelism and functional programming. In *The Data Parallel Programming Model*, pages 220–251, 1996.

[22] LUSTRE.

[23] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, (27):61–92, 2001.

[24] F. Maraninchi, Y. Rémond, and Y. Raoul. MATOU: An Implementation of Mode-Automata into DC. In *Compiler Construction*, Berlin (Germany), March 2000. Springer verlag.

[25] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of Discrete-Event Controllers based on the SIGNAL Environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325–346, October 2000.

[26] H. Marchand, E. Rutten, M. Le Borgne, and M. Samaan. Formal Verification of programs specified with SIGNAL : Application to a Power Transformer Station Controller. *Science of Computer Programming*, 41(1):85–104, August 2001.

[27] Model-Driven Architecture (MDA). http://www.omg.org/mda.

[28] Planet MDE. Model driven engineering. http://planetmde.org.

[29] Object Management Group. MOF Query / Views / Transformations. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01, Nov. 2005.

[30] Object Management Group (OMG). http://www.omg.org.

[31] Guy-René Perrin and Alain Darte, editors. *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, volume 1132 of *Lecture Notes in Computer Science*. Springer, 1996.

[32] Ch. Ratel. *Définition et réalisation d'un outil de vérification Formelle de Programmes* LUSTRE *: Le système LESAR*. PhD thesis, UJF, 1992.

[33] L. Rioux, T. Saunier, S. Gerard, A. Radermacher, R. de Simone, T. Gautier, Y. Sorel, J. Forget, J.-L. Dekeyser, A. Cuccuru, C. Dumoulin, , and C. Andre. MARTE: A new profile RFP for the modeling and analysis of real-time embedded systems. In *UML-SoC'05, DAC 2005 Workshop UML for SoC Design*, Anaheim, CA, June 2005.

[34] D.-C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), 2006.

A Generated code for the downscaler

The generated code could be found here.

```
node TILER_INPUT_VERTICALFILTER_1 (Tiler_Input_Array:int^4^8)
 returns
      (Tiler_Pattern_0:int^8;Tiler_Pattern_1:int^8;
      Tiler_Pattern_2:int^8;Tiler_Pattern_3:int^8);
 let
      (Tiler_Pattern_0[0],...,Tiler_Pattern_0[7])
      = (Tiler_Input_Array[0,0],...,Tiler_Input_Array[0,7]);

      (Tiler_Pattern_1[0],...,Tiler_Pattern_1[7])
      = (Tiler_Input_Array[1,0],...,Tiler_Input_Array[1,7]);

      ...

      (Tiler_Pattern_3[0],...,Tiler_Pattern_3[7])
      = (Tiler_Input_Array[3,0],...,Tiler_Input_Array[3,7]);
tel

node TILER_OUTPUT_VERTICALFILTER_2 (Tiler_Pattern_0:int^4;
      Tiler_Pattern_1:int^4;Tiler_Pattern_2:int^4;
      Tiler_Pattern_3:int^4;Tiler_Pattern_4:int^4;
      Tiler_Pattern_5:int^4;Tiler_Pattern_6:int^4;
      Tiler_Pattern_7:int^4)
 returns
      (Output_Array:int^4^4);
 let
      (Output_Array[0,0], Output_Array[0,3]) =
        (Tiler_Pattern_0[0],...,Tiler_Pattern_0[3]);
      ...
      (Output_Array[3,0], Output_Array[3,3]) =
        (Tiler_Pattern_7[0],...,Tiler_Pattern_7[3]);
tel

node TILER_INPUT_DOWNSCALER_3 (Tiler_Input_Array:int^640^480)
returns
      (Tiler_Pattern_0:int^8^8,
      ...
      Tiler_Pattern_479:int^8^8);
let
      (Tiler_Pattern_0[0,0],Tiler_Pattern_0[1,0],
```

```
Tiler_Pattern_0[2,0],Tiler_Pattern_0[3,0],
Tiler_Pattern_0[4,0],Tiler_Pattern_0[5,0],
Tiler_Pattern_0[6,0],Tiler_Pattern_0[7,0],
Tiler_Pattern_0[0,1],Tiler_Pattern_0[1,1],
Tiler_Pattern_0[2,1],Tiler_Pattern_0[3,1],
Tiler_Pattern_0[4,1],Tiler_Pattern_0[5,1],
Tiler_Pattern_0[6,1],Tiler_Pattern_0[7,1],
Tiler_Pattern_0[0,2],Tiler_Pattern_0[1,2],
Tiler_Pattern_0[2,2],Tiler_Pattern_0[3,2],
Tiler_Pattern_0[4,2],Tiler_Pattern_0[5,2],
Tiler_Pattern_0[6,2],Tiler_Pattern_0[7,2],
Tiler_Pattern_0[0,3],Tiler_Pattern_0[1,3],
Tiler_Pattern_0[2,3],Tiler_Pattern_0[3,3],
Tiler_Pattern_0[4,3],Tiler_Pattern_0[5,3],
Tiler_Pattern_0[6,3],Tiler_Pattern_0[7,3],
Tiler_Pattern_0[0,4],Tiler_Pattern_0[1,4],
Tiler_Pattern_0[2,4],Tiler_Pattern_0[3,4],
Tiler_Pattern_0[4,4],Tiler_Pattern_0[5,4],
Tiler_Pattern_0[6,4],Tiler_Pattern_0[7,4],
Tiler_Pattern_0[0,5],Tiler_Pattern_0[1,5],
Tiler_Pattern_0[2,5],Tiler_Pattern_0[3,5],
Tiler_Pattern_0[4,5],Tiler_Pattern_0[5,5],
Tiler_Pattern_0[6,5],Tiler_Pattern_0[7,5],
Tiler_Pattern_0[0,6],Tiler_Pattern_0[1,6],
Tiler_Pattern_0[2,6],Tiler_Pattern_0[3,6],
Tiler_Pattern_0[4,6],Tiler_Pattern_0[5,6],
Tiler_Pattern_0[6,6],Tiler_Pattern_0[7,6],
Tiler_Pattern_0[0,7],Tiler_Pattern_0[1,7],
Tiler_Pattern_0[2,7],Tiler_Pattern_0[3,7],
Tiler_Pattern_0[4,7],Tiler_Pattern_0[5,7],
Tiler_Pattern_0[6,7],Tiler_Pattern_0[7,7]) =
(Tiler_Input_Array[0,0],Tiler_Input_Array[0,1],
Tiler_Input_Array[0,2],Tiler_Input_Array[0,3],
Tiler_Input_Array[0,4],Tiler_Input_Array[0,5],
Tiler_Input_Array[0,6],Tiler_Input_Array[0,7],
Tiler_Input_Array[1,0],Tiler_Input_Array[1,1],
Tiler_Input_Array[1,2],Tiler_Input_Array[1,3],
Tiler_Input_Array[1,4],Tiler_Input_Array[1,5],
Tiler_Input_Array[1,6],Tiler_Input_Array[1,7],
Tiler_Input_Array[2,0],Tiler_Input_Array[2,1],
Tiler_Input_Array[2,2],Tiler_Input_Array[2,3],
Tiler_Input_Array[2,4],Tiler_Input_Array[2,5],
```

```
      Tiler_Input_Array[2,6],Tiler_Input_Array[2,7],
      Tiler_Input_Array[3,0],Tiler_Input_Array[3,1],
      Tiler_Input_Array[3,2],Tiler_Input_Array[3,3],
      Tiler_Input_Array[3,4],Tiler_Input_Array[3,5],
      Tiler_Input_Array[3,6],Tiler_Input_Array[3,7],
      Tiler_Input_Array[4,0],Tiler_Input_Array[4,1],
      Tiler_Input_Array[4,2],Tiler_Input_Array[4,3],
      Tiler_Input_Array[4,4],Tiler_Input_Array[4,5],
      Tiler_Input_Array[4,6],Tiler_Input_Array[4,7],
      Tiler_Input_Array[5,0],Tiler_Input_Array[5,1],
      Tiler_Input_Array[5,2],Tiler_Input_Array[5,3],
      Tiler_Input_Array[5,4],Tiler_Input_Array[5,5],
      Tiler_Input_Array[5,6],Tiler_Input_Array[5,7],
      Tiler_Input_Array[6,0],Tiler_Input_Array[6,1],
      Tiler_Input_Array[6,2],Tiler_Input_Array[6,3],
      Tiler_Input_Array[6,4],Tiler_Input_Array[6,5],
      Tiler_Input_Array[6,6],Tiler_Input_Array[6,7],
      Tiler_Input_Array[7,0],Tiler_Input_Array[7,1],
      Tiler_Input_Array[7,2],Tiler_Input_Array[7,3],
      Tiler_Input_Array[7,4],Tiler_Input_Array[7,5],
      Tiler_Input_Array[7,6],Tiler_Input_Array[7,7]);
      ...
tel

node TILER_OUTPUT_DOWNSCALER_4 (Tiler_Pattern_0:int^4^4,
      ..., Tiler_Pattern_479:int^4^4, )
 returns
      (Output_Array:int^320^240);
let
      (Output_Array[0,0],Output_Array[1,0],
      Output_Array[2,0],Output_Array[3,0],
      Output_Array[0,1],Output_Array[1,1],
      Output_Array[2,1],Output_Array[3,1],
      Output_Array[0,2],Output_Array[1,2],
      Output_Array[2,2],Output_Array[3,2],
      Output_Array[0,3],Output_Array[1,3],
      Output_Array[2,3],Output_Array[3,3]) =
        (Tiler_Pattern_0[0,0],Tiler_Pattern_0[1,0],
        Tiler_Pattern_0[2,0],Tiler_Pattern_0[3,0],
        Tiler_Pattern_0[0,1],Tiler_Pattern_0[1,1],
        Tiler_Pattern_0[2,1],Tiler_Pattern_0[3,1],
        Tiler_Pattern_0[0,2],Tiler_Pattern_0[1,2],
```

```
        Tiler_Pattern_0[2,2],Tiler_Pattern_0[3,2],
        Tiler_Pattern_0[0,3],Tiler_Pattern_0[1,3],
        Tiler_Pattern_0[2,3],Tiler_Pattern_0[3,3]);
      ...
tel

node TILER_OUTPUT_HORIZONTALFILTER_5 (Tiler_Pattern_0:int^4;
      Tiler_Pattern_1:int^4;Tiler_Pattern_2:int^4;
      Tiler_Pattern_3:int^4;Tiler_Pattern_4:int^4;
      Tiler_Pattern_5:int^4;Tiler_Pattern_6:int^4;
      Tiler_Pattern_7:int^4)
returns
      (Output_Array:int^4^8);
let
      (Output_Array[0,0],...Output_Array[3,0]) =
        (Tiler_Pattern_0[0],Tiler_Pattern_0[1],
        Tiler_Pattern_0[2],Tiler_Pattern_0[3]);

      ...
      (Output_Array[0,7],...Output_Array[3,7]) =
        (Tiler_Pattern_7[0],Tiler_Pattern_7[1],
        Tiler_Pattern_7[2],Tiler_Pattern_7[3]);
 tel

node TILER_INPUT_HORIZONTALFILTER_6 (Tiler_Input_Array:int^8^8)
returns
      (Tiler_Pattern_0:int^8;Tiler_Pattern_1:int^8;
      Tiler_Pattern_2:int^8;Tiler_Pattern_3:int^8;
      Tiler_Pattern_4:int^8;Tiler_Pattern_5:int^8;
      Tiler_Pattern_6:int^8;Tiler_Pattern_7:int^8);
let
      (Tiler_Pattern_0[0],...,Tiler_Pattern_0[7]) =
      (Tiler_Input_Array[0,0],...,Tiler_Input_Array[7,0]);
      ...
      (Tiler_Pattern_7[0],...,Tiler_Pattern_7[7]) =
      (Tiler_Input_Array[0,7],...,Tiler_Input_Array[7,7]);

tel

node APP_VFILTER (VFILTERINPUT_ARRAY_0:int^8)
 returns
      (VFILTEROUTPUT_ARRAY_1:int^4);
```

```
 let
      (VFILTEROUTPUT_ARRAY_1) =
        INVOCATION_VFILTER(VFILTERINPUT_ARRAY_0);
 tel

node APP_HFILTER (HFILTERINPUT_ARRAY_0:int^8)
 returns      (HFILTEROUTPUT_ARRAY_1:int^4);
 let
      (HFILTEROUTPUT_ARRAY_1) =
        INVOCATION_HFILTER(HFILTERINPUT_ARRAY_0);
 tel

node APP_VERTICALFILTER (INPUT_ARRAY_0:int^4^8)
 returns      (OUTPUT_ARRAY_1:int^4^4);
 var
      TILER_IN_0_PATTERN_0:int^8;
      ...
      TILER_IN_0_PATTERN_7:int^8;
      TILER_OUT_1_PATTERN_0:int^4;
      ...
      TILER_OUT_1_PATTERN_7:int^4;
let
      (TILER_IN_0_PATTERN_0,TILER_IN_0_PATTERN_1,
      TILER_IN_0_PATTERN_2,TILER_IN_0_PATTERN_3,
      TILER_IN_0_PATTERN_4,TILER_IN_0_PATTERN_5,
      TILER_IN_0_PATTERN_6,TILER_IN_0_PATTERN_7) =
        TILER_INPUT_VERTICALFILTER_1(INPUT_ARRAY_0);

      (OUTPUT_ARRAY_1) =
        TILER_OUTPUT_VERTICALFILTER_2(
        TILER_OUT_1_PATTERN_0,TILER_OUT_1_PATTERN_1,
        TILER_OUT_1_PATTERN_2,TILER_OUT_1_PATTERN_3,
        TILER_OUT_1_PATTERN_4,TILER_OUT_1_PATTERN_5,
        TILER_OUT_1_PATTERN_6,TILER_OUT_1_PATTERN_7);

      (TILER_OUT_1_PATTERN_0,TILER_OUT_1_PATTERN_1,
      TILER_OUT_1_PATTERN_2,TILER_OUT_1_PATTERN_3,
      TILER_OUT_1_PATTERN_4,TILER_OUT_1_PATTERN_5,
      TILER_OUT_1_PATTERN_6,TILER_OUT_1_PATTERN_7) =
        TASK_REPETITION_VFILTER(
        TILER_IN_0_PATTERN_0,TILER_IN_0_PATTERN_1,
        TILER_IN_0_PATTERN_2,TILER_IN_0_PATTERN_3,
```

```
        TILER_IN_0_PATTERN_4,TILER_IN_0_PATTERN_5,
        TILER_IN_0_PATTERN_6,TILER_IN_0_PATTERN_7);
tel

node TASK_REPETITION_VFILTER(PATTERN_IN_0_0:int^8;
      PATTERN_IN_1_0:int^8; PATTERN_IN_2_0:int^8;
      PATTERN_IN_3_0:int^8; PATTERN_IN_4_0:int^8;
      PATTERN_IN_5_0:int^8; PATTERN_IN_6_0:int^8;
      PATTERN_IN_7_0:int^8)
returns
      (PATTERN_OUT_0_1:int^4;PATTERN_OUT_1_1:int^4;
      PATTERN_OUT_2_1:int^4; PATTERN_OUT_3_1:int^4;
      PATTERN_OUT_4_1:int^4; PATTERN_OUT_5_1:int^4;
      PATTERN_OUT_6_1:int^4; PATTERN_OUT_7_1:int^4);
 let
      (PATTERN_OUT_0_1) = APP_VFILTER(PATTERN_IN_0_0);
      (PATTERN_OUT_1_1) = APP_VFILTER(PATTERN_IN_1_0);
      (PATTERN_OUT_2_1) = APP_VFILTER(PATTERN_IN_2_0);
      (PATTERN_OUT_3_1) = APP_VFILTER(PATTERN_IN_3_0);
      (PATTERN_OUT_4_1) = APP_VFILTER(PATTERN_IN_4_0);
      (PATTERN_OUT_5_1) = APP_VFILTER(PATTERN_IN_5_0);
      (PATTERN_OUT_6_1) = APP_VFILTER(PATTERN_IN_6_0);
      (PATTERN_OUT_7_1) = APP_VFILTER(PATTERN_IN_7_0);
 tel

node APP_HORIZONTALFILTER (INPUT_ARRAY_0:int^8^8)
 returns (OUTPUT_ARRAY_1:int^4^8);
 var
      TILER_IN_0_PATTERN_0:int^8; TILER_IN_0_PATTERN_1:int^8;
      TILER_IN_0_PATTERN_2:int^8; TILER_IN_0_PATTERN_3:int^8;
      TILER_IN_0_PATTERN_4:int^8; TILER_IN_0_PATTERN_5:int^8;
      TILER_IN_0_PATTERN_6:int^8; TILER_IN_0_PATTERN_7:int^8;
      TILER_OUT_1_PATTERN_0:int^4;TILER_OUT_1_PATTERN_1:int^4;
      TILER_OUT_1_PATTERN_2:int^4;TILER_OUT_1_PATTERN_3:int^4;
      TILER_OUT_1_PATTERN_4:int^4;TILER_OUT_1_PATTERN_5:int^4;
      TILER_OUT_1_PATTERN_6:int^4;TILER_OUT_1_PATTERN_7:int^4;
 let
      (TILER_IN_0_PATTERN_0,TILER_IN_0_PATTERN_1,
      TILER_IN_0_PATTERN_2,TILER_IN_0_PATTERN_3,
      TILER_IN_0_PATTERN_4,TILER_IN_0_PATTERN_5,
      TILER_IN_0_PATTERN_6,TILER_IN_0_PATTERN_7) =
        TILER_INPUT_HORIZONTALFILTER_6(INPUT_ARRAY_0);
```

```
      (OUTPUT_ARRAY_1) =
        TILER_OUTPUT_HORIZONTALFILTER_5
        (TILER_OUT_1_PATTERN_0, TILER_OUT_1_PATTERN_1,
        TILER_OUT_1_PATTERN_2,TILER_OUT_1_PATTERN_3,
        TILER_OUT_1_PATTERN_4, TILER_OUT_1_PATTERN_5,
        TILER_OUT_1_PATTERN_6, TILER_OUT_1_PATTERN_7);

      (TILER_OUT_1_PATTERN_0,TILER_OUT_1_PATTERN_1,
      TILER_OUT_1_PATTERN_2,TILER_OUT_1_PATTERN_3,
      TILER_OUT_1_PATTERN_4,TILER_OUT_1_PATTERN_5,
      TILER_OUT_1_PATTERN_6,TILER_OUT_1_PATTERN_7) =
        TASK_REPETITION_HFILTER(
        TILER_IN_0_PATTERN_0, TILER_IN_0_PATTERN_1,
        TILER_IN_0_PATTERN_2, TILER_IN_0_PATTERN_3,
        TILER_IN_0_PATTERN_4, TILER_IN_0_PATTERN_5,
        TILER_IN_0_PATTERN_6, TILER_IN_0_PATTERN_7);
 tel

node TASK_REPETITION_HFILTER (PATTERN_IN_0_0:int^8;
      PATTERN_IN_1_0:int^8;PATTERN_IN_2_0:int^8;
      PATTERN_IN_3_0:int^8;PATTERN_IN_4_0:int^8;
      PATTERN_IN_5_0:int^8;PATTERN_IN_6_0:int^8;
      PATTERN_IN_7_0:int^8)
 returns
      (PATTERN_OUT_0_1:int^4;PATTERN_OUT_1_1:int^4;
      PATTERN_OUT_2_1:int^4;PATTERN_OUT_3_1:int^4;
      PATTERN_OUT_4_1:int^4;PATTERN_OUT_5_1:int^4;
      PATTERN_OUT_6_1:int^4;PATTERN_OUT_7_1:int^4);
 let
      (PATTERN_OUT_0_1) = APP_HFILTER(PATTERN_IN_0_0);
      (PATTERN_OUT_1_1) = APP_HFILTER(PATTERN_IN_1_0);
      (PATTERN_OUT_2_1) = APP_HFILTER(PATTERN_IN_2_0);
      (PATTERN_OUT_3_1) = APP_HFILTER(PATTERN_IN_3_0);
      (PATTERN_OUT_4_1) = APP_HFILTER(PATTERN_IN_4_0);
      (PATTERN_OUT_5_1) = APP_HFILTER(PATTERN_IN_5_0);
      (PATTERN_OUT_6_1) = APP_HFILTER(PATTERN_IN_6_0);
      (PATTERN_OUT_7_1) = APP_HFILTER(PATTERN_IN_7_0);
 tel

node APP_DOWNSCALER (INPUT_ARRAY_1:int^640^480)
 returns
```

```
      (OUTPUT_ARRAY_0:int^320^240);
 var
      TILER_OUT_0_PATTERN_0:int^4^4;
      ...
      TILER_OUT_0_PATTERN_478:int^4^4;
      TILER_IN_1_PATTERN_0:int^8^8;
      ...
      TILER_IN_1_PATTERN_478:int^8^8;
 let
      (OUTPUT_ARRAY_0) =
        TILER_OUTPUT_DOWNSCALER_4(TILER_OUT_0_PATTERN_0);

      (TILER_IN_1_PATTERN_0,...,TILER_IN_1_PATTERN_478) =
        TILER_INPUT_DOWNSCALER_3(INPUT_ARRAY_1);

      (TILER_OUT_0_PATTERN_0,...,TILER_OUT_0_PATTERN_478) =
        TASK_REPETITION_HVFILTER(
        TILER_IN_1_PATTERN_0,...,TILER_IN_1_PATTERN_478);
 tel

node TASK_REPETITION_HVFILTER (PATTERN_IN_0_1:int^8^8,
      ..., PATTERN_IN_0_478:int^8^8)
 returns
      (PATTERN_OUT_0_0:int^4^4,...,PATTERN_OUT_0_478:int^4^4);
 let
      (PATTERN_OUT_0_0) = APP_HVFILTER(PATTERN_IN_0_1);
      ...
      (PATTERN_OUT_0_478) = APP_HVFILTER(PATTERN_IN_0_478);
 tel


node APP_REPDSMAIN (APP_REPDSMAIN_INPUT_ARRAY_0:int^640^480)
 returns
      (APP_REPDSMAIN_OUTPUT_ARRAY_1:int^320^240);
 let
      (APP_REPDSMAIN_OUTPUT_ARRAY_1) =
        APP_DOWNSCALER(APP_REPDSMAIN_INPUT_ARRAY_0);
 tel

node APP_HVFILTER (APP_HVFILTER_INPUT_ARRAY_0:int^8^8)
 returns
      (APP_HVFILTER_OUTPUT_ARRAY_1:int^4^4);
```

```
 var
      APP_HVFILTER_LOCAL_ARRAY_0:int^4^8;
 let
      (APP_HVFILTER_OUTPUT_ARRAY_1) =
        APP_VERTICALFILTER(APP_HVFILTER_LOCAL_ARRAY_0);

      (APP_HVFILTER_LOCAL_ARRAY_0) =
        APP_HORIZONTALFILTER(APP_HVFILTER_INPUT_ARRAY_0);
 tel

node MAIN_APPLICATION (APP_REPDSMAIN_INPUT_ARRAY_0:int^640^480)
 returns (APP_REPDSMAIN_OUTPUT_ARRAY_1:int^320^240);
 let
      (APP_REPDSMAIN_OUTPUT_ARRAY_1)=
        APP_REPDSMAIN(APP_REPDSMAIN_INPUT_ARRAY_0);
 tel
```

Unité de recherche INRIA Futurs
Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399