A Model for the Mixed-Design of Data-Intensive and Control-Oriented Embedded Systems

Abdoulaye Gamatié, Eric Rutten, Huafeng Yu

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Model for the Mixed-Design of Data-Intensive and Control-Oriented Embedded Systems

Abdoulaye Gamatié — Éric Rutten — Huafeng Yu

N° 6589

July 2008

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6589--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
LILLE - NORD EUROPE

A Model for the Mixed-Design of Data-Intensive and Control-Oriented Embedded Systems

Abdoulaye Gamatié* , Éric Rutten† , Huafeng Yu‡

Thème COM — Systèmes communicants
Équipes-Projets DaRT et Pop-Art

Rapport de recherche n° 6589 — July 2008 — 23 pages

Abstract: This paper presents a model and its semantics for the design of embedded systems that contain data-intensive parts such as multimedia applications, and require adaptivity w.r.t. criteria such as platform resources or quality of service (QoS). The proposed solution relies on a combination of: *i)* the repetitive model of computation dedicated to the design of high-performance embedded systems and *ii)* reactive control features based on finite state machines and modes. It is defined within a framework, called GASPARD2, that implements automatic transformations that lead to various target languages, e.g., synchronous languages, SystemC, VHDL. The new model offers the adequate expressive power to describe complex behaviors of high-performance embedded systems. It also reconciles execution models dedicated to regular computations and control-oriented models that rather lead to irregular computations.

Key-words: Repetitive MoC, reactive control, high-performance embedded systems, modes, design and modeling, GASPARD2

* CNRS/LIFL - INRIA Lille-Nord Europe, Cité Scientifique, Bât. A, 40 avenue Halley, 59650 Villeneuve d'Ascq, France, abdoulaye.gamatie@lifl.fr

† INRIA Rhône-Alpes, Inovallée – 655, avenue de l'Europe, Montbonnot 38334 Saint-Ismier cedex, FRANCE, eric.rutten@inria.fr

‡ INRIA Lille-Nord Europe/LIFL, Bât. A, 40 avenue Halley, 59650 Villeneuve d'Ascq, France, huafeng.yu@inria.fr

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley, 59650 Villeneuve d'Ascq
Téléphone : +33 3 59 57 78 00 — Télécopie : +33 3 59 57 78 50

Un modèle pour la conception mixte de systèmes embarqués combinant traitement intensif de données et contrôle

Résumé :

Ce rapport présente un modèle et sa sémantique pour la conception de systèmes embarqués contenant du traitement intensif de données (par exemple, les systèmes multimédia) et exigeant une adaptation par rapport à des critères tels que les ressources de plates-formes ou la qualité de service. La solution proposée ici repose sur une combinaison : *i*) d'un modèle de calcul répétitif dédié à la conception de systèmes embarqués à hautes performances et *ii*) des notions de contrôle réactif basées sur les machines à états finis et les modes. Elle est définie dans un cadre, appelé Gaspard, qui met en œuvre des transformations automatiques vers différents langages cibles comme les langages synchrones, SystemC ou VHDL. Le nouveau modèle offre un pouvoir expressif intéressant pour décrire des comportements complexes des systèmes visés. Il réconcilie également les modèles d'exécution dédiés aux calculs réguliers avec les modèles orientés contrôle qui induisent plutôt des calculs irréguliers.

Mots-clés : Modèle d'exécution répétitif, contrôle réactif, systèmes embarqués à hautes performances, modes, conception et modélisation, GASPARD2

1 Introduction

Today, it is easy to observe how modern embedded systems have become very sophisticated and have been requiring more and more computing resources. This is particularly true for *high-performance* systems, which massively adopt architecture paradigms with multiple processors or cores. The range of concerned application domains is wide: state-of-the-art multimedia applications such as high-definition digital television, medical imaging, biometric data processing, sonar, radar, etc. All these applications are characterized by *data-intensive* computations, which can be efficiently achieved with parallel implementations. On the other hand, embedded systems usually have specific requirements that must be imperatively dealt with during their design. Typically, for mission-critical systems, such as sonar or radar, the resource constraints (e.g. limited memory capacity or energy power) impose the ability to guarantee non functional properties. In addition, the reliability of these systems necessitates the verification of their functional properties. Further important requirements are *adaptivity* and *reconfigurability*, which are sometimes needed in order to make the systems flexible enough to execute w.r.t environment and platform constraints.

From the above observations, there is clearly a need of practical, well-founded frameworks enabling to suitably address all mentioned aspects about the design of embedded systems. Such frameworks should provide designers with adequate description *models* and development tools. Here, we present a model for the design of data-intensive and control-oriented embedded systems within a framework devoted to the development of high-performance *system-on-chip* (SoC).

High-performance system-on-chip. The increasing integration capacity of transistors on a single chip promotes the implementation of parallel architectures on-chip. As a result, in recent years, *multiprocessor system-on-chip* (MPSoC) has become mainstream for embedded systems with intensive parallel computations. They offer very interesting computational performances, while reducing power consumption. MPSoCs consist of platforms composed of several processing elements, memory and I/O components that are interconnected by an on-chip dedicated structure (e.g. see the Tile64 architecture of Tilera[1]). MPSoC-based design of embedded systems needs new development methodologies in order to reduce the complexity of design space exploration and to increase the productivity of engineers. One solution consists in considering high-level models that are expressive enough to describe all aspects of MPSoC systems, and associated automatic transformations that refine high-level descriptions into lower level ones. The resulting refined descriptions are usable for various purposes.

Our design environment, called GASPARD2 [16], exactly relies on this solution. It adopts the *model-driven engineering* (MDE) approach to implement the methodology illustrated in Figure 1. Here, an MPSoC system under design is modeled using the OMG standard profile dedicated to *Modeling and Analysis of Real-time and Embedded systems* (MARTE[2]). This profile extends UML with new concepts that can be used to model the software and hardware parts as well as the mapping of the former on the latter. Such models contain the useful information that enable to address different design aspects: parallelism, per-

[1] `http://www.tilera.com`.
[2] `www.omgmarte.org`.

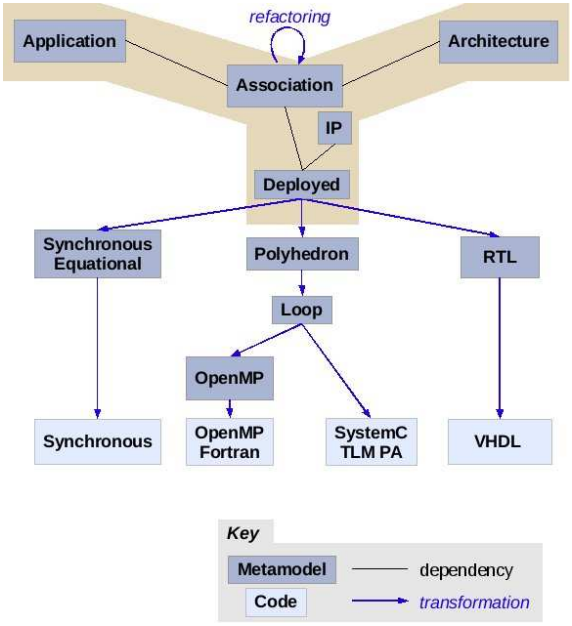

Figure 1: The GASPARD2 design methodology.

formance and scheduling, etc. In GASPARD2, the high-level models are refined towards specific technologies: synchronous languages [1] for formal validation, SystemC for simulation, OpenMP Fortran for execution and VHDL for circuitry synthesis. At each level of this refinement, the concepts are characterized by a dedicated *metamodel*, and the transitions from one level to another are obtained via automatic model transformations w.r.t. corresponding metamodels. The backbone environment that implements this methodology is ECLIPSE.

The design of MPSoC in GASPARD2 specifically relies on the *repetitive* model of computation (MoC) [3], which offers a very suitable way to express and manage different degrees of parallelism in a system. This MoC is inspired by ARRAY-OL [3,6], a mixed graphical-textual domain-specific language originally dedicated to intensive signal processing applications. It offers an elegant way to describe both *task parallelism* and *data parallelism* in the applications. Another major advantage of the repetitive MoC is the *regularity* of the structures and computations it describes. This feature is profitably exploited to define efficient algorithms and compilation techniques towards well-known high-performance architectures such as SIMD.

Control in high-performance computations. Let us consider a scenario of adaptation in a last generation cellular phone. Such a phone integrates video-streaming applications that provide the user with video-on-demand programs, or television broadcast. Such applications are data-intensive and often perform in different modes in order to fulfil their functionality according to various criteria: *Quality of service (QoS)* e.g., regarding image display, the following aspects are taken into account:

- *modes*: high *vs.* medium *vs.* low resolution, black and white *vs.* colour display, and compression level.
- *levels of quality*: an interesting feature is to have "graceful" degradation w.r.t. criteria from the multimedia application domain. These criteria have to decide whether black and white in high resolution is better than colour in medium resolution, or whether it is preferable to have average quality of images to avoid abrupt changing of quality, or to keep the highest quality.

Platform resource consumption. It can have quite different characteristics if several algorithm versions of the functionality are available, regarding: computing (in terms of WCET or CPU load), communication (switching on or off signal compression according to available bandwidth), memory footprint, access time between main and secondary memories, and energy.

The management of the above aspects leads to the definition of an *adaptation policy w.r.t. environment.* The general goal is *no matter how the environment changes, functionality must be fulfilled, at a good level.* One of the reacting modes in a phone is an incoming message, with attached images. If the receipt mode involves immediate download, then it could imply degraded video quality in order to release some resources. Otherwise, the receipt can be shut off by the user while looking at video uninterrupted and undisturbed, and resumed later. This latter point illustrates how knowledge of the application state allows for lookahead in the control.

The above example gives an idea of the adaptation criteria, quality levels on functionality, application-specific aspects and resource management policies that have to be dealt with during the design of high-performance embedded systems. It particularly motivates the need of design models combining two basic features: *i)* concepts that enable to express data-intensive computations and *ii)* concepts that offer a way to express the adaptivity w.r.t. various constraints such as QoS. This paper aims at answering this demand.

Our proposition. Our contribution is to enrich the repetitive MoC with new constructs that enable the expression of control between different modes of computation. An important challenge is to preserve the regularity of the repetitive MoC in presence of control concepts (which potentially lead to irregular computations) so as to continue to benefit from this feature for efficient implementations. We define an associated formal model, and in addition, we propose structured constructs that take into account hierarchy and composition in the control extension of GASPARD2 models. This extension is inspired by reactive mode automata [5,13] and improves the preliminary informal suggestion of [10]. The defined formal model is the semantic support for reasoning during the different transformations (which are not in the scope of this paper) supported by the framework illustrated in Figure 1. The presented structured control constructs offer a better expressivity than in [10] and enable complex yet very useful models as illustrated in Section 4.

The remainder of the paper is organized as follows: Section 2 introduces the basic design elements of the repetitive MoC by defining a corresponding behavioral semantics. Then, Section 3 presents the proposed extension enabling to define mixed data-intensive and control-oriented specifications. A few illustrative situations are given in Section 4, describing some adaptivity scenarios.

Section 5 discusses our solution w.r.t. existing works. Finally, concluding remarks are given in Section 6.

2 Repetitive structure modeling

We present the design concepts of GASPARD2 and we propose an associated behavioral semantics. Most of these concepts have been integrated in MARTE standard profile.

2.1 An overview of the concepts

The main data type manipulated in GASPARD2 is *multidimensional array*. Three kinds of tasks are distinguished: *elementary, repetitive* and *hierarchical* tasks. Let $\mathcal{T}$ denote the set of any of these kinds of tasks. The abstract grammar presented in Figure 2 describes the basic specification concepts of GASPARD2. By convention, the notation $x : X$ in the grammar means that X is the type of x, and $\{X\}$ denotes a set of elements typed X.

$Task$	$::=$	$Interface; Body$	$(r1)$
$Interface$	$::=$	$i, o : \{Port\}$	$(r2)$
$Port$	$::=$	$id; type; shape$	$(r3)$
$Body$	$::=$	$Body^h \mid Body^r \mid Body^e$	$(r4)$
$Body^e$	$::=$	$some\ function$	$(r5)$
$Body^r$	$::=$	$t_i, t_o : \{Tiler\}; (\mathbf{s_r}; Task); \{Ird\}$	$(r6)$
Ird	$::=$	$Connexion; \overrightarrow{d}; c_p$	$(r7)$
$Connexion$	$::=$	$p_i, p_o : Port$	$(r8)$
$Tiler$	$::=$	$Connexion; (F; \mathbf{o}; P)$	$(r9)$
$Body^h$	$::=$	$\{Task\}; \{Connexion\}$	$(r10)$

Figure 2: A grammar of GASPARD2 concepts.

All tasks share common features. They have the same global structure, as described in rule $(r1)$:

- an *interface* defined in rule $(r2)$ that specifies input and output *ports*, respectively represented by i and o. Ports are characterized in rule $(r3)$ by their *identifier* the *type* of received array elements, and the *shape* (i.e. dimension) of these arrays. We denote by $\mathcal{P}$ and $\mathcal{V}$ respectively the set of ports, and their associated value domain (i.e. set of arrays).
- a *body* (rule $(r4)$), which describes the function defined by the task.

The remaining rules are explained in section 2.2. Before going through their explanation, we have to note that there are several existing programming languages to deal with high performance, mostly for parallel scientific computing. The most popular is High-Performance Fortran [9], which manipulates multidimensional arrays and proposes parallel loop constructs and regular data distributions. More recent language proposals are StreamIt [18] and the high-productivity computing languages [12]: Chapel, Fortress and X10. The main objective of these languages is to facilitate the programming of next generation parallel systems and significantly increase the productivity of programmers. They are defined upon existing programming languages, e.g. Fortran for

Fortress and Java for X10. They natively support the control structures of their underlying languages.

While all above languages are dedicated to programming, the GASPARD2 formalism is rather devoted to high-level modeling. Via its tiling construct (see Section 2.2.2), it offers a very elegant and powerful abstraction level that allows one to describe the way manipulated data are accessed by computing elements. Then, the transformation chains implemented in its design environment automatically generate multi-target code from high-level models. Regarding all these aspects, the ALPHA language [20] is very close to GASPARD2. However, a notable difference is that ALPHA manipulates polyhedra instead of arrays. This leads to different specification styles.

2.2 Behavioral semantics

We first introduce some basic definitions inspired by [2], which will be used to define the GASPARD2 model.

Definition 1 (Environment)**.** Let $P \subset \mathcal{P}$ represents a set of ports, an environment ε associated with P is defined as a function $P \rightarrow \mathcal{V}$.

The set of environments associated with P is noted ε_P. A port (or a set of ports) p taking a value v in the execution environment ε is noted $p(v) \in \varepsilon$, or equivalently $\varepsilon(p) = v$.

Definition 2 (Environment composition)**.** Let $\varepsilon_1 \in \varepsilon_{P_1}$ and $\varepsilon_2 \in \varepsilon_{P_2}$ denote two environments. They are composable iff $\forall p \in P_1 \cap P_2$, $\varepsilon_1(p) = \varepsilon_2(p)$. Their composition, noted $\oplus$, is therefore as follows:

$$\begin{array}{rccc} \oplus : & \varepsilon_{P_1} \times \varepsilon_{P_2} & \rightarrow & \varepsilon_{P_1 \cup P_2} \\ & (\varepsilon_1, \varepsilon_2) & \mapsto & \varepsilon_1 \cup \varepsilon_2 \end{array}$$

The behavioral semantics of a task is given by a labelled transition system where the states consist of tasks as follows:

$$\frac{C}{T_1 \xrightarrow{\varepsilon} T_2}$$

where $T_1, T_2 \in \mathcal{T}$, ε denotes an execution environment of the tasks T_1, T_2, and C is a condition on T_1, T_2 and ε. The environment ε fixes the value of ports associated with T_1 and T_2 during a transition. The condition C must be satisfied in order to perform the transition between T_1 and T_2 according to ε. For any task $T \in \mathcal{T}$, we denote by $[\![T]\!]$ its corresponding semantics, i.e., the function that transforms its inputs into its outputs. For syntactical convenience, we use a "dot" notation to designate sub-parts of a concept according to the grammar of Figure 2, e.g., if I_1 denotes an interface, we write $I_1.i$ to designate its input ports.

2.2.1 Elementary tasks

An elementary task E (rule $(r5)$) informally consists of a function that is executed atomically. We adopt a graphical notation slightly simplified from [3], illustrated by Figure 3.

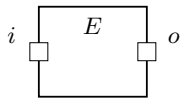

Figure 3: An elementary task.

Definition 3 (Elementary task)**.** Let E be an elementary task. Its behavioral semantics within an environment ε is as follows:

$$\frac{\varepsilon(o) = \phi(\varepsilon(i))}{E \xrightarrow{\varepsilon} E}$$

where $\phi = [\![E.Body]\!]$ and $(i, o) = E.Interface$.

2.2.2 Repetitive tasks: data parallelism

A repetitive task R (rule ($r6$)) expresses data-parallelism. In Figure 4, $T \in \mathcal{T}$ denotes the basic functionality to be replicated on different subsets of data obtained from the input arrays of task R. The resulting *instances* of T are assumed to be independent and schedulable following any order, even in parallel. In rule ($r6$), T is denoted by $Task$ in the task body. The attribute $\mathbf{s_r}$ denotes the *repetition space*, which enables to determine the number of task instantiations, i.e., $|\mathbf{s_r}|$. It is defined itself as a multidimensional array. 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.

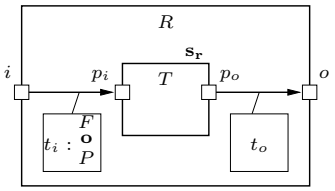

Figure 4: A repetitive task: data parallelism.

Each task instance consumes and produces sub-arrays, called *patterns* or *tiles*, which have the same shape. They are constructed by *tilers* (rule ($r9$)), which are associated with each pair of ports, called *Connexion* in rule ($r8$). A tiler extracts (resp. stores) patterns from (resp. in) an array based on the following information: F: *fitting* matrix (describing how array elements fill patterns); $\mathbf{o}$: *origin* of the *reference pattern*; and P: a *paving* matrix (specifying how patterns cover an array). We briefly recall below the basic principles for pattern fitting and array paving. For more details, the reader may refer to [3].

Given a tile, let its *reference element* denote the origin point from which all its other elements can be extracted. The *fitting* matrix is used to determine these elements. Their coordinates, represented by $\mathbf{e_i}$, are built as the sum of the coordinates of the reference element and a linear combination of the fitting vectors, the whole modulo the size of the array (since arrays are toroidal) as follows:

$$\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e_i} = \text{ref} + F \times \mathbf{i} \bmod \mathbf{s}_{\text{array}} \quad (1)$$

where $\mathbf{s}_{\text{pattern}}$ is the shape of the pattern, $\mathbf{s}_{\text{array}}$ is the shape of the array and F is the fitting matrix. Figure 5 illustrates the fitting result for a $(2,3)$-pattern with the tiling information indicated on the same figure. The fitting index-vector $\mathbf{i}$, indicated in each point-wise element of the pattern, varies between $\binom{0}{0}$ and $\binom{1}{2}$. The reference element is characterized by index-vector $\binom{0}{0}$.

Now, for each repetition instance, one needs to specify the reference elements of the input and output tiles. The reference elements of the reference repetition are given by the *origin* vector, $\mathbf{o}$, of each tiler. The reference elements of the other repetitions are built relatively to this one. As above, their coordinates are built as a linear combination of the vectors of the *paving* matrix as follows:

$$\forall \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}}, \text{ref}_{\mathbf{r}} = \mathbf{o} + P \times \mathbf{r} \bmod \mathbf{s}_{\text{array}} \quad (2)$$

where $\mathbf{s}_{\text{repetition}}$ is the shape of the repetition space, P the paving matrix and $\mathbf{s}_{\text{array}}$ the shape of the array. The paving illustrated by Figure 5 shows how a $(2,3)$-patterns tile a $(6,6)$-array. Here, the paving index-vector $\mathbf{r}$, varies between $\binom{0}{0}$ and $\binom{2}{1}$.

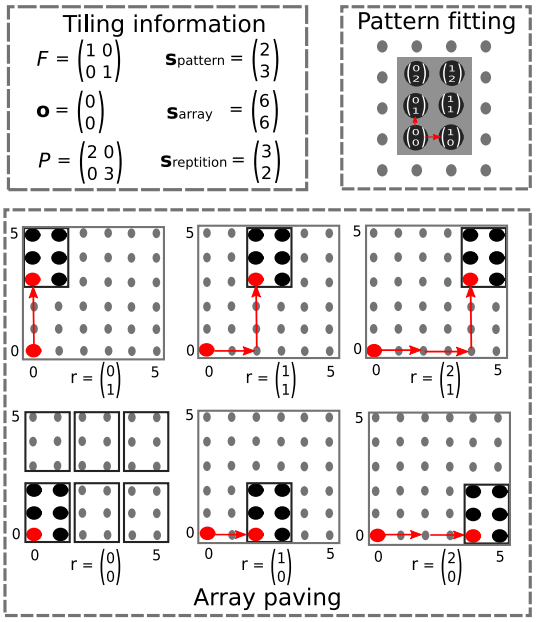

Figure 5: Example of paving and fitting scenarios.

Given a repetitive task R, we denote the tiling operation on an input or output array α of R by using the following convention: $\alpha = \biguplus_t(\alpha^k)$, meaning that α is tiled by the set of patterns $\{\alpha^k | k \in 1..|\mathbf{s}_{\mathbf{r}}|\}$, according to the tiler t. This notion is extended to a set $\{\alpha_1, .., \alpha_j\}$ of j input or output arrays as follows: $\{\alpha_1, .., \alpha_j\} = \biguplus_{t_1..t_j}(\alpha_1^k)..(\alpha_j^k)$, which is equivalent to $\alpha_1 = \biguplus_{t_1}(\alpha_1^k) \wedge .. \wedge \alpha_j = \biguplus_{t_j}(\alpha_j^k)$. On the other hand, for short we will mean by $\{p\}_{1..j}$ the enumeration of j patterns $(p_1..p_j)$.

Definition 4 (Repetitive task)**.** Let R be a repetitive task with s inputs and q outputs. Its behavioral semantics within an environment ε is as follows:

$$\frac{\begin{array}{c}\forall k \in 1..|\mathbf{s_r}|,\ \{p_o^k\}_{1..q} = \phi(\{p_i^k\}_{1..s}),\\ \varepsilon(i) = \biguplus_{t_{i_1}..t_{i_s}} (p_{i_1}^k)..(p_{i_s}^k),\ \varepsilon(o) = \biguplus_{t_{o_1}..t_{o_q}} (p_{o_1}^k)..(p_{o_q}^k)\end{array}}{R \xrightarrow{\varepsilon} R}$$

where $\phi = [\![R.Body.Task]\!]$ and $(i, o) = R.Interface$.

Task instances may sometimes depend on other task instances. For example, this happens when computing the sum of the elements of an array by considering the partial sum previously calculated at each step. Such a constraint therefore induces a total execution order on a repetitive task.

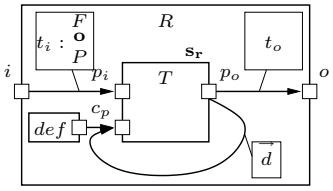

Figure 6: Inter-repetition dependency.

Figure 6 illustrates a simplified notation for a repetitive task with an *inter-repetition dependency*, characterized by rule $(r7)$. *Connexion* represents the pair of ports connected by the dependency link: one is an input to the repeated task T e.g., c_p, and the other is one of its outputs e.g., p_o. The vector $\vec{d}$ specifies the coordinates of the inter-repetition dependency link on the repetition space. For each repetition, c_p denotes a new pattern value to be used as input in the next repetition. Initially, c_p holds a default value, given by def. Note that there could be at the same time several inter-repetition dependencies within a repetitive task since an instance may require values from more than one instance to compute its outputs. This is why rule $(r6)$ specifies a set of dependency link vectors $\{Ird\}$.

Definition 5 (Inter-repetition dependency)**.** Let R be a repetitive task with s inputs and q outputs, where $u \leq q$ outputs are associated with an inter-repetition dependency link. Its behavioral semantics within an environment ε is as follows:

$$\frac{\begin{array}{c}\forall k \in 1..|\mathbf{s_r}|,\ R.Body_k \xrightarrow{(\{p_i^k\}_{1..s+u}, \{p_o^k\}_{1..q})} R.Body'_k,\\ R.Body_{k+1} = R.Body'_k,\\ \varepsilon(i) = \biguplus_{t_{i_1}..t_{i_s}} (p_{i_1}^k)..(p_{i_s}^k),\ \varepsilon(o) = \biguplus_{t_{o_1}..t_{o_q}} (p_{o_1}^k)..(p_{o_q}^k)\end{array}}{R \xrightarrow{\varepsilon} R}$$

where

$$\frac{\begin{array}{c}\{p_o^k\}_{1..q} = \phi(\{p_i^k\}_{1..s}, Body_{ind(\mathbf{r}_k + Body_k.Ird_1.\vec{d})}.Ird_1.c_p, ...,\\ Body_{ind(\mathbf{r}_k + Body_k.Ird_u.\vec{d})}.Ird_u.c_p),\\ \forall l \in 1..u,\ Body'_k.Ird_l.c_p = p_{o_l}^k\end{array}}{Body_k \xrightarrow{(\{p_i^k\}_{1..s+u}, \{p_o^k\}_{1..q})} Body'_k}$$

and $\phi = [\![R.Body.Task]\!]$, $(i, o) = R.Interface$, and the expression $ind(\mathbf{r}_k) \in 1..|\mathbf{s_r}|$ returns an index value associated with vector $\mathbf{r}_k$.

In the first part, just like in Definition 4, the repetitive task R performs the whole repetition and becomes R itself, while ε has the corresponding values for the i and o arrays. The difference is that, for each repetition designated by k in the repetition space $\mathbf{s_r}$, the body $R.Body_k$ performs a transition into $R.Body'_k$, and the order imposed by the dependency makes that the next $R.Body_{k+1}$ is the current $R.Body'_k$. Note that initially in this recurrence, the input patterns of $R.Body_k$ include the default values specified in def for each inter-repetition dependency.

In the second part of the definition, the latter transition is defined: the repeated function ϕ is computed, and the port value c_p of the body's Ird is updated in order to produce $R.Body'_k$. The computation of each $R.Body'_k$ instance takes into account all patterns produced by all other instances which it depends on. This is achieved by calculating the combination of the current position-vector $\mathbf{r}_k$ in $\mathbf{s_r}$ and the dependency vector $\vec{d}$, allowing one to retrieve the c_p values.

2.2.3 Hierarchical tasks: task parallelism

A hierarchical task is defined by an acyclic dependency graph of tasks, as illustrated by an example in Figure 7.

Given any two tasks $T_1, T_2 \in \mathcal{T}$, their functional composition, denoted by $H = T_1 \triangleright T_2$, consists of the definition of a *unidirectional data dependency* relation from the output ports of T_1 to the input ports of T_2. The body of the resulting hierarchical task H (see rule $(r10)$) consists of *i)* the set of tasks $\{T_1, T_2\}$ and *ii)* a set of connexions C s.t. $\forall c = (p_i, p_o) \in C, \forall \varepsilon \in \varepsilon_{\mathcal{P}},\ p_i \in T_1.o, p_o \in T_2.i,\ \varepsilon(p_1) = \varepsilon(p_2)$.

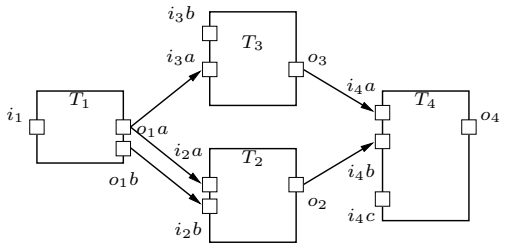

Figure 7: Task parallelism.

Hierarchical tasks $T \in \mathcal{H}$ are obtained by combining tasks of $\mathcal{T}$ pairwise using the above composition operation.

Definition 6 (Hierarchical task)**.** A hierarchical task $R = \{T_1 \triangleright T'_1; ...; T_n \triangleright T'_n\}$ consists of an acyclic graph of tasks, obtained by composition. Its behavioral semantics within a set of environments $\{\varepsilon_1, ..., \varepsilon_n\}$ is as follows:

$$\frac{(T_1 \triangleright T'_1) \xrightarrow{\varepsilon_1} (T_1 \triangleright T'_1), ..., (T_n \triangleright T'_n) \xrightarrow{\varepsilon_n} (T_n \triangleright T'_n),\ \ \varepsilon_1 \ ... \ \varepsilon_n \text{ are composable}}{\{T_1 \triangleright T'_1; ...; T_n \triangleright T'_n\} \xrightarrow{\varepsilon_1 \oplus ... \oplus \varepsilon_n} \{T_1 \triangleright T'_1; ...; T_n \triangleright T'_n\}}$$

where

$$\frac{T_1 \xrightarrow{\varepsilon_1} T_1,\ T_2 \xrightarrow{\varepsilon_2} T_2,\ \varepsilon_1 \text{ and } \varepsilon_2 \text{ are composable}}{(T_1 \triangleright T_2) \xrightarrow{\varepsilon_1 \oplus \varepsilon_2} (T_1 \triangleright T_2)}$$

Figure 7 illustrates situations where outputs are featured in multiple dependencies, and where input or outputs are not featured in any dependency (i.e., they are part of the interface of the hierarchical task).

The task resulting from the successive composition of several tasks forms a hierarchical acyclic graph, where nodes are tasks, and edges are labeled by arrays exchanged between tasks through their interface ports.

2.3 Examples

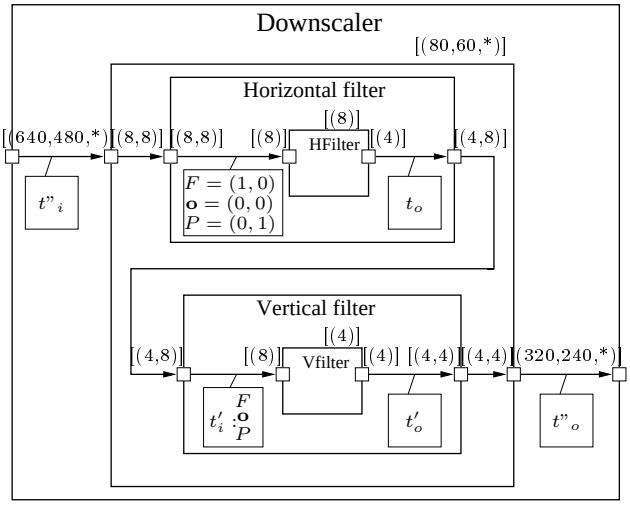

Figure 8: Example of a downscaler.

Figure 8 shows the example of a repetition of a hierarchical task, where the sub-tasks are themselves repetitions. Here, for the sake of simplicity, only the shape information is shown on task ports.

This model represents a downscaler, which takes an infinite array (intuitively, a flow, which is noted *) of images of size 640×480; the output is a flow of 320×240 images. Inside, a repetition space of shape [(80,60,*)] is applied to the hierarchical task. The input tiler t''_i accordingly extracts patterns of shape [(8,8)], and the output tiler t''_o reconstructs, from patterns of shape [(4,4)], the output image. The hierarchical task is instantiated a number of times given by the repetition space, and each repetition involves the execution of one instance of horizontal filter and of one instance of vertical filter, the latter taking as input some of the outputs of horizontal filter. Each of them is itself a repetition, with its own tilers and repeated function (respectively, *Hfilter* and *Vfilter*) applied to patterns.

Beyond the modeling of system behaviors and computations, the repetitive MoC of GASPARD2 also enables to describe purely structural features of a hardware architecture as well as the mapping of application functionalities on hardware architectures. Figure 9 shows how a hardware architecture model representing a 16×16-grid of processing units. Each unit is composed of a crossbar, a memory and a MIPS processor. Here, the inter-repetition dependency link is used to express the way processing units are inter-connected in

the grid. Using the same concepts, one can also represent both task allocation and data allocation of a software application model (e.g. the downscaler) on a hardware architecture model (e.g. the 16×16-grid of processing units) [4]. For that, tilers are used at each extremity of an allocation link to specify, e.g. which application task and data instances are associated with which platform memory and processors instances. They enable to describe different types of regular distributions: per block, cyclic or k-cyclic.

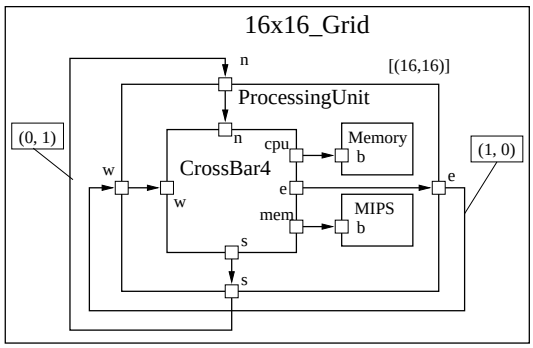

Figure 9: A 16×16-grid of processing units.

A very interesting feature of the above models is their elegant way to allow a compact representation of the parallelism degree inherent to a data-parallel application as well as a massively parallel architecture, and the block allocation of the first on the second. The obtained models do not suffer from any scalability problem regarding the parallelism degree, contrarily to other modeling formalisms. This is a major advantage of using the repetitive model for high-performance system specification.

3 A reactive control extension

The reactive control modeling presented here relies on *finite state machines* described following the same style as the GASPARD2 task models introduced in the previous section. The connection between the control and data parts is established by implementing differents *modes* for data tasks. Through this modeling, the regularity inherent to the repetitive MoC remains preserved while computations become controllable.

3.1 Mode tasks

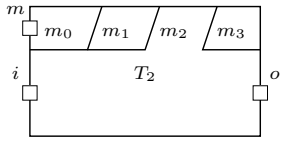

Figure 10: A mode task.

A *mode task* expresses a choice among several possible alternative computations [10]. Figure 10 illustrates such a task, inspired by windows with multiple tabs. It is composed of several modes, identified by some values of an enumerated data type: m_0, m_1, m_2, ..., m_k. The computation defined by the task T_k transforms the input data i into the output data o according to the mode m_k determined by the input mode value m. We extend the language as follows:

$$\begin{array}{lcll} Body^{mt} & ::= & \{(m_k, T_k) : (mode\_id, Task)\}, \forall i \neq j & \\ & & \Rightarrow T_i.Interface = T_j.Interface & (r11) \\ Body & ::= & Body^{mt} \mid Body^{h} \mid Body^{r} \mid Body^{e} & (r4') \end{array}$$

The interface of $Body^{mt}$ comprises at least an input m denoting the received mode value on which relies the choice of the mode to execute. All tasks T_k associated with modes m_k have the same interface, such that: $\forall T_k,\ Interface(T_k) = Interface(Body^{mt}) \backslash \{m\}$.

The modes run exclusively, meaning that whenever the mode task executes, only the task T_k associated with the selected mode m_k is computed. This is also the case in mode automata [5, 13]. It is particularly useful when analyzing the behavior of the mode task since it eliminates by construction the risk for possible interaction between faulty and non-faulty modes, hence favoring safe designs.

Given a collection $(m_1, T_1), ..., (m_k, T_k)$ of modes and their associated tasks in a mode task, we consider an arbitrary order of evaluation to choose the mode to execute: *from left to right* in Figure 10. The chosen mode is always the first encountered, which satisfies the input mode value m. When two modes or more are identical, only the first one according to the previous order is executed. This is akin to "`case`" and "`match`" statements in respectively the synchronous languages SIGNAL and LUCID SYNCHRONE [1].

Definition 7 (Mode task)**.** Let M be a mode task. Its behavioral semantics within an environment ε is as follows:

$$\frac{\varepsilon(m) = m_k, \varepsilon(o) = \phi(\varepsilon(i))}{M \xrightarrow{\varepsilon} M}$$

where $\phi = [\![M.Body.T_k]\!]$ and $i, o \in M.Interface$.

The way a mode task M interacts with any other task is dictated by the nature of its embedded tasks T_k. E.g., if T_k features an elementary task then whenever its associated mode m_k is selected, M reacts as an elementary task; if T_k features a repetitive task, MT will reacts as a repetitive task and so on. Note that since all T_k's have identical interfaces, the difference of their nature is not externally visible.

3.2 Transition functions

A great advantage of introducing *transition functions* in GASPARD2 is that they can be used to define mode values that serve to achieve different computations. So, they are ideal companions of mode tasks. Figure 11 illustrates a transition function in a particular context, corresponding to the proposal of an automaton component in GASPARD2 [10].

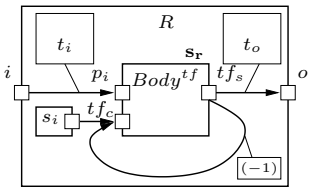

Figure 11: A transition function in context.

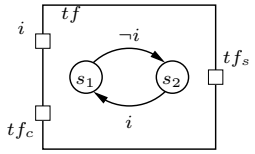

Figure 12: Simple transition function.

Context and interface of a transition function A transition function is a task defined by an interface and a body $Body^{tf}$, that computes, given some inputs p_i from its environment (used in the transition conditions), and a current state tf_c, the new value of its state tf_s (resulting from the transition). Hence, it has to be used within the context of a repetition, with an inter-repetition dependency, so that the result tf_s from the previous repetition (-1) is used as input tf_c. The initial state s_i is given as default value. This context is just a very classical encoding of an automaton as a sequential circuit.

In order to dissociate the mode value from the particular coding of states inside the body of the transition function, one can insert an elementary task μ that transforms each state value into a mode value. Here, for the sake of simplicity, we consider the direct coding of modes as state values. In this context, the automaton performs transitions on an array of inputs, and produce an array of state/mode values as output (see Figure 11).

We will propose further other interesting contexts within which transition functions can be used. The idea is always that a transition function is defined in the scope of a repetition, and this defines the pace of its reactions.

Body of a transition function In order to define what transition to take according to the current state and an incoming input, we construct the body $Body^{tf}$ in terms of a state graph notation[3], as exemplified in Figure 12, which is easier than a complex conditional statement:

$$
\begin{array}{lcll}
Body^{tf} & ::= & \{(tf\_id; S; Tr; s_i)\} & (r12) \\
S & ::= & \{state\_id \mid (state\_id; Body^{tf}; reset)\} & (r13) \\
Tr & ::= & \{(state\_id; label; state\_id)\} & (r14) \\
Body & ::= & Body^{tf} \mid Body^{mt} \mid Body^{h} \mid Body^{r} \mid & \\
 & & Body^{e} & (r4'')
\end{array}
$$

A body is a set of state graphs, each of which is a four-tuple: a name tf_id, a set S of states, a set Tr of transitions and an initial state s_i. States in the

[3]Even though our notation of transition functions is very close to that of automata, it does not have the same meaning. It only specifies a set of transitions between states. Fireable transitions are selected upon the values of the inputs (see Figure 12), denoting the labelling condition and the current state from which the transitions should take place.

set S are either a simple state with a name $state\_id$, or recursively, a state graph characterized by a triple: its name $state\_id$, the sub-body $Body^{tf}$, and a Boolean *reset* stating whether reinitialization should occur when re-entering the state. Transitions are triples with the name of the source state, a *label* which is a Boolean expression on inputs, and the name of the destination state.

An important required property of these state graphs is that they are deterministic, meaning that for each state $s \in \mathcal{S}$, if there are several possible transitions from s, the values of input events satisfy only one transition.

The control part of a system can be described hierarchically. In this case, the state can have a sub-body. We define its behavior in a similar way as StateCharts, Esterel and SyncCharts, or Mode Automata. The interface must feature corresponding inputs and outputs for each of the transition functions, as shown in Figure 13, where state h_2 of the high-level transition function h has a sub-body l, for which the star $*$ indicated that rst is true.

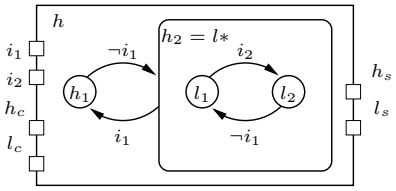

Figure 13: Hierarchical transition function.

Definition 8 (Hierarchical transition function)**.** Let a be a hierarchical transition function, its semantics within an environment ε is as follows:

$$\frac{\begin{array}{c}\varepsilon(a\_id_c) = s_{c_a}, \exists (s_{c_a}; e; s_{d_a}) \in Tr_a, \varepsilon(e) = tt, \\ (s_{d_a}; b; rst) \in S_a \Rightarrow \\ \Big(\varepsilon(a\_id_s) = s_{d_a}, rst \Rightarrow \varepsilon(b.tf\_id_s) = b.s_i \wedge \\ \neg rst \Rightarrow \varepsilon(b.tf\_id_s) = \varepsilon(b.tf\_id_c)\Big) \\ (s_{d_a}; b; rst) \notin S_a \Rightarrow \varepsilon(a\_id_s) = s_{d_a}\end{array}}{(a\_id; S_a; Tr_a; s_{i_a}) \xrightarrow{\varepsilon} (a\_id; S_a; Tr_a; s_{i_a})}$$

and

$$\frac{\begin{array}{c}\varepsilon(a\_id_c) = s_{c_a}, \forall (s_{c_a}; e; s_{d_a}) \in Tr_a, \varepsilon(e) = ff, \\ (s_{c_a}; b; rst) \in S_a \Rightarrow \Big(\varepsilon(a\_id_s) = s_{c_a}, b \xrightarrow{\varepsilon} b\Big) \\ (s_{c_a}; b; rst) \notin S_a \Rightarrow \varepsilon(a\_id_s) = s_{c_a}\end{array}}{(a\_id; S_a; Tr_a; s_{i_a}) \xrightarrow{\varepsilon} (a\_id; S_a; Tr_a; s_{i_a})}$$

where

$$\forall (s; b; rst) \in S_a, s \neq \varepsilon(a\_id_s), \varepsilon(b.tf\_id_s) = \varepsilon(b.tf\_id_c).$$

In the above definition, given a transition function a, from its current state a_id_c:

- either there exists a transition for which the labelling expression e evaluates to true (tt), and
  - if the destination state s_{d_a} has a sub-body $b.tf\_id$, i.e. s_{d_a} is of the form $(s_{d_a}; b; rst)$:

  - ∗ either *reset* is true, then the output value for $b.tf\_id_s$ is the initial state $b.s_i$;
  - ∗ or not, then the output value for $b.tf\_id_s$ is the input current state $b.tf\_id_c$.
- otherwise, s_{d_a} is simply returned;

- or, for all outgoing transitions the label expression evaluates to false (*ff*), and the current state s_{c_a} remains the same. Then,
  - if this state s_{c_a} of the transition has a sub-body $b.tf\_id$, the transitions of $b.tf\_id$ are performed in the same environment;
  - otherwise, s_{c_a} is simply returned.

Finally, the value of states that are not concerned by current transitions in the hierarchical transition function remain unchanged.

Transition functions can be also combined so as to obtain a parallel execution as illustrated in Figure 14. The associated behavior is that all functions make their transitions in parallel, within the same global transition.

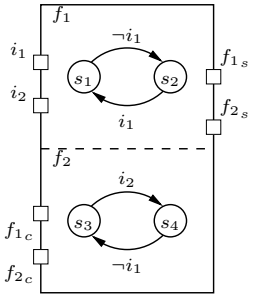

Figure 14: Parallel transition function.

Definition 9 (Parallel transition functions)**.** Let A be a parallel transition function, its semantics within an environment ε is as follows:

$$\frac{\forall a \in A.Body, a \xrightarrow{\varepsilon} a}{A \xrightarrow{\varepsilon} A}$$

Parallel transition functions may synchronize through data dependencies, the output from the one being an input of the other. These data dependencies have to be acyclic. Of course, parallel and hierarchical constructions of transition functions can be combined freely.

4 Typical design examples

The above constructs can be now considered to define behaviors or structures that combine data-intensive computations with control. We propose ways of using the transition functions and mode tasks, which are remarkable in that they exhibit a behavior comparable to, e.g., mode automata or structured hierarchical automata.

Mode automaton An example of useful macro construct is one that offers a similar semantics to synchronous mode automata [5, 13]. Statements representing the data intensive part are executed depending on the state, which is updated by transitions, both at the same rate or clock. In Figure 15, which illustrates the phone example mentioned in Section 1, this macro construct consists of a repetitive task RT with an inter-repetition dependency. Here, the repeated task is a hierarchical task HT in which, a mode task executes a data-intensive algorithm to define the resolution of some images received from a source, depending on the power status in a phone. The status information is characterized by the output values of the transition function TF, defined by its state graph. Typically, each state of TF encodes a power level. We refer to the repeated task as a *mode transition function*, and the enclosing repetitive task as a *controlled task*.

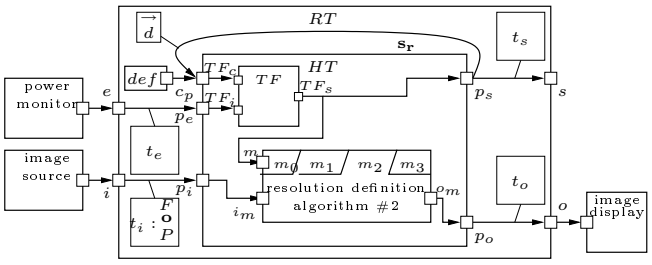

Figure 15: Example of mode automaton.

Hierarchical task with parallel transition functions Parallel automata can be simply encoded as shown in Figure 16 by a repetitive task in which the mode transition functions associated with each automaton are defined at the same hierarchical level, while sharing the same repetition space. Hence, they make their transitions simultaneously at each step. This particular pattern of construction is remarkable in the sense that it is similar to the synchronous composition of transition functions, with unilateral data dependencies. Here,

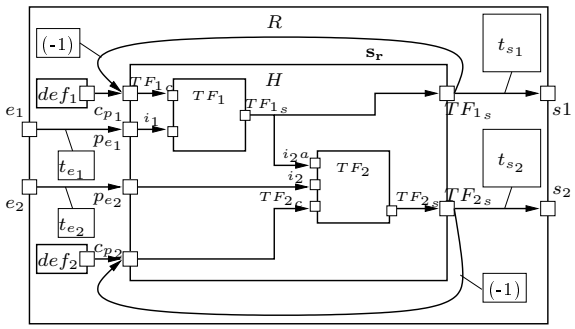

Figure 16: Parallel transition functions.

transition function TF_1 performs one step at each repetition, taking as input a pattern from the array e_1. It produces TF_{1s} as output, which is integrated in the resulting array s_1, and which is also used as input by the other transition

function TF_2. The function TF_2 takes another input from the array e_2, and produces TF_{2_s} in the *same repetition*, as TF_1 because both are in the same repeated hierarchical task. Finally, TF_{2_s} is integrated in s_2.

Mode task with hierarchical transition functions This particular pattern of construction is remarkable in the sense that it is an encoding of a hierarchical automaton in terms of repetitive tasks, as shown in Figure 17. It behaves similarly to hierarchical automata in reactive languages [5, 13].

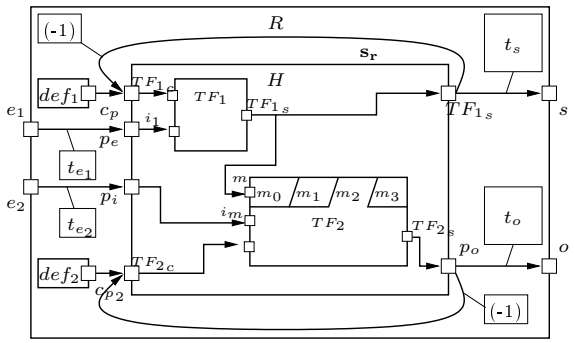

Figure 17: A hierarchy of transition functions.

At each step (or repetition), the upper-level automaton, characterized by the mode transition function TF_1, makes a transition. This results in selecting a mode to run within the mode task. The selected modes are themselves automata characterized by the mode transition function TF_2. They represent the lower level automata in the global one. One can notice that both TF_1 and TF_2 share the same repetition space. It means that their transitions are performed synchronously.

Oversampled transition functions Another original combination is illustrated in Figure 18, where the automaton corresponding to transition function TF_2 performs a whole run at the pace of repetition R_l, during each single step of transition function TF_1, which is at the step of repetition R_h. Nested repetitions give way to nested clocks, in a form of oversampling.

In the above resulting models, context switches from one mode to another, at a given hierarchical level, are only performed *between* repetitions defined at this level. Contrarily to usual exceptions, which must be served immediately, here one has to wait for the completion of the current repetition so as to preserve the regularity of the repetitive execution schema. However, thanks to the hierarchy of our models, fine grain controllability is possible by defining the switch functions at the suitable repetition granularity levels. For instance, consider a repetitive task R that transforms a set of images such that each repetition instance R_k of R transforms one image from the set. The instances R_k are themselves repetitive tasks for which each instance R_{kl} transforms a pixel-line from an image. Both repetition levels associated with R_k and R_{kl} can be associated with transition functions to control at the same time what algorithms apply to a whole image and within an image, what specific algorithms apply to a pixel-line.

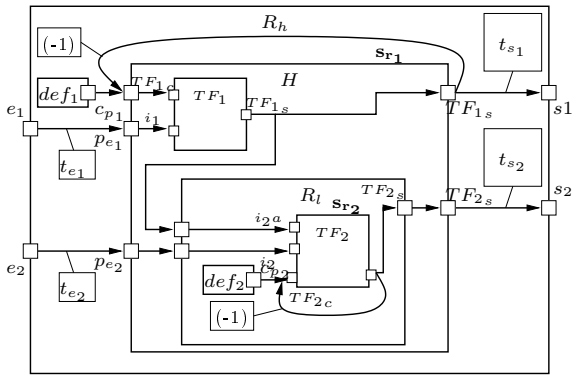

Figure 18: Oversampling transition functions.

5 Discussion and related work

The combination of control and data-parallel features to define high-performance algorithms has been investigated for several years in the context of various languages, e.g. Mentat [19], PSather [11], MasPar programming language [8]. These studies particularly consider *control parallelism* that amounts to a concurrent execution of different instruction streams. They showed that the execution performances of parallel systems can be significantly improved. The concepts used to describe control in these studies are mostly the usual system-level scheduling and synchronization mechanisms such as fork/join instructions, master/slave model or monitors. In our model, control is described in terms of computation *modes*: there are several possible ways to compute the same data, which are considered exclusively at any time. The way data-parallel computations switch from a mode to another is described by transition functions. For this purpose, a few constructs have been identified that can be expressed using the data-parallel concepts of GASPARD2. This limited degree of control is expressive enough to permit the modeling of both application-level and architecture-level adaptability scenarios in high-performance systems. The result could be considered for either simulation or circuitry synthesis or formal verification.

Similar control-oriented concepts have been introduced in other dataflow models to express dynamic changes or reconfiguration in streaming applications [17] [14]. The solutions proposed in these studies consist of *Synchronous Data Flow* (SDF) model variants that integrate new features to specify modes. For instance, in [17], authors use a specific notion called *scenario* to express how the execution is carried out in their SDF variant. A major difference between our GASPARD2 model and SDF variants comes from the interesting expressivity offered by the repetitive model of computation, which is more suitable for the uniform and compact expression of the parallelism in complex high-performance embedded systems such as MPSoC.

In comparison with [10], we have extended expressivity inside, by allowing for parallel and hierarchical transition functions in the body, and outside, in the sense that our transition function can be used in contexts other than the sole automaton component. We have also considered the use of control and modes

not only at the application level, with switches between different functionalities, as was the case before, but also to represent different execution modes, with switches between different implementations of the same functionality, which is a way to consider the control of architectural aspects in the model.

Beyond the solution adopted here to describe control aspects in GASPARD2, an alternative solution that may be also considered consists in using *clocks*. For instance in [15], the authors define a specific clock notion, called *affine clock*, which is well adapted to specify synchronization relations in the data-parallel language ALPHA [20]. Ideally, such a clock notion should be multidimensional so as to be adequate to GASPARD2 data types. The multidimensional time model proposed in [7] can therefore appear as a possible solution idea to this issue.

6 Conclusion

We have presented a model that serves, in a design environment called GASPARD2, for the development of high-performance embedded systems with adaptability scenarios w.r.t. various aspects: QoS, platform-dependent constraints, etc. This model combines the repetitive model of computation (MoC) with ideas based on finite state machines and modes. We formally defined the semantics of the resulting mixed model that could be used further for reasoning on designs. We showed, through simple examples, that this model increases the expressivity in GASPARD2 while still preserving the benefits of regularity of the repetitive MoC.

The existence of transformation chains in GASPARD2, towards different target technologies (SystemC, VHDL, OpenMP Fortran and synchronous languages) offers the opportunity to exploit the new model from various viewpoints in the future. In particular, simulation and formal verification will be made possible via supported synchronous languages (currently LUSTRE and SIGNAL). Control can be comfortably extracted under the form of reactive mode automata that may be considered for behavioral simulation and formal verification by model-checking. The generation of synchronous mode automata from GASPARD2 models consists of the enhancement of the existing transformation chain [21].

References

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

[2] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

[3] P. Boulet. Formal Semantics of ARRAY-OL, a Domain Specific Language for Intensive Multidimensional Signal Processing. Technical Report 6467, INRIA, France, March 2008. `http://hal.inria.fr/inria-00261178/en`.

[4] P. Boulet, P. Marquet, E. Piel, and J. Taillard. Repetitive allocation modelling with marte. In *Forum on specification and Design Languages, FDL'07,* Barcelona, Spain, september 2007.

[5] J.-L. Colaço, G. Hamon, and M. Pouzet. Mixing signals and modes in synchronous data-flow systems. In *6th ACM & IEEE International conference on Embedded software (EMSOFT'06)*, pages 73–82, New York, NY, USA, 2006. ACM Press.

[6] A. Demeure and Y. Del Gallo. An array approach for signal processing design. In *Sophia-Antipolis Conf. on Micro-Electronics (SAME'98), France*, Oct. 1998.

[7] P. Feautrier. Some efficient solutions to the affine scheduling problem, part ii. multidimensional time. *Int. journal of parallel programming*, 21(6):389–420, December 1992.

[8] V. Garg and D. E. Schimmel. Exploitation of control parallelism in data parallel algorithms. In *5th Symp. on the Frontiers of Massively Parallel Computation (Frontiers'95)*, Washington, DC, USA, 1995.

[9] High Performance Fortran Forum. High performance fortran language specification, January 1997. `http://hpff.rice.edu/versions/hpf2`.

[10] Ouassila Labbani, Jean-Luc Dekeyser, Pierre Boulet, and Éric Rutten. Introducing control in the gaspard2 data-parallel metamodel: Synchronous approach. In *Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES'05), Montego Bay, Jamaica*, october 2005.

[11] Chu-Cheow Lim, Jerome A. Feldman, and Stephan Murer. Unifying control- and data-parallelism in an object-oriented language. In *Joint Symposium on Parallel Processing*, pages 261–268, Waseda University, Tokyo, May 1993.

[12] E. Lusk and K. Yelick. Languages for High-Productivity Computing: The DARPA HPCS Language Project. *Parallel Processing Letters*, 17(1):89 – 102, march 2007.

[13] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(1-2):219–254, 2003.

[14] S. Neuendorffer and E. Lee. Hierarchical reconfiguration of dataflow models. In *2nd Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE'04)*, pages 179–188, june 2004.

[15] I.M. Smarandache, T. Gautier, and P. Le Guernic. Validation of mixed SIGNAL-ALPHA real-time systems through affine calculus on clock synchronisation constraints. In *World Congress on Formal Methods (2)*, pages 1364–1383, 1999.

[16] The GASPARD2 platform. `https://gforge.inria.fr/projects/gaspard2`.

[17] B.D. Theelen, M.C.W. Geilen, T. Basten, J.P.M. Voeten, S.V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *4th Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE'06)*, pages 185–194, july 2006.

[18] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conf. on Compiler Construction*, Grenoble, France, 2002.

[19] Emily A. West. Combining control and data parallelism: Data parallel extensions to the mentat programming language. Technical Report CS-94-16, University of Virginia, Charlottesville, VA, USA, 18, 1994.

[20] D. Wilde. The ALPHA language. Technical Report 827, IRISA - INRIA, Rennes, 1994. `www.irisa.fr/centredoc/publis/PI/1994`.

[21] H. Yu, A. Gamatié, E. Rutten, and J.-L. Dekeyser. *Embedded Systems Specification and Design Languages, Lecture Notes Electrical Engineering, Vol. 10, Villar Eugenio (Ed.)*, chapter 13: Model Transformations from a Data Parallel Formalism towards Synchronous Languages. Spr. Verlag, 2008.

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

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