HAL
archives-ouvertes.fr

Introducing Control in the Gaspard2 Data-Parallel Metamodel: Synchronous Approach

Ouassila Labbani, Jean-Luc Dekeyser, Pierre Boulet, Eric Rutten

▶ To cite this version:

Ouassila Labbani, Jean-Luc Dekeyser, Pierre Boulet, Eric Rutten. Introducing Control in the Gaspard2 Data-Parallel Metamodel: Synchronous Approach. [Research Report] RR-5794, INRIA. 2006, pp.49. inria-00070228

HAL Id: inria-00070228
https://hal.inria.fr/inria-00070228

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Introducing Control in the Gaspard2 Data-Parallel Metamodel: Synchronous Approach

Ouassila Labbani — Jean-Luc Dekeyser — Pierre Boulet — Éric Rutten
Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille
Bâtiment M3, Cité Scientifique

59655 Villeneuve d'Ascq Cedex, France

N° 5794

Janvier 2006

Thème COM

Rapport de recherche

ISSN 0249-6399

Introducing Control in the Gaspard2 Data-Parallel Metamodel: Synchronous Approach

Ouassila Labbani , Jean-Luc Dekeyser , Pierre Boulet , Éric Rutten
Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille
Bâtiment M3, Cité Scientifique
59655 Villeneuve d'Ascq Cedex, France

Thème COM — Systèmes communicants
Projet DaRT

Rapport de recherche n° 5794 — Janvier 2006 — 46 pages

Abstract:
In this document, we study the introduction of control in the Gaspard2 application UML metamodel by using the synchronous reactive system principles. This allows to take the change of running mode into account in the case of data parallel applications, and to study more general ways of mixing control and data parallel processing.

Our study is applied to a particular context using two different models, exclusively dedicated to the process of computation or control. The computation part represents the Gaspard2 application metamodels based on the Array-OL language. This Language is often used to specify the data dependencies and the potential parallelism in intensive signal processing applications manipulating multidimensional data. The control part is represented by an automaton structure based on the Mode-Automata concept which makes it possible to clearly identify the different modes of a task and the switching conditions between modes.

For this kind of applications, mixing control and data parallel processing, we propose an UML metamodel allowing to better visualize and control the construction of the system by clarifying, at a height abstraction level, the various relations and the possible interactions of this system.

The proposed UML metamodel makes it possible to describe and to model the control automata, the different running modes and the link between control and computation parts. It also allows to clearly separate control and data parts by respecting the concurrency, the parallelism, the determinism and the compositionality of the Gaspard2 models.

Key-words: Data parallelism, Array-OL, Gaspard2, UML, Reactive systems, Synchronous approach, Running modes, Control/data combination

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 72 92 59 ??

Introduction du Contrôle dans le Métamodèle d'Application de Gaspard2: Approche Synchrone

Résumé :

Ce document étudie l'introduction du contrôle dans le métamodèle UML d'application de Gaspard2 en se basant sur les principes de systèmes réactifs synchrones. Cette notion permet de prendre en considération la possibilité du changement de modes de fonctionnement dans le domaine du parallélisme de données, et d'étudier des mécanismes plus généraux mixant du contrôle et des traitements de données parallèles.

Notre étude est appliquée dans un contexte particulier utilisant deux modèles différents, dédiés exclusivement aux traitements de données ou du contrôle. La partie calcul représente les métamodèles d'application de Gaspard2 basés sur le langage Array-OL. Ce langage est principalement utilisé pour la spécification des dépendances de données et du parallélisme potentiel dans des applications de traitement de signal intensif traitant des données multidimensionnelles. La partie contrôle est représentée par une structure d'automate basée sur le concept d'automates de modes permettant d'identifier clairement les différents modes de fonctionnement pour une tâche et les conditions de changement entre modes.

Pour ce type d'applications, mixant du contrôle et des traitements de données parallèles, nous proposons un métamodèle UML permettant de mieux visualiser et contrôler la construction du système en clarifiant, à haut niveau d'abstraction, ses différentes relations et interactions possibles.

Le métamodèle UML proposé permet de décrire et de modéliser les automates de contrôle, les différents modes de fonctionnement et le lien entre la partie contrôle et la partie calcul. Il permet aussi de séparer de façon claire entre le contrôle et les traitements de données en respectant la concurrence, le parallélisme, le déterminisme et la compositionnalité des modèles de Gaspard2.

Mots-clés : Parallélisme de données, Array-OL, Gaspard2, UML, Systèmes réactifs, Approche synchrone, Modes de fonctionnement, Combinaison contrôle/données

Contents

1 Introduction and Motivation **4**

2 Context **5**
2.1 Intensive Signal Processing and Data Parallelism 6
2.1.1 Array-OL 6
2.1.2 System on Chip Co-design and the Gaspard2 Environment 11
2.1.3 UML Profile for Modeling Gaspard2 Components 12
2.2 Synchronous Real-Time Reactive Systems 16
2.2.1 Real-Time Reactive Systems 16
2.2.2 Synchronous Approach 19
2.2.3 Synchronous Behavior and Automaton Structure UML Modeling . . . 20
2.3 Mixing Control and Data Processing 22
2.3.1 Mode-Automata 23
2.3.2 Control/Data Flow Separation Methodology 24

3 Introducing Control in the Gaspard2 Data-Parallel Metamodel **26**
3.1 Modeling of the Control Part 26
3.2 Modeling of the different Running Modes 28
3.3 Modeling the Link between the Control Part and the Different Running Modes . 29

4 Degrees of Granularity and Control Dependency for the Control of Parallel Applications **33**
4.1 External Control: Changing Modes According to the External Environment . . 34
4.2 Internal Control: Changing Modes According to the Computation Results . . . 39

5 Conclusion and Future Work **42**

1 Introduction and Motivation

Computation intensive multidimensional data applications are more and more present in several application domains such as image and video processing or detection systems (radar, sonar, ...). The main characteristics of these applications is that they operate in real time conditions and are generally complex and critical. They are also multidimensional since they manipulate multidimensional data structured into arrays.

To study intensive signal processing applications, some computation models have been proposed to model and implement these systems. Among these models, we can find *MDSDF* (MultiDimensional Synchronous Dataflow) [10], *GMDSDF* (Generalized MultiDimensional Synchronous Dataflow) [26] and *Array-OL* (Array Oriented Language) [11].

In industry, the preoccupations of efficiency, re-use, and reliable programming are well-known. On the one hand, the need expressed by the customer, often unskilled in the control or data processing design, is generally imprecise and incomplete, and the designer is unfamiliar to the customer area. It is thus a question to find a common language between these two different worlds. On the other hand, an additional challenge is posed by the division of work. The development is performed by several designers since the various parts of a system are always more or less interdependent. For that, developers and designers must be able to agree on the interaction functionalities and the used formalisms by using common models.

A model is a simplification and/or an abstraction of reality. Modeling consists in identifying the interesting or relevant characteristics of a system to be able to study its behavior according to these characteristics. A good model must has two main characteristics [27]: first, it must facilitate the comprehension and reduce the complexity of the studied system; and second, it must allow the simulation and the verification of its various concepts.

In our study, we are interested in modeling parallel applications using the Array-OL model and one of its development environments, *Gaspard2*[1]. The Gaspard2 environment uses UML2 principles [24] for modeling and studying applications. This makes it possible to better visualize and control the construction of the system by clarifying their various relations and possible interactions. This model allows to easily program intensive signal processing applications. It is used to specify the *data parallelism* and the *data dependencies* between tasks. The Gaspard2 environment is a model driven System-on-Chip co-design environment. The designs are based on a Y development model in which, from two UML[2] models describing the application and the hardware architecture, the application is mapped to the hardware architecture in an association model. This association model is then projected onto lower level descriptions such as simulation or synthesis models.

Signal processing applications modeled in Array-OL can be considered as purely data flow based and only represent an intensive data processing without any reaction or control concepts. The goal of our work is to introduce, in the Gaspard2 application metamodel, the control concepts and the possibility to change running modes according to the execution context of the studied applications.

The introduction of control into data parallel applications requires the definition of a clear model and a rigorous semantics allowing to take various types of applications into account, mixing control and data parallel processing. This concept gives a *reactive* behavior to the studied parallel applications. In this case, the systems are not only describable by transformational relations, specifying outputs from inputs, but also by relations between outputs and inputs via

[1] `http://www.lifl.fr/west/gaspard`
[2] `http://www.uml.org`

their possible combinations in time. Consequently, the combination of descriptions including complex sequences of events, actions, conditions and information flow allows to synthesize the global behavior of a *reactive system* [1].

The complexity of reactive systems comes from the complex characteristics of the reactions to the different occurrences of discrete events [2]. This complexity can make it difficult to model the behavior of such systems and exposes them to errors. It becomes necessary to introduce rigorous design methods and formalisms to specify the behavior of these critical systems. Among these formalisms, the *synchronous approach* represents a significant contribution to this field [3]. It is based on the *synchrony hypothesis* which considers the execution of a reactive system as an infinite succession of instantaneous reactions.

In this document, we use the concept of synchronous reactive systems to introduce the control parts in the Gaspard2 application UML metamodel. The basic idea is inspired by the principles of *Mode-Automata* [4] used in the case of synchronous reactive systems to clearly express the different running modes of an application and the conditions of switching between modes.

The metamodel that we propose must, on the one hand, clearly separate control and data flow parts [12], and on the other hand, respect concurrency, parallelism, determinism and compositionality of Gaspard2 models. In this model, the computation part represents a set of parallel tasks (signal processing, image processing, . . .) while the control part represents the switching conditions between the different running modes of the tasks according to control values. These values can be provided by the environment (pressing a button, temperature changes, . . .) or by the computation part (result of a preceding computation, dependency between tasks, . . .).

In the following sections, and after a presentation of the used concepts, we study the possibility to take the different changes of mode in a parallel application metamodel into account. This concept requires a good definition of the *degree of granularity* of applications or the *clock signal* for which the control values can be taken into account. Our work is based on the Gaspard2 application metamodel [17] and proposes a UML solution for the modeling of the control automata, the different running modes and the link between the two in the case of a *synchronous approach*.

The document is organized like this: in the second section, we present the main concepts used in our study. In this section, we outline the area of intensive signal processing, by presenting Array-OL language and Gaspard2 environment, and the area of synchronous reactive systems. For these concepts, we study the existing UML devices, and we also present some existing work on the combination between control and data processing, in particular the Mode-Automata concept. The third section is devoted to the introduction of the control in the Gaspard2 data-parallel metamodel. In this section, we propose a UML modeling solution for the control parts, the different running modes and the link between control and parallel processing. The last section shows, through examples, that the introduction of the control into data-parallel applications requires the definition of a *degree of granularity* to delimit the different execution cycles for these applications.

2 Context

In this section, we give a definition of the concepts used in our study. The context of this study can be classified into two main parts. The first part is related to intensive signal processing applications and presents the Array-OL language and its Gaspard2 environment in particular. The second part is about synchronous reactive systems and the Mode-Automata concept. For

both parts, we also study the existing UML devices and concepts which can be used to model these kind of applications. After that, we present a global view on the combination between control and data processing by refering some existing work present in the literature, and in particular the Mode-Automata concept.

2.1 Intensive Signal Processing and Data Parallelism

2.1.1 Array-OL

Array-OL (Array Oriented Language) has been developed by Alain Demeure at TUS (Thales Underwater System) in 1995 [21]. It is a specification language allowing to express parallel applications by the way of data dependencies. This language has mainly been introduced to model intensive signal processing applications manipulating a great number of data in a regular way. It is based on a multidimensional model and makes it possible to express the whole potential parallelism of these applications (data or task parallelism).

The data are structured in arrays which can have only one infinite dimension generally used to express time. Each task of the application consumes one or more arrays by “pieces” of the same size called *patterns*, and produces a new patterns for the output arrays. This process continues, and the produced arrays can be consumed in their turn.

In the Array-OL model, the different tasks are connected to each other using data dependencies. The expression of these dependencies initially allows to define a minimal partial order on the execution of these tasks. The compiler can then complete this partial order in an efficient parallel execution. When a data dependency is expressed between two tasks, it means that one of these two tasks needs whole or part of the data produced by the other task to be able to perform its computations. The compilation of Array-OL models has largely been studied by Soula, Dumont et al. [15, 16].

The Array-OL models can have hierarchical compositions on several description levels. In a hierarchical model, the data dependencies are mainly approximative until the lowest level where these dependencies are completely expressed. The description of an application in Array-OL uses two models. The *global model*, defines the sequence of the different parts of the application, in other words, the task parallelism, and the *local model*, specifies the elementary actions performed on the table elements and the existing data parallelism of the different tasks.

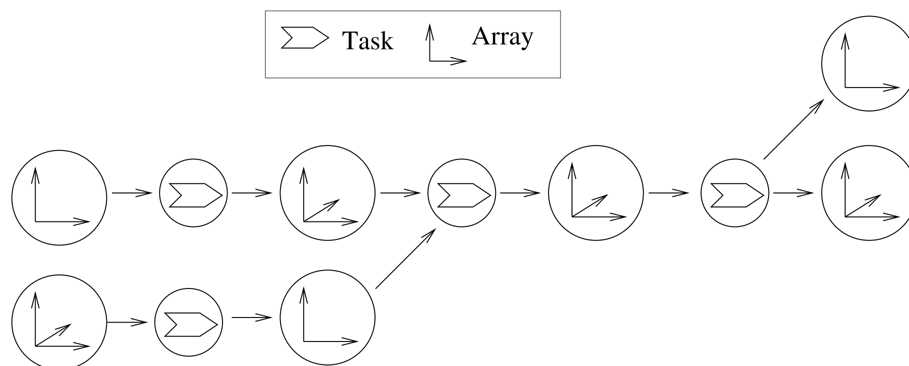


Figure 1: *A global model*

The **global model** is a simple directed acyclic graph where each node represents a task and each edge represents a multidimensional array (figure 1). The number of incoming or outgoing arrays is not limited. These multidimensional arrays may have one infinite dimension that is generally used to represent time.

At the execution time of each task, the incoming arrays are consumed and the output arrays are produced. The number of produced or consumed arrays is equal to the number of inputs or outputs edges for each task. The graph relating to the global model thus represents a task graph and not a data flow graph. There is no implicit repetition of the task graph as in stream languages. The streams are explicit in the arrays (by the infinite dimension for example). The model is thus strictly single assignment at the array element level.

In this model, there is no correlation neither between the number of input and output arrays nor between the number of dimensions of these arrays. Thus it is possible for a task to consume two two-dimensional arrays and produce a three-dimensional array. The creation of dimensions can be very useful, for example in the case of a FFT (Fast Fourier Transformation) which creates a frequential dimension. Moreover, arays used in Array-OL models are *toric* since the consumption or the production of their elements can be made modulo their size.

Using only the global model of the application, it is possible to schedule the execution of the different tasks. However, it is impossible to express the data parallelism present in our application. For this reason, the introduction of the local model becomes necessary.

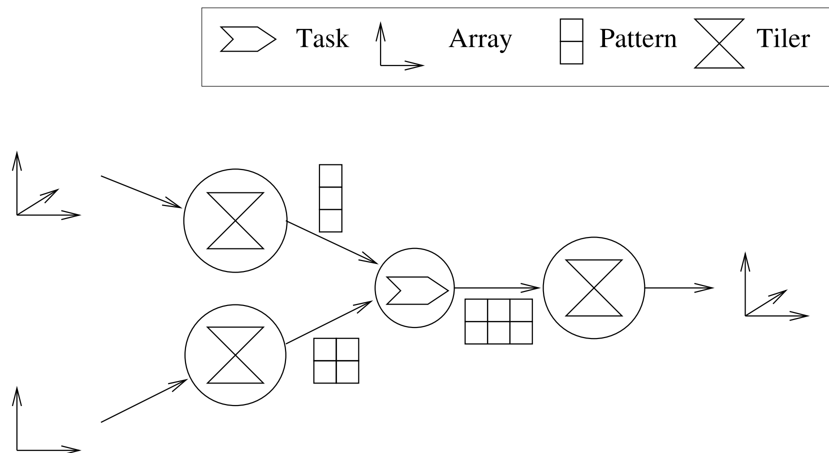


Figure 2: *A local model*

The **local model** allows to express the data parallelism expressed by data parallel repetitions. In this model, the task is always made of a repetitions constructor, where each repetition of the embedded task is independent. That repeated task is applied to a subset of the elements of each input array to produce data elements stored in each output array (figure 2).

The way in which the task consumes and produces its input and output arrays can be analyzed through each couple $(task,\ array)$. Such couples are called *half-task*. Let $(T1,\ A1)$ be a half-task, if $A1$ is an input array, then $T1$ takes a finite subset of $A1$ elements to achieve its processing. In a similar way, if $A1$ is an output array, then $T1$ provides to this array a finite subset of elements which has just calculated. The size and the shape of the element set associated to an array is the same from a repetition to another. In the local model, each element set is

called a *pattern*, and in order to express hierarchical constructions, the patterns are themselves multidimensional arrays.

To each couple $(task, \ array)$ is associated a *tiler* which contains informations necessary to the construction of the different patterns. These informations are as follows:

- $\overrightarrow{o}$: the origin of the reference pattern
- $\overrightarrow{d}$: the shape of the pattern (size of all the dimensions)
- P: a "paving" matrix allowing to describe how the patterns cover the array
- F: a "fitting" matrix describing how to fill the pattern with the array elements
- $\overrightarrow{m}$: the shape of the array (size of all the dimensions)

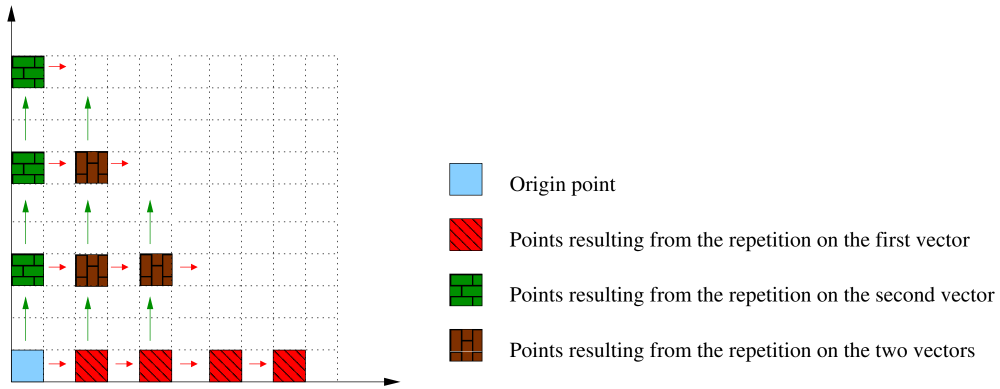


Figure 3: *Paving example*

To enumerate the different patterns, each half-task has, via its tiler, a *paving matrix* and a starting point called *origin*. The paving matrix, P, is composed of a set of paving vectors used to identify the origin of each array pattern, one for each repetition. For example, if we consider a two-dimensional array with as origin point $\left(\begin{smallmatrix}0\\0\end{smallmatrix}\right)$ and as paving matrix $\left(\begin{smallmatrix}2&0\\0&3\end{smallmatrix}\right)$, then the different patterns start at the points shown by figure 3.

The coordinates of the first points of each pattern are calculated as the sum of the coordinates of the origin point and a linear combination of the paving vectors, the whole modulo the size of the array since arrays in Array-OL model are toric as indicated by the equation 1.

$$\forall \overrightarrow{x_q}, \overrightarrow{0} \leq \overrightarrow{x_q} < \overrightarrow{Q}, \overrightarrow{r_q} = (\overrightarrow{o} + P \times \overrightarrow{x_q}) \mod \overrightarrow{m} \quad (1)$$

In this equation, $\overrightarrow{x_q}$ represents the pattern of index q, $\overrightarrow{Q}$ represents the space of repetition, and $\overrightarrow{r_q}$ represents the origin of the pattern of index q.

The *fitting matrix* is represented by a set of vectors where each vector is associated to a pattern dimension. The fitting vectors are used to identify the array elements of each pattern starting from the origin point. Figure 4 represents two simple examples using an unidimensionel array with as origin $(\,0\,)$ and the associated pattern is a two-dimensional array of size $\left(\begin{smallmatrix}3\\2\end{smallmatrix}\right)$. This example shows that it is possible to have more dimensions in the pattern than in the array. In the first case the fitting matrix is $(\,1\ 3\,)$. Each vector of this matrix is used to fill one dimension

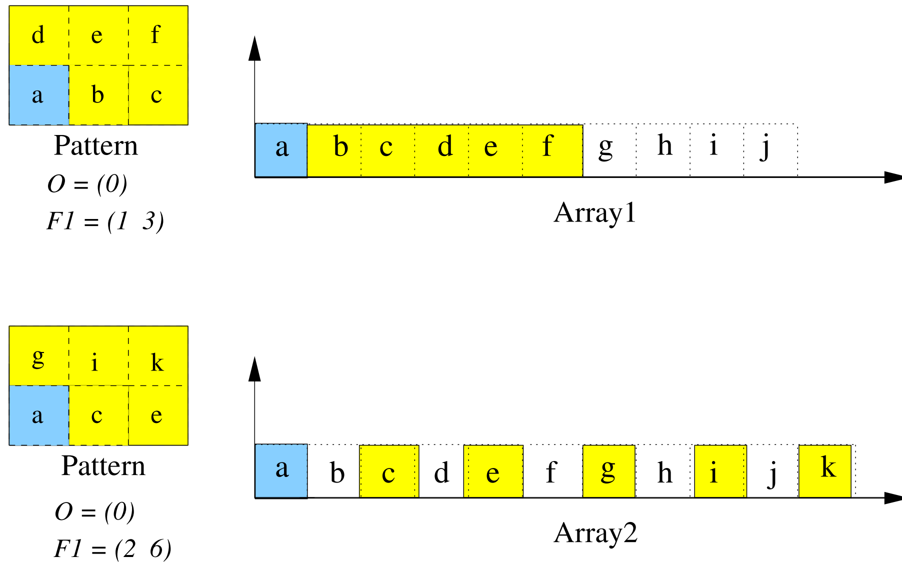


Figure 4: *Fitting example*

of the pattern. In the second case, the fitting matrix is $\begin{pmatrix} 2 & 6 \end{pmatrix}$. In this case, the different elements of a pattern do not correspond to consecutive elements in the array.

The use of the fitting matrix is similar to that of the paving matrix. The array elements constituting a pattern are calculated as the sum of the coordinates of the first element of this pattern and a linear combination of the fitting matrix, the whole modulo the size of the array since arrays in Array-OL model are toric as indicated by the equation 2.

$$\forall \vec{x_d}, \vec{0} \leq \vec{x_d} < \vec{d}, (\vec{r_q} + F \times \vec{x_d}) \mod \vec{m} \tag{2}$$

In this equation, $\vec{x_d}$ represents the pattern element of index d.

The $\mod \vec{m}$ part of the above equations ensures that all the data elements of a pattern correspond to array elements. The modulo allows to handle cases such as toroidal physical spaces or the cyclic frequency dimensions obtained after an FFT or a DCT.

Figure 5 illustrates the combination of the paving and the fitting concepts for two successive repetitions through five examples:

1. A basic case.
2. Elements of patterns are not necessarily parallel to the axes.
3. Two successive patterns intersect.
4. Two successive patterns overlap.
5. A toric array with as origin $\begin{pmatrix} 2 \\ 0 \end{pmatrix}$ and not $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$.

To illustrate the use of the Array-OL language, we study the modeling of an academic example which is the product of two matrices. This simple example makes it possible to show the potency of the local model for the expression of the data parallelism in an algorithm. Let $A1$

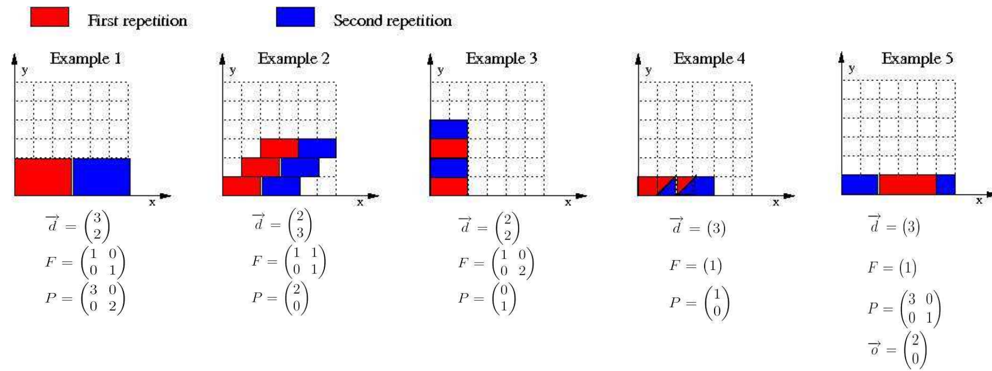


Figure 5: *Fitting examples*

a 3×5 matrix and $A2$ a 5×2 matrix. We calculate the matrix product $A1 \times A2 = A3$ with $A3$ of size 3×2. The calculation of the matrix product boil down to the calculation of the scalar product of each line of $A1$ by each column of $A2$. The different scalar products are independent and can then be performed in parallel.

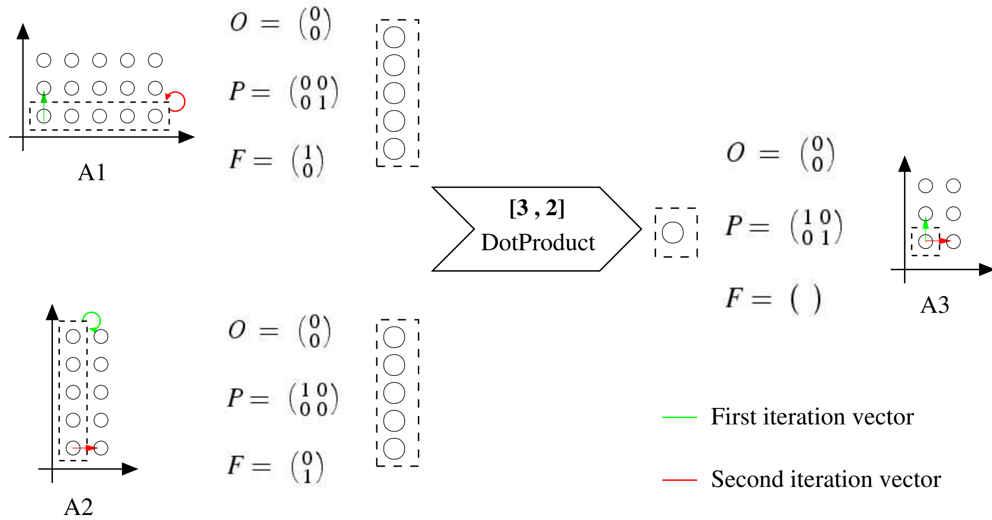


Figure 6: *Matrix product example*

The Array-OL language allows to express all this parallelism by identifying the necessary input patterns to produce the output patterns. Let the elementary task `DotProduct` taking as input two vectors of size 5, and producing as output a scalar corresponding to the scalar product of the two input vectors. In the local model illustrated by figure 6, the repetition space of the task `DotProduct` is defined by the vector $[3,2]$ (one task for the production of each point of the output array). This example shows the possibility to express all the potential parallelism in an application which can facilitate its interpretation by a compilation or optimization tools.

2.1.2 System on Chip Co-design and the Gaspard2 Environment

In the 90's, the evolution of the integration technologies on chips allowed the appearance of a new paradigm in the embedded systems area called *System-on-Chip* (Soc). These systems represent the integration on the same chip of a complex platform which includes programmable processors, memory units (data/instructions), interconnection mechanisms and hardware functional units (Digital Signal Processors, application specific circuits). These components can be generated for a particular application, they can also be obtained from IP (Intellectual Property) providers. The ability to re-use software or hardware components is without any doubt a major asset for a codesign system. SoC design covers a lot of different viewpoints including as much the application modeling by the aggregation of functional components, the assembly of existing physical components, the verification and the simulation of the modeled system.

Gaspard2 is an under development model driven Integrated Development Environment for SoC (System on Chip) visual co-modeling. It is developed in the context of INRIA DaRT[3] project, extends the Array-OL language and allows modeling, simulation, testing and code generation of SoC applications and hardware architectures.

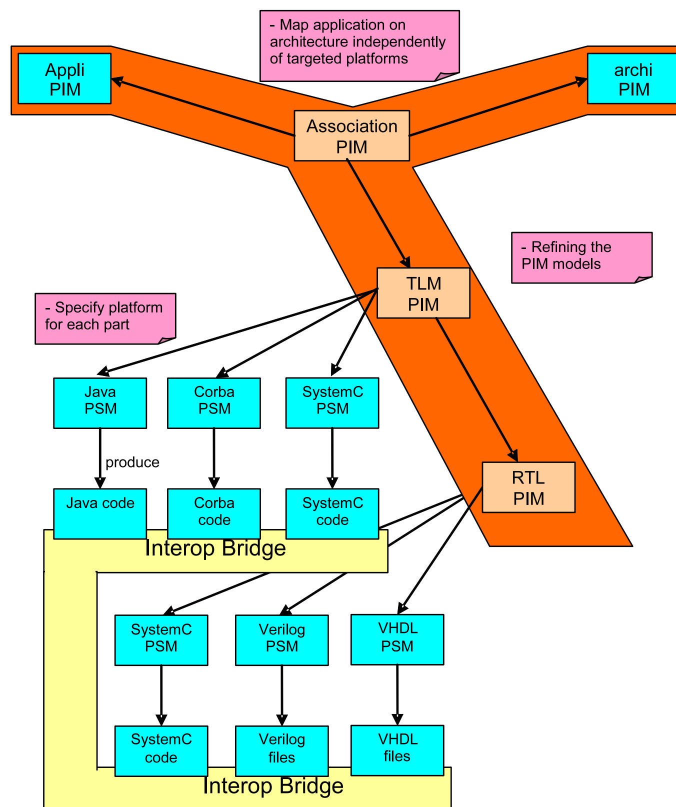


Figure 7: *Representation of the Y model relating to the Gaspard2 flow design*

The Gaspard2 environment is mainly dedicated to the specification of signal processing applications. It is based on a *model oriented* methodology according to a Y design flow (figure 7).

[3] `www.lifl.fr/west`

The Gaspard2 design flow is located at the border of two research areas: the system on chip design, and the model driven engineering. The Y model in Gaspard2 corresponds to a design flow that we generally define in the case of system on chip design. It is mainly defined around three concepts: the *application* which specifies the functionality of the system, the *hardware architecture* which will be used to support the execution of the application and perform its functionalities, and the *association* which specifies the mapping of the application on a given hardware architecture.

In this approach, the concepts and semantics of each design level (application, architecture and association) are platform independent. No component is associated with an execution, simulation or synthesis technology. Such an association targets a given technology (Java, SystemC RTL, SystemC TLM, VHDL, . . .). Once all the components are associated with some technology, the deployment is realized. This is done by the refinement of the PIM association model (Platform Independant Model) to the PIM TLM model first (Transaction Level Model), and to the PIM RTL model second (Register Transfer Level).

The DaRT project contribution in the Gaspard2 design flow can be defined around three main fields: co-design, optimization techniques, and simulation. These three concepts have as a common objective the reduction of the development cycle time, and the proposed solutions take the regular and the parallel aspect of the studied systems into account.

The *co-design* allows the co-modeling of software and hardware architecture parts of high performance systems. Its objective is to propose an environment allowing a high abstraction modeling level for these systems. The applied *optimization techniques* are mainly transformations of a data parallel constructions. These transformations are used to optimize mapping and scheduling of an application on a given architecture, and are strongly inspired by the techniques used in high performance computation area. The *simulation* objective consists in simulating systems made up of differents IPs specified at various abstraction levels, in various languages, and possibly in a distributed way. The simulation also allows to verify the functionalities of the system, and in particular the adequacy of the mapping and scheduling.

The starting point in Gaspard2 consists in modeling the application, the architecture and the association by using a Gaspard2 UML2.0 profile [17]. These models are then imported in an Eclipse[4] plug-in via model transformation to a specific metamodel using ModTransf[5], and studied by applying mapping and scheduling algorithms and automatic SystemC[6] code generation.

The model definitions of the Gaspard2 environment are based on a component oriented methodology. This methodology makes it possible to clearly separate the application and the hardware architecture and facilitate the re-use of existing software and hardware IPs. It also defines an association model that gives directives on the mapping of the application on a particular architecture.

2.1.3 UML Profile for Modeling Gaspard2 Components

In Gaspard2 environment, the *application* and *hardware architecture* are described by different metamodels. Some concepts from these two metamodels are similar in order to unify and so simplify their understanding and use. Models for application and hardware architecture may be done separately. At this point, it becomes possible to map the application model on the hardware architecture model. For this purpose, a third metamodel is introduced, named

[4] `http://www.eclipse.org`
[5] `http://www.lifl.fr/west/modtransf`
[6] `http://www.systemc.org`

association metamodel, to express associations between the functional components and the hardware components.

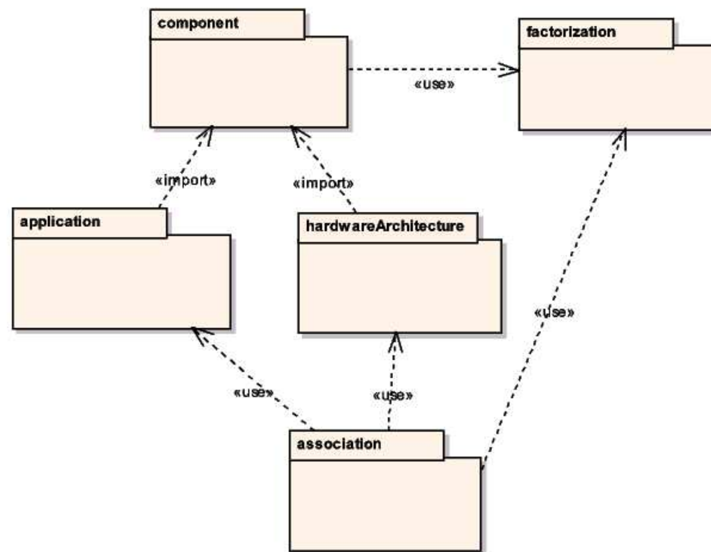


Figure 8: *Different packages of the Gaspard2 profile*

In [18], Arnaud Cuccuru proposes an UML2.0 profile for modeling the various concepts of Gaspard2 models. This profile is defined around five packages: `component`, `factorization`, `application`, `hardwareArchitecture` and `association` as shown by figure 8. The general philosophy of this hierarchy is to define a maximum of common parts for the various aspects of the Y model.

The `component` package contains the basic elements of the component oriented approach used in the Gaspard2 profile. Its main objective consists in favouring the re-use concept by defining a methodology support based on the re-use of software and hardware IPs. This is done by using a set of mechanisms: *encapsulation*, *composition/assembly* and *parametering*. The *encapsulation* allows to make a component independent of the environment in which it is used in order to facilitate and to encourage its re-use. To do that, this mechanism uses two UML concepts: `port` and `interface`. *Composition* and *assembly* describe the structure of the component by using the UML concepts of `part` and `connector`. While the *parametering* makes it possible to associate parameters to components and to fix instance values for these parameters.

The `factorization` package contains structural factorization mechanisms inspired by the Array-OL model. These mechanisms make it possible to express the multidimensional aspect and the relation between the pattern elements of inputs and outputs arrays of a task. In this case, the array paving and fitting mechanisms, used to express the potential parallelism of an application, are generalized by considering that these mechanisms simply make it possible to express dependencies, or links, between the input pattern elements and the output pattern elements in a task. Arrays and data dependencie are respectively regarded as arrays of modeling elements and the relations between these elements.

The `hardwareArchitecture` package allows to specify the hardware architecture supporting the system on a high abstraction level. The objective is to be able to describe the used resources and the topology of their interconnections in order to make rapidly efficient decisions. The `hardwareArchitecture` concept describes a set of hardware components which represent an abstraction of the physical used resources. These resources can be of type `ELEMENTARY` (ex: IPs), `COMPOUND` (executable architecture), or `REPETITIVE` (parallel architecture).

The objective of the `application` package consists in modelling applications by single expression of the data dependencies. This paradigm is directly derived from the Array-OL language. It proposes a component oriented approach based on the definition of the concepts present in the `component` package, and by using the factorization mechanisms described in the `factorization` package. Applications modelled using this profile do not give any information on their execution model. The same application model can be executed sequentialy, in pipe-line, or in parallel.

The `association` package introduces concepts giving directives on the mapping of the application on a hardware architecture. Association concept includes two aspects: the *characterization* and the *allocation*. The objective of the *characterization* is to provide informations used to manage the mapping. While the *allocation* consists in defining the mapping of calculations and data on the most adapted resources by taking the hierarchy and the repetitive aspects of the system into account.

In this profile, we can find the three main concepts of the Y model in Gaspard2: `application`, `hardwareArchitecture` and `association`. The `application` and the `hardwareArchitecture` packages share the same component definition introduced in the `component` package.

In this document, we are only interested in the `application` part. This part allows to model the data dependencies and the parallelism of applications based on the Array-OL model. In the application metamodel, the `ApplicationComponent` concept refines the `Component` concept by adding an applicative connotation. The application components can be seen as a set of functions. These functions perform calculations on the input data coming from their external environment through input ports (*provided* ports in UML terminology) and produce results to their environment through output ports (*required* ports in UML terminology). These application components can be mainly described by using three types of components: `AppElementaryComponent`, `AppComponent` and `AppRepetitiveComponent`.

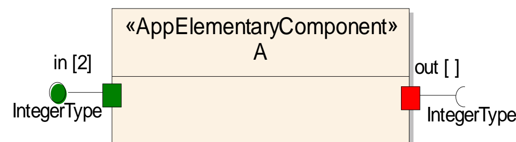


Figure 9: *Example of an* `AppElementaryComponent` *component*

The `AppElementaryComponent` component represents a particular component which does not have any description of structure or behavior. Figure 9 represents a simple example of an `AppElementaryComponent`. This example defines a function taking as input a pattern of two integer elements and produce as result a pattern of only one boolean element.

The `AppComponent` component is used to define compound components. For example, if the result of the elementary task *A* presented in figure 9 is used by another elementary task *B*

which produces a pattern of two boolean elements, then the component AppComponent is used to group and represent the relation between the two tasks A and B as shown by figure 10.

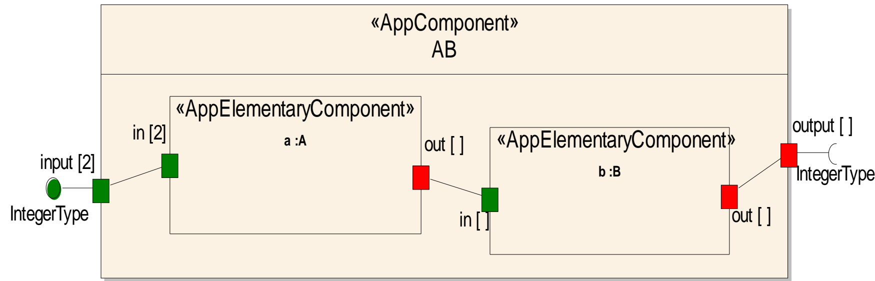


Figure 10: *Example of an* AppComponent *component*

The AppRepetitiveComponent component allows to describe the repetition of the different tasks modeled in this component. This repetition relates to the repetitive concept of the Array-OL model. Thus, by using a special connector (RepetitiveConnector), it is possible to give information on the origin, the paving and the fitting matrices for each input or output array. Figure 11 gives a simple example on the repetition of the AB task presented in figure 10. In this example, the model receives as input an array of two dimensions $8 \times *$ representing an infinity of 8 integer vectors, and produces as result an array of three dimensions $2 \times 4 \times *$ representing an infinity of 2×4 boolean arrays.

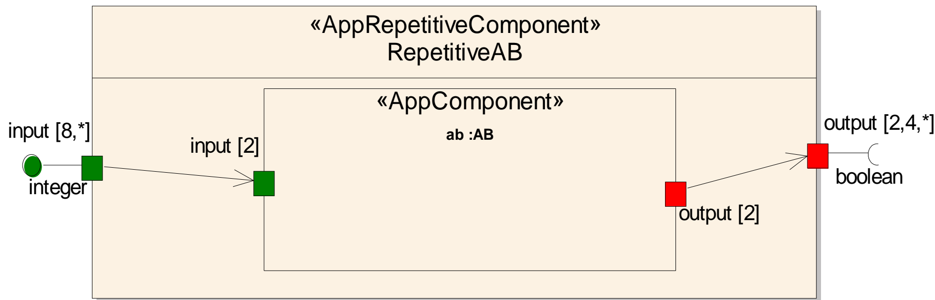


Figure 11: *Example of an* AppRepetitiveComponent *component*

To illustrate the application metamodel concepts, we study a simple example that we call ProdAdd. This example represents the matrix product $A3 = A1 \times A2$ presented in section 2.1.1, and the matrix addition $A5 = A3 + A4$, when $A4$ is of size 3×2. We remember that $A1$ is of size 3×5, $A2$ is of size 5×2, and $A3$ is of size 3×2. The model relating to this application is represented by figure 12. In this application, each point of the matrix result $A5$ can be calculated independently.

The Gaspard2 application metamodel allows to describe the data dependencies and the potential parallelism present in applications. However, this metamodel does not contain any representation of the control and the possibility of changing running modes according to the execution context of the studied applications.

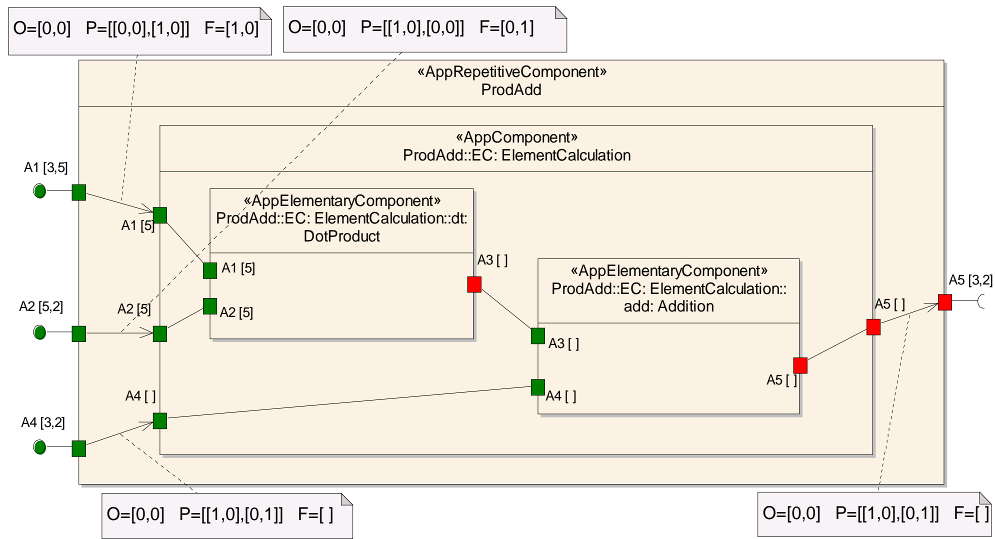


Figure 12: *ProdAdd example*

In the following section, we study the introduction of control into the Gaspard2 application metamodel in order to take more general parallel applications mixing control and data processing into account. The introduction of control into a parallel application can be done by giving a *reactive behavior* which has been largely studied in the case of the synchronous reactive systems.

2.2 Synchronous Real-Time Reactive Systems

2.2.1 Real-Time Reactive Systems

D. Harel et A. Pnueli classify computer systems in three main categories: *transformational*, *interactive* and *reactive* according to their degree of interaction with their environment which can be a human user or a physical process [1]. Historically, transformational and interactive systems come from the traditional programming, to which are added event management mechanisms, synchronization and concurrent programming, while reactive systems result from the electric, automatic and embedded systems area.

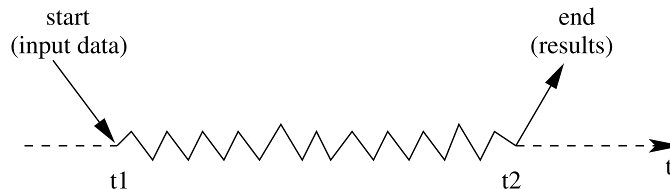


Figure 13: *Execution of a transformational system*

Transformational systems are classical programs generally based on complex data structures and algorithms. They perform calculations starting from the data provided at input points, to

produce results at output points before terminating as shown by figure 13. The interaction with the environment is thus limited to the acquisition of the data and the production of results, as in the case of a compiler for instance.

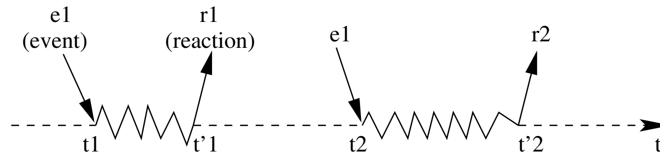


Figure 14: *Execution of an interactive system*

Interactive and *Reactive* systems are computer systems that react continuously to their environment, by producing results at each invocation. These results depend on data provided by the environment, and on the internal state of the system. The difference between these two types of system is in the entity which controls the interaction. In an interactive system, as a data base or an operating system for instance, the production of results and the handling of events follow the initiative of the system which imposes its own rhythm as shown by figure 14. Contrary to these

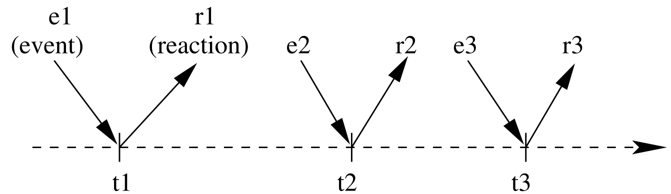


Figure 15: *Execution of a reactive system*

systems, the interaction rhythm of a reactive system is fixed by its environment. A reactive system must always be able to provide an immediate answer when the environment requests it. The evolution of such system is thus a succession of a *reactions* caused by its environment, each reaction is instantaneous compared to the time scale of the environment (figure 15). The human-machine interfaces and the industrial processes represent typical examples of reactive systems.

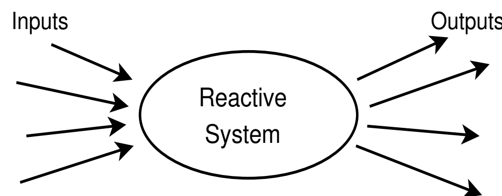


Figure 16: *Representation of a reactive system by a black box*

Reactive systems are systems with *discrete* events. Their behavior can be represented by a sequence of *reactions* to external stimulis. In [1], D. Harel and A. Pnueli have given to reactive systems the image of a black box that react to its environment at a speed determined by the latter (figure 16).

To give prominence to the interaction aspects of reactive systems, it is possible to consider that the execution of such a system always follows the diagram of figure 17. The loop of this

```
<Initialisation>
For each input event Do
    <Calculate outputs>
    <Give results>
```

Figure 17: *Execution schema of a reactive system*

diagram is executed several times, even indefinitely. Since the reactive system is based on the description of the interaction aspects with its environment, it has always an interesting behavior even if its execution does not finish [28]. The execution of this system is generally diveded in cycles corresponding to a discrete logical time scale relating to the execution cycles of the system. This logical time scale does not necessarily correspond to the physical time, and it is then possible to define the input and output series indexed by integers.

Reactive systems are in particular defined in the *real-time* system area. The "real-time" word is often used to qualify interactive applications which must rapidly react to their environment with a satisfactory response time by the user. Real-time reactive system is then a reactive system subjected to additional constraints of time. Avionics, transportation, communication and signal processing systems represent typical examples of real-time reactive systems.

Real-time reactive applications cover a very large number of fields. They are generally present in the embedded systems and have six main characteristics, in particular if they play a critical role by bringing into play human lives (*safety critical*) or fundamental missions (*mission critical*). These characteristics are as follows [29, 30]:

- **Strong temporal constraints**: real-time reactive systems have often an imperative requirement on reaction or execution times. For some applications, the no respect of time constraints is strictly forbidden. For the others, the no respect of times must be systematically detected and replaced by exception processes.
- **Previsibility**: the previsibility allows to anticipate the behavior of the system which must be perfectly *deterministic* by producing exactly the same result sequence for the same event sequence.
- **Reliability**: the critical nature of real-time reactive systems requires that the specification of these systems must certify their behavior, in particular under extreme conditions which increase the breakdown risks of these systems. This is often concretized by doubling or tripling the critical elements.
- **Robustness**: when the environment is not the predicted one or one of the system elements breaks down, the system must be able to switch to a security mode, either by finishing properly the tasks in progress, or by continuousing to provide its main functions.
- **Concurrency**: concurrency is used to specify reactive systems witch components are concurrent and cooperate to define the whole behavior of the system. This characteristic allows users to specify their system in a clear and concise way.
- **Hierarchy**: real-time reactive systems made up of only one block are very rare. For simplification reasons, it is frequent to define a control system as a common orchestra

head for a set of subsystems. Each subsystem manages only one part of the application, and can be, in its turn, decomposed on more elementary systems, and so on...

Specification of software or hardware real-time reactive systems behavior is complex. It can lead to important errors that are difficult to fix. Indeed, such systems are not only described by transformational relationships, specifying outputs from inputs, but also by the links between outputs and inputs via their possible combinations in one step [31]. Modeling reactive systems is therefore a difficult activity.

2.2.2 Synchronous Approach

The development of a real-time reactive application must be a rigorous process. This process requires in particular languages adapted for the specification of these applications and reliable tools for their automatic verification.

In the beginning of the 80's, the family of synchronous languages and formalisms has been a very important contribution to the reactive system area [19]. Synchronous languages have been introduced to make programming reactive systems easier [32]. They are based on the *synchrony hypothesis* that does not take reaction time in consideration, and supposes that each reaction is instantanous and atomic. In this case, each activity can then be dated on the discrete time scale.

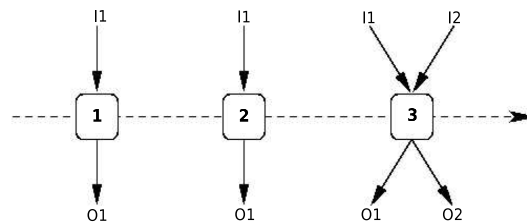


Figure 18: *Execution example of a synchronous system*

The synchronous hypothesis defines a discrete logical time scale consisted of *instants* corresponding to each reaction of the system. The events that trigger the reaction are *simultaneous*, and the reaction time of a particular component of the system and the communication time between components are *null*. A reaction is thus by construction instantaneous and atomic, which avoids concurrent and partial reactions of the system, origin of indeterminism [33]. In this case, and for each reaction, several events can be taken into account to produce instantaneously results, and any reaction in progress must be finished before another reaction can be started. Figure 18 illustrates this property. The main characteristic of this hypothesis is the possibility to compose synchronous systems to obtain other synchronous systems whose behavior is deterministic and perfectly defined.

In [34], the synchronous approach has been presented as a good solution to the reactive programming problem. The synchrony hypothesis has an abstract view on the interactions and makes it possible to simplify the expression of reactive behaviors. This hypothesis supposes that synchronous systems are very well composed. They are easier to describe, to simulate and to verify than asynchronous systems.

Synchronous languages are devoted to the design, programming and validation of reactive systems. They have a formal semantics and can be efficiently compiled into C code, for instance. Moreover, these formalisms make it possible to validate and verify formally the behavior of the

system. In this field, we often speak about tools and approaches for simulation, verification and code generation for reactive systems specified in a synchronous language. These languages can be classified into two main families: *declarative languages* and *imperative languages*.

Declarative or data flow languages like Lustre [35, 36, 37], Signal [38, 39, 40], SynDEx [41] and the HPTS model [42] are used when the behavior of the system to be described has some regularity like in signal-processing. Their main task consists in consuming data, performing calculations and producing results.

Imperative or control flow languages like Esterel [43, 44, 45], Argos [46], SyncCharts [47] and StateCharts [48] are more appropriate for programming systems with discrete changes and whose control is dominant: for instance coffee machines. Their purpose is to manage the processing of data by imposing an execution order to operations and by choosing one operation among several exclusive.

The mathematical fondation of synchronous languages guarantee noticeable properties for the high-level constructions [45]. The parallel composition of the Esterel language for example is perfectly deterministic, whereas competition in the asynchronous languages generally introduces indeterminism. The compilers are based on the synchronous language semantics to produce deterministic and efficient execution code, and to verify some properties by detecting for example the deadlock situations. This mathematical rigorous semantics made it possible to build automatic tools for formal verification allowing to test the programs before their execution.

Before the synchronous model has been proposed, the more used approaches to implement reactive systems were automata, high-level languages with executives real-time multitasks and the parallel asynchronous languages. These techniques were largely extended to adapt them to reactive descriptions. However, according to [49], the traditional techniques do not bring satisfactory solutions to the real-time reactive systems programming. It is largely preferable to develop dedicated languages rather than to adapt existing languages. Dedicated languages to synchronous programming allow to take directly the specificities of these systems into account on a high abstraction level, in particular the determinism and the fine control of reactive processes.

2.2.3 Synchronous Behavior and Automaton Structure UML Modeling

The UML modeling of the synchronous behavior and its concepts are an interesting research subject. For example, in [20], R. De Simone and C. André propose a UML subprofile, using a synchronous version of state and activity diagrams, to express the synchronous reactive behavior. Their proposition gives a synchronous solution to the UML state machine limitations by allowing the description of the absence and the simultaneous events.

In UML modeling, the term *reactive* is applied to objects that respond dynamically to incoming events of interest and whose behavior is driven by the order of arrival of those events. Such objects are usually modeled and often implemented as finite state machines (FSM) called *StateCharts* [9].

The UML StateCharts specify a set of concepts used for modeling discrete behavior through finite state-transition systems. They are an object-based variant of Harel StateCharts [5]. The semantics of the UML StateCharts are described in terms of the operations of the hypothetical machine that implements a state machine specification. In this state machine, states represent the existence conditions of the class they define, and the transitions are represented by directed arcs with named event triggers optionally followed by actions. Figure 19 gives a small example of an UML StateCharts.

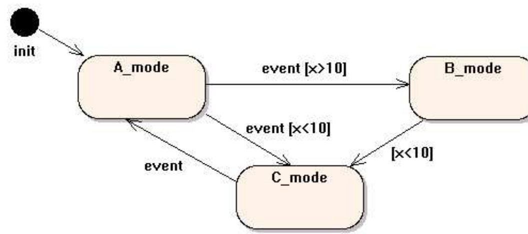


Figure 19: *Simple example of a StateChart*

The finite state-transition system specifies the events of interest to a reactive object, the set of states that object may assume, and the actions (and their order of execution) in response to incoming events in any given state. This is crucial in many systems because the allowable sequences of primitive behaviors may be restricted.

The StateCharts example of figure 19 specifies the various states (shown as rounded rectangles) and the transitions (arrows) that indicate how the system responds to different events. We see that the system responds to the event `event` according to the X value. A state is a condition of existence of an instance that is distinguishable from other such conditions.

The object may execute actions when a state is entred or exited, when an event is received (although a transition isn't taken), or when a transition is taken. The actions may be any kind of action defined in UML, like call actions (to invoke a method defined in this or another object), event generation, or even the execution of a primitive action statement such as $++X$.

If an event occurs while the system is in a state that doesn't specifically handle that event, it is ignored. The syntax for transitions is:

$$event-name\ (parameter-list)\ [guard-condition]\ /\ action-expression$$

The $event-name$ is the name of an occurrence of interest that's processed by the object's state machine, such as `event`. If the $event-name$ is left blank, the event fires as soon as the state is reached. When the object is in the predecessor state (from which the transition arises) and the named event is sent to the object, the transition fires and the object moves to the new state executing the specified actions along the way. Parameters may be passed to the object along with the event and manipulated in the actions on the transition. A Boolean expression inside the square brackets is a guard. If the event fires and the guard evaluates to TRUE, the transition is taken; if the guard evaluates to FALSE, the event is ignored.

Responses to events may have a time value associated with them (via a UML constraint), but the main characteristic of the UML StateCharts is that they have a *run-to-completion* semantics which imposes that no other event can be taken into account before the processing of the previous event is fully completed. This means that once an event is received and the object determines that it will respond to it, the object executes first the exit actions of the predecessor state, followed by the transition actions, and finally the entry actions of the subsequent state, in that precise order, and not processes any new events until that set of actions has completed. The different actions are normally specified in the implementation language. This assumption simplifies the transition function since concurrency conflicts are avoided during the processing of events. Moreover, the absence of an event instance cannot be taken into account in the UML StateCharts which makes it difficult to express highly reactive system behavior.

Statecharts are a great way to specify behavior as a set of actions to be executed when an object is in some given state. The statechart remembers the condition of existence for the object as the object's state. Through the use of separation of the object's lifecycle into different states, hierarchical nesting of states, and the *or-states* and *and-states*, you can model any mundane or highly complex behavior.

The only reason for which we have chosen to use the UML StateCharts model is the simplicity of this model which has a well defined semantics. However, it is always possible to model the control part by using a more sophisticated UML metamodel for the specification of the automaton structures as for example the UML subprofile introduced in [20].

2.3 Mixing Control and Data Processing

In section 2.2.2, we have presented two synchronous language families mainly dedicated to the specifications of regular or discrete behaviors for embedded reactive systems. However, rarely these systems have an exclusively regular or discrete behavior. The most realistic and used embedded systems combine control and data processing. Such global systems may be totally specified with imperative languages, but data dependences between operations can not be clearly specified and furthermore problems may occur due to shared variables. Similarly, they may be totally specified with declarative languages, but the control is hidden in data dependencies making it difficult to specify tests and branchings necessary for verification or optimization purposes. For these reasons, we need efficient tools and methods taking in consideration this kind of systems.

Several approaches have been proposed in this domain. We can find the *multi-languages* approach which combines imperative and declarative languages, like using Lustre and Argos [22]. In [50], Leszek Holenderski and Axel Poigné present a multi-paradigm language for programming synchronous reactive systems, called LEA. It is obtained by integrating three existing synchronous programming languages: Lustre, Esterel and Argos. [30] studies the multi-model programming using Esterel and Argos through a production cell example.

The multi-languages approach is based on a linking mechanism and allows the re-use of existing code. However, when using several languages it is very difficult to ensure that the set of corresponding generated codes will satisfy the global specification. Another design method consists in using a *transformational* approach which allows the use of both types of languages for specification but, before code generation, the imperative specifications must be translated into declarative specifications, or vice-versa, allowing to generate a unique code instead of multiple ones. N. Pernet and Y. Sorel give in [23] an example of this approch which translates SyncCharts, a control flow language, into SynDEx, a data flow language which allows automatic distributed code generation.

The transformational approach is efficient for describing reactive systems combining control and data processing. However, there are systems whose behavior is mainly regular but can switch instantaneously from a behavior to another. They are the systems with *running modes*. The most adapted method to describe this kind of system consists in using a *multi-styles* approach which makes it possible to describe with only one language the various behaviors of the system. The Mode-Automata represent a significant contribution in this field. Their goal consists in adding an automaton structure to Lustre programs. [51] shows how two different programming styles for reactive systems may be mixed, having a common semantical basis.

Presented works study the control and data processing combination. However, all these studies do not take the data parallel processing into account. The parallel processing represents

generally a set of tasks which can be executed in parallel to define the global behavior of the system like in signal and image processing.

In the literature, few works have been proposed to introduce the control into a parallel computation field. For instance, in 1998, Smarandache studies application co-design using the Signal relational language and the Alpha functional language [14]. This approach uses the C language as a support of communication between the two levels of specification and does not define any specification model allowing the modeling of parallel applications with the control concepts. Another example can be found in Ptolemy [13] which proposes a multidimensional computation model (MDSDF) and an automata model (FSM). However, the combination of these two concepts has never been studied.

In this document, we choose to use the concept of Mode-Automata, allowing to describe the different running modes of the system, to introduce the control concepts and the reactive behavior to the Gaspard2 application metamodel. This study allows to take into account the different changes of mode in a parallel application metamodel.

2.3.1 Mode-Automata

One way of facing the complexity of a system is to decompose it into several "independent" tasks. Of course the tasks are never completely independent, but it should be possible to find a decomposition in which the tasks are not too strongly connected with each other. Different formalisms are used in the reliability engineering framework in order to design these models of systems under study: *boolean formalisms* like block diagrams, and *states/transitions formalisms* like Petri nets.

Mode-Automata have been proposed in [4]. They introduce, in the domain-specific data-flow language Lustre for reactive systems, a new construct devoted to the expression of *running modes*. It corresponds to the fact that several definitions (equations) may exist for the same output, that should be used at distinct periods of time. This concept allows to decompose the specification of the system into several tasks called *modes* by assigning data operations to discrete states.

A Mode-Automaton is an input/output automaton. It has a finite number of states, that are called *modes*. At each moment, the system is in one and only one mode, and can change its mode if an event occurs. For each mode, a transfer function determines the values of output flows from the values of input flows. Mode-Automata can be combined in order to design hierarchical models. The structure of Mode-Automata allows to clearly specify where the modes differ and the conditions for changing modes which makes it possible to better understand the behavior of the system.

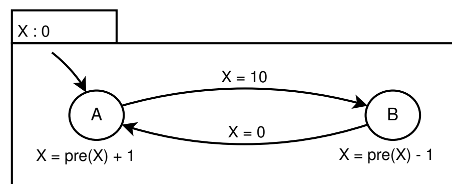


Figure 20: *Mode-automaton: simple example*

Figure 20 represents a simple example of mode-automaton. It has two states, and equations attached to them. The transitions are labeled by conditions on X. The important point is that

X and its memory are *global* to both states. The only thing that changes when the automaton switches its state is the transition function; the memory is preserved.

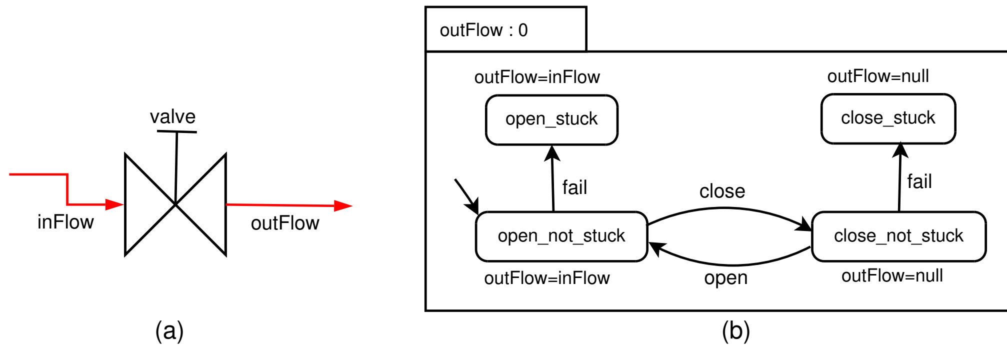


Figure 21: *Valve example*

Consider, for instance, the valve example presented in [52] (figure 21.(a)). Assume that it can be either open or closed and that it may be stuck, either open or closed. The valve changes from open to closed (resp. from close to open) if it is not stuck and if the event `close` (resp. `open`) occurs. It gets stuck when the event `fail` occurs. If the valve is open, its output flow equals its input flow. Otherwise, its output flow is null. Figure 21.(b) shows the Mode-Automaton that describes such a valve.

Mode-Automata are a programming model made of operations on automata taken from the definition of Argos, and data flow equations taken from Lustre. The notion of running mode corresponds to the fact that there may exist several definitions (equations) for the same output, that should be used in distinct periods of time. Faced with this kind of system, users usually write Lustre programs in which modes are encoded by Boolean flows, and the outputs that depend on modes are described by equations of the following form:

$$X \; = \; if \; (mode1) \; then \; \ldots \; else \; if \; (mode2) \; then \; \ldots$$

If several variables have the same modes, other equations with the same conditional structure are added, and the mode-structure is duplicated. There was an obvious need for something more readable and modifiable than this encoding of modes by conditional structures.

2.3.2 Control/Data Flow Separation Methodology

In [12], we have proposed a new design methodology for complex synchronous reactive systems. Our approach is based on a clear separation between control and data flow parts, and combines Scade [25] with the concept of Mode-Automata [4].

Scade is a graphical development environment commercialized by Esterel Technologies[7]. It couples both data processing and state machines styles modeled respectively by the synchronous languages Lustre and Esterel. Scade was defined to help and assist the development of critical embedded systems. It provides several tools, specially for simulation, verification and code generation. Mode-Automata, as described in section 2.3.1, allow to decompose the specification of the system into several tasks by assigning data operations directly to discrete states. The

[7] `www.esterel-technologies.com`

structure of Mode-Automta allows to clearly specify where the modes differ and the conditions for changing modes wich makes it possible to better understand the behavior of the system.

We have shown that the currently available syntax of Scade does not follow a separation design methodology and leads to a mixture of control and data flow representation. This mixture can make difficult the understanding of the system and the re-use of existing applications. To fill this gap, we have proposed to introduce the concept of running modes into Scade specifications to combine the advantages of the two approaches, and to develop a new design methodology separating control and data flows parts.

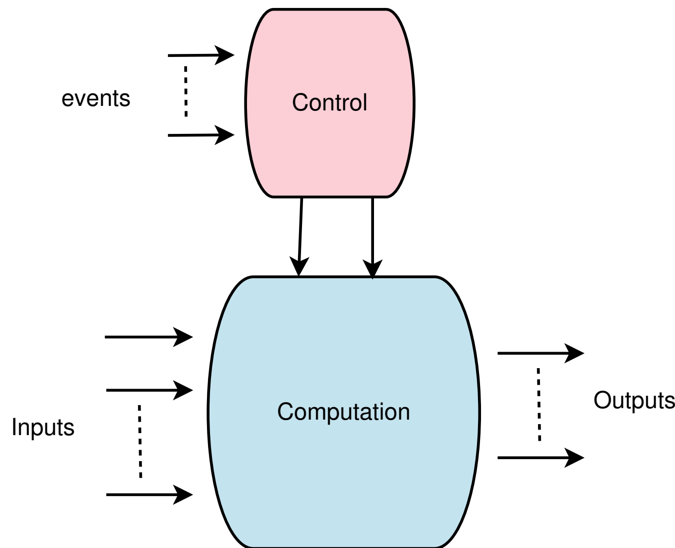


Figure 22: *Global view of a control/data flow separation model*

To do that, and by studying Scade and Mode-Automata, we have noticed that it is necessary to introduce special operators in Scade models to be able to take into account the concept of running modes. In this approach, we can clearly distinguish inputs and outputs of the system, control parts, and data parts as shown by figure 22. Our model is also hierarchical, and the lowest level in the hierarchy represents an homogeneous part that can exclusively contain control or elementary calculation.

The introduction of a design methodology separating clearly control and data flow parts allows to have a more readable model. The different parts of this model can be studied separately by using the most appropriate existing tools for each part.

Morever, a modular specification of the different parts of the system allows to benefit from the modular development. It facilitates the re-use of existing applications, the modification, the introduction and the deletion of modes.

This technique allows to simulate and verify separately the different parts of the system, and consequently have a considerable gain in verification time and memory capacities since the number of states of the verified module is much smaller than that of the complete system. The automaton structure is also exclusive, and to each state of the automaton is associated only one activity. The diffe+rent activities are then exclusive and can be studied separately. This methodology facilitates also the localisation of the different errors while avoiding the modification of the whole application which can be time and resource consuming.

In the following sections, we study the introduction of the control concepts in the data parallel systems, and in particular, in the Gaspard2 application metamodel. To do that, we use the different concepts presented in the previous section which can be summarized in the synchronous reactive approach, Mode-Automata, control/data separation and the UML modeling.

3 Introducing Control in the Gaspard2 Data-Parallel Metamodel

In this section, we study the introduction of the control models into the Gaspard2 application metamodel. To do that, it is necessary to define a modeling concept for the control parts, the different running modes and the link between control and parallel processing.

3.1 Modeling of the Control Part

The control part represents an automaton structure based on the Mode-Automata concept. It allows to clearly specify the various running modes of the system and the switching conditions between modes. For modeling this part and introducing it into the Gaspard2 application metamodel, we define a particular component stereotyped `ControlComponent`. This component produces a *mode array*, possibly multidimensional, depending on the input events and its current mode as shown by figure 23. To each `ControlComponent` component is associated the

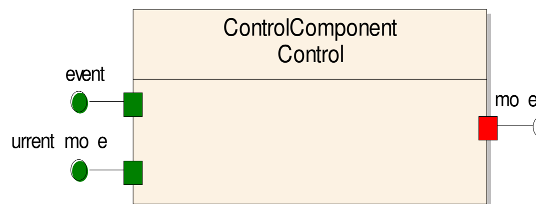


Figure 23: *Example of a* `ControlComponent` *component*

transition function of the control automaton. In other words, it *performs one step* of the automaton. In our UML metamodel, this can be represented by an *activity diagram* [9]. Figure 24 gives a simple example of a control automaton and the representation of the behavior of its transition function by an activity diagram.

Modeling the repetition of the control component, when the automaton *repeatedly performs steps*, consists in introducing a *dependency relation* between the various instances of the transition function. In the Gaspard2 application metamodel, this dependency relation is called *inter-repetition dependency*, and it introduces a regular and total order relation on the different repetitions of the task on which it is defined.

The inter-repetition dependency shows that the calculation of each repetition of index i depends on the calculation result of an other repetition of index $i-n$, with $0 < n \leq i$. This concept can only have significance inside a repetitive component, and in the case of a control component, n always equals to 1 since the goal of an automaton is to memorize its previous state after each execution time.

Using this concept, the mode-automaton structure can be represented by a transition and an *inter-repetition dependency* between the different instances as shown by figure 25. In this

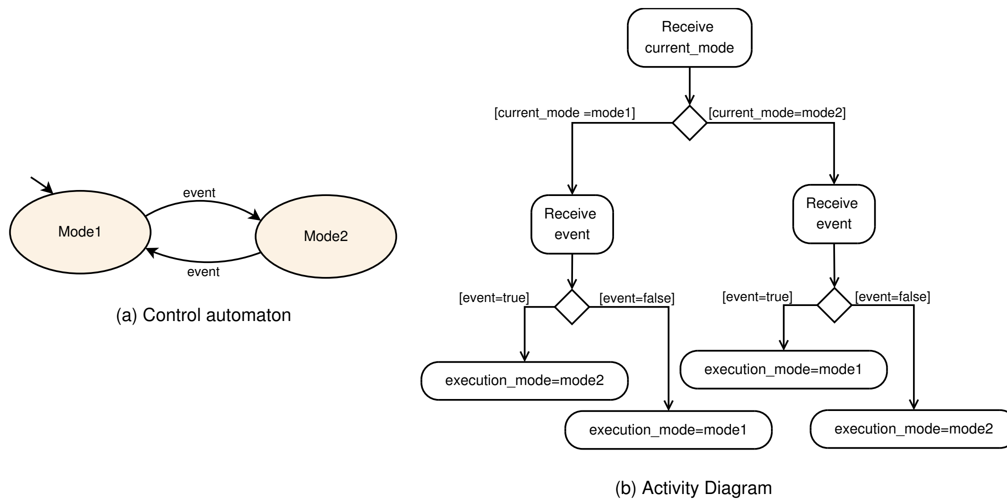


Figure 24: *Representation of the transition function of a control automaton by an activity diagram*

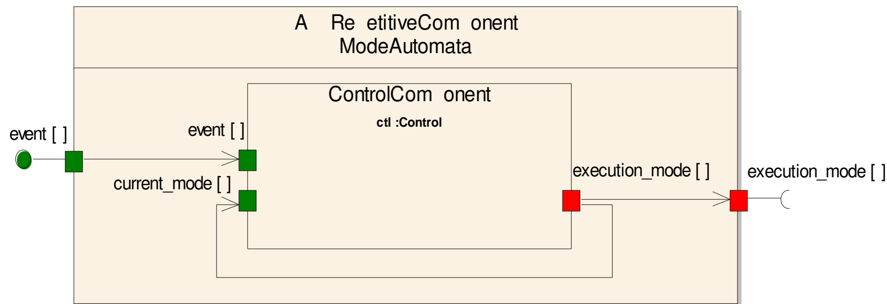


Figure 25: *Representation of the control repetition*

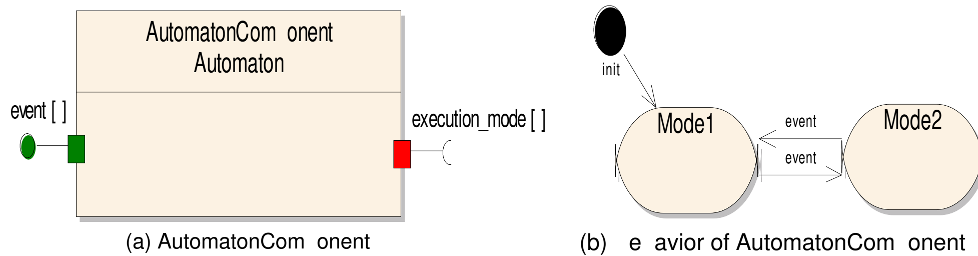
model, an additional information `InitialMode` is introduced on the inter-repetition dependency relation to specify the initial mode of the controlled task.

In the case of a more complex control automata, it is difficult to understand the control model if we represent the automaton structure by a `ControlComponent` component and a dependency relation, and its behavior by an activity diagram. For clarity reasons, it is preferable to represent the control part by an explicit automaton structure in terms of states and transitions.

For these reasons, we propose to introduce, in the application metamodel, a particular component stereotyped `AutomatonComponent`. This component receives as input one or more event arrays and produces as output a mode array. To keep the general semantics of a reactive control automaton, the input and output arrays of the `AutomatonComponent` are regarded as *data flows*. This hypothesis gives to this component a different semantics from that of the Gaspard2 applications.

In the Gaspard2 application metamodel, the dimensions of the arrays represent indifferently time or space. The order of execution is only constrained by the data dependencies. However, in

the control automaton structure, the introduction of the inter-repetition dependency between the different instances of the transition function imposes the introduction of the flow concept to the input and output arrays of an automaton component. This dependency relation makes it possible to memorize the previous states of the automaton and then to respect the general semantics of a control automaton. In our metamodel, and when the control part is described by an automaton structure, we consider that the flow concept is implicitly described for the different input and output arrays of the automaton component. To model the behavior of the control automaton in our metamodel, we propose to use the UML StateCharts [9] structure as shown by figure 26.



(a) AutomatonCom onent (b) e avior of AutomatonCom onent

Figure 26: *Representation of the control automata by StateChart*

Representing the control part behavior by an activity diagram or a StateCharts can be interesting since it gives a more detailed description on the behavior of the application. However, it is also possible to consider the control part, represented by a `ControlComponent` component or an `AutomatonComponent` component, as an elementary component whose behavior is directly represented in the selected implementation language. This concept is similar to that used in the case of the `AppElementaryComponent` component since it avoids users to specify the behavior of the control parts by considering them as a black boxes.

3.2 Modeling of the different Running Modes

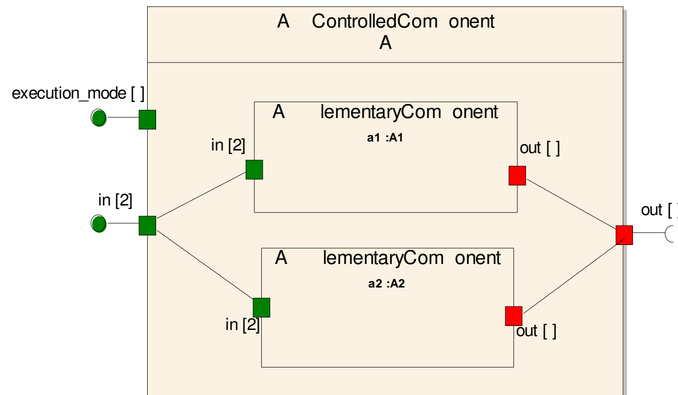


Figure 27: *Representation of the different running modes*

The controlled application, which can be replaced by different running modes, is represented by a particular component stereotyped `AppControlledComponent`. This component consists of several running modes, each mode being represented by a *part*[8] relating to the predefined Gaspard2 components. The different parts in the same `AppControlledComponent` component must have the same interface and are not connected between them. At each moment, one and only one part is activated at the same time according to the mode information available on the `Mode` port. Figure 27 represents the modeling of two running modes $A1$ and $A2$ for an elementary task A.

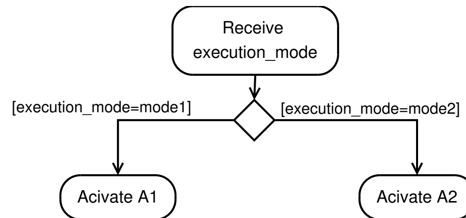


Figure 28: *Behavior of the `AppControlledComponent` component A*

As shown by figure 27, the component `AppControlledComponent` has a particular port stereotyped `Mode`. This port makes it possible to specify the running mode to be activated, it is never connected and is only used by an *switch* associated to the `AppControlledComponent` component to express its behavior. The switch behavior can be represented by an activity diagram in our UML metamodel as shown by figure 28.

The `AppControlledComponent` component has a particular semantics different from that used in the other Gaspard2 components. It is the only component which allows to link different mode outputs ports to the same corresponding output port of the controlled component. Since the interpretation associated to the `AppControlledComponent` component supposes that only one mode can be activated at the same time, each output port in this component has only one definition at a given time, which respects the basic semantics of the Gaspartd2 models.

The expression of the repetitive factor around the component `AppControlledComponent` just consists in using the Gaspard2 predefined repetitive component stereotyped `AppRepetitiveComponent` as shown by figure 29. This figure represents a particular case in which only one control value is used for the calculation of an output image. In other words, all points of the output image are calculated in the same calculation mode according to the value of `execution_mode`.

3.3 Modeling the Link between the Control Part and the Different Running Modes

At each computation time, one and only one running mode is activated according to the information provided by the control part. This information can be the name of the mode to be activated or any other index allowing to distinguish the modes in a clear and single way. In our metamodel, we suppose that the control part provides to the computation part an information on the name of the mode to activate. According to this information, the computation part can activate or not the various modes of the system.

[8]UML concept specifying an instance of a component

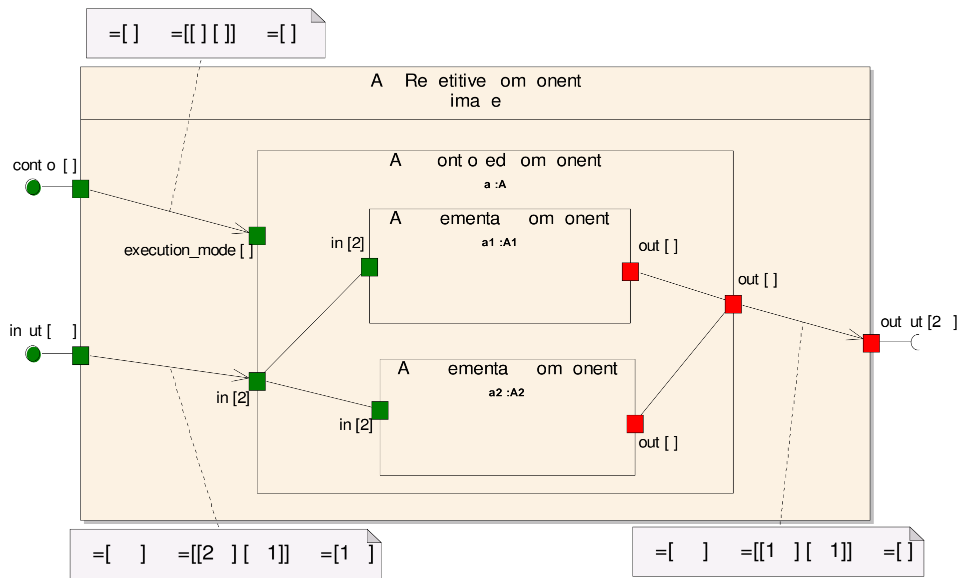


Figure 29: *Representation of the repetition of the* `AppControledComponent` *component*

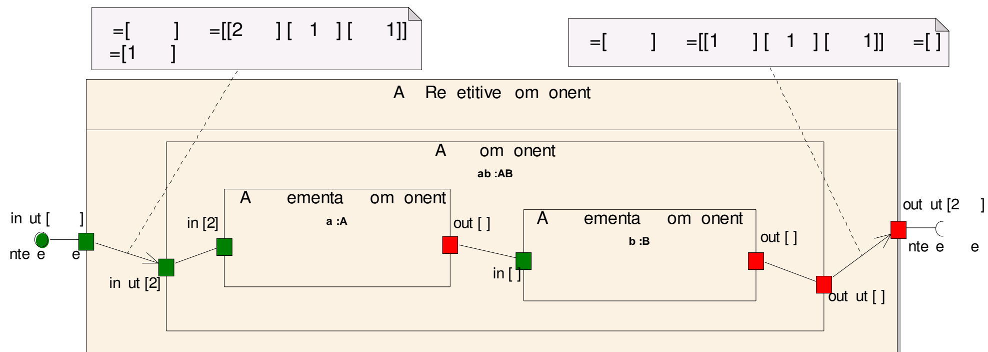


Figure 30: *Simple example of an Array-OL model: UML model*

For a better understanding of this concept, we consider the simple example of an Array-OL model represented by figure 30. In this model, the system takes as input a three dimensional array of $4 \times 4 \times *$ and returns as output a three dimensional array of $4 \times 2 \times *$. The executed task T is a parallel and repetitive one. For each repetition, an instance of the compouned task AB processes an input pattern of two elements to produce an output pattern of one element as explained by figure 31.

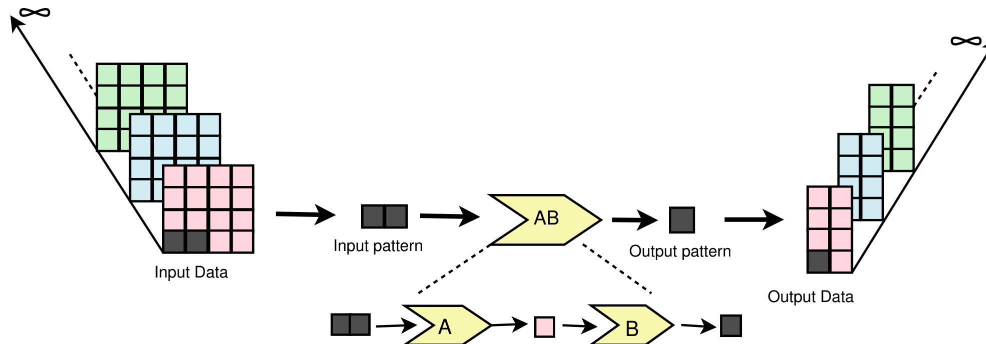


Figure 31: *Simple example of an Array-OL model: Global view*

In the following, we replace the elementary task A by a controlled one with two different running modes $A1$ and $A2$. This task is controlled by a control component `ControlAB` as shown by figure 32. In this example, the control and the controlled parts are represented in the same repetitive component, and we notice that the inter-repetition dependency relation on the control component is defined on a higher hierarchy level (on the `AppComponent` component) since this dependency relation can have only significance inside a repetitive component. This explains the introduction of the new ports on the `ControlAB` component to make possible the expression of this dependency relation and its use by the `ControlComponent` component.

It is also possible to separately represent the repetitive part of the control and that of the computation as it is shown by figure 33. The two representations (figures 32 and 33) are equivalent since they represent the same behavior. In this case, users can choose between these two modeling ways according to their studied applications and the possible re-use of existing models.

Figures 32 and 33 represent the case where the control part is modeled by a component of type `ControlComponent`. This representation completely respects the semantics of the Gaspard2 application metamodel. In this case, the model allows to express all the potential parallelism of an application without any order constraints. These constraints can be introduced explicitly by using an inter-repetition dependency on the control component. However, if we want to represent the control part by an automaton structure modeled by a StateChart for example, we must use the particular component `AutomatonComponent`. This component introduces a new semantics different from that of the other components in the Gaspard2 metamodel. As we consider that any `AutomatonComponent` component consumes and produces a set of data *flow*, the flow and order concepts are implicitly introduced in the model without using the dependency relation since the controlled part in the application must follow the rhythm of the control part. In this particular case, it becomes necessary to represent the two parts, the automaton and the repetitive computation, in a separated way as shown by figure 34.

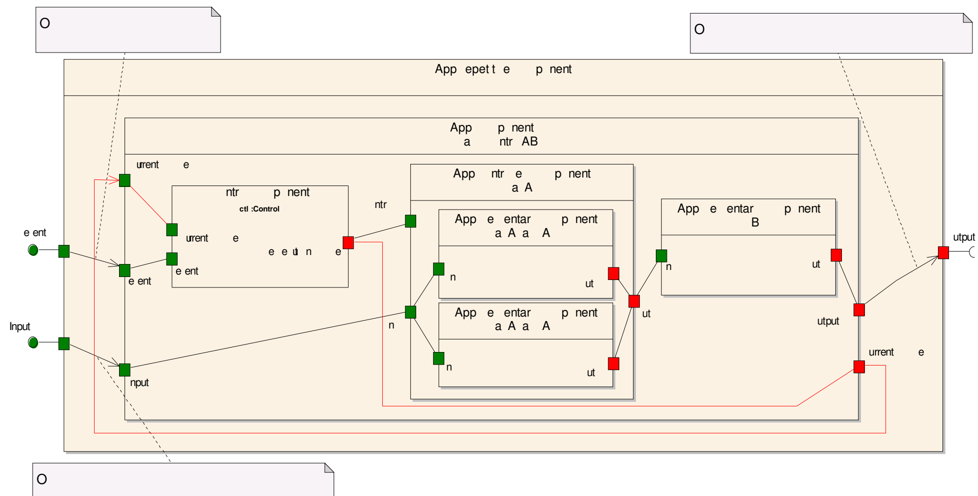


Figure 32: *Representation of the control part and the running modes in the same repetitive component*

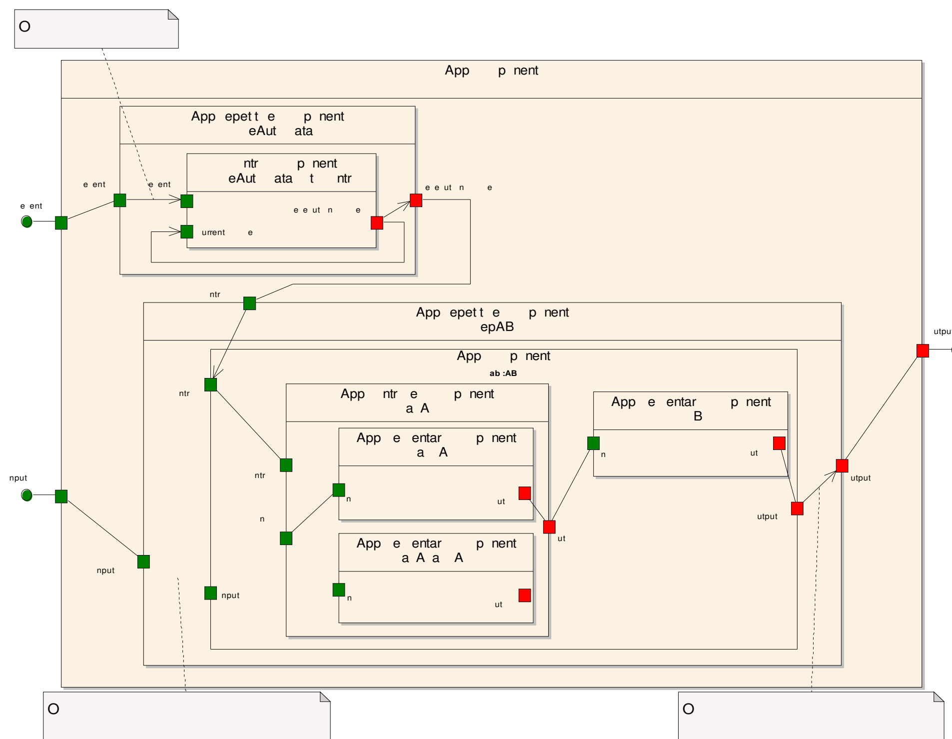


Figure 33: *Representation of the control part and the running modes in different repetitive components*

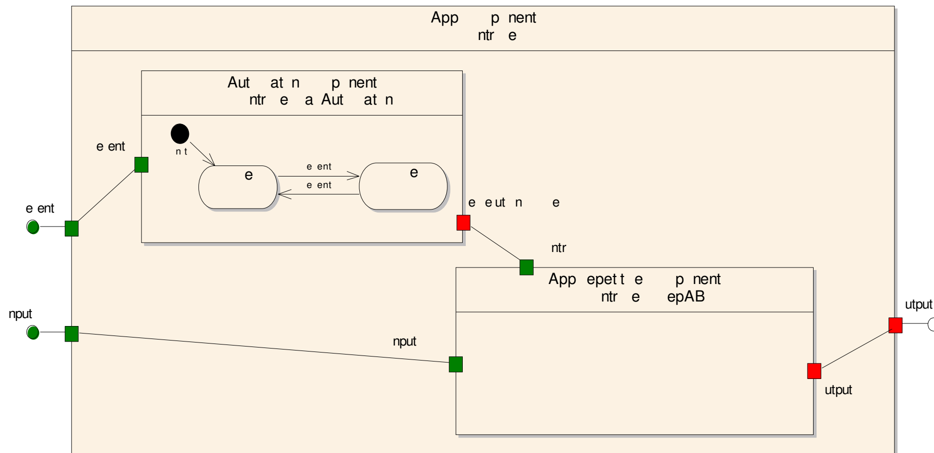


Figure 34: *Representation of the control part and the running modes using an automaton structure*

The introduction of the control concepts in the Gaspard2 application metamodel supposes that the control values (mode array) must be present with the data values to launch the calculation model. The control part must also follow the arrival rate of the events since a control dependency is defined on the various repetitions of the control automaton. This approach imposes the introduction of the `flow` concept in the Gaspard2 application metamodel which can break the assumption of the *unified space-time specification* by imposing a partial order on the execution of the different parallel tasks. This unification is one of the main characteristics of the Gaspard2 metamodel and can be useful for modeling more general applications. However, the introduction of the `flow` concept into the metamodel can facilitate the understanding of the model and makes it more realistic since the input values, either control or data, are mainly generated by sensors and thus represent a control or a data flow structure. Moreover, the logical division of time between discrete instants allows to properly define mathematical models and operational semantics. Our approach also strictly respects the parallelism and concurrency of the model. It is deterministic, compositional and can easily be introduced into Gaspard2 application metamodel.

4 Degrees of Granularity and Control Dependency for the Control of Parallel Applications

The introduction of the control into data parallelism applications requires the definition of a *degree of granularity* or a *control dependency* for these applications. This concept allows to delimit the different execution cycles or *clock signals* in which it becomes possible to take the control values and then the various changes in the running modes into account.

Changing the calculation modes expresses generally the reaction of an application to its execution context. This context can depend on an event from its external environment, or on an internal calculation result. In both cases, it is important to delimit the various moments in which the change of modes become possible, in particular in the case of parallel applications.

In this section, we introduce a controlled component concept by studying different points of view: external and internal control.

4.1 External Control: Changing Modes According to the External Environment

According to the studied application and the selected semantics, several definitions of the degree of granularity are possible. In this section, we study a particular approach allowing to define the various moments at which it becomes possible to take the changes of modes into account. This approach, which we call *synchronous approach*, supposes that the data and control values are available at the same time and follow the same basic clock. In this context, the control model produces a *mode table* which is used by the application to determine the execution mode for the different repetitions. In other words, the mode table only represents an input data like all other input computation data.

The proposed model can have a first very simplistic impression. However, this approach imposes a good choice of the degree of granularity to be able to take the data values into account at the same time as the control values by respecting the semantics and the behavior of the application.

To define the degree of granularity in the Gaspard2 application metamodel, we need to modify the granularity of input and output patterns, and consequently, to modify paving and fitting matrices. This approach can also be seen as a *data oriented* approach since it depends on the input data and just takes the necessary set of data to perform a controllable computation.

For a better understanding of this concept, we consider the example of the Array-OL model represented by figure 30. In the following, we introduce a control module which makes it possible to change the running modes of the elementary task A. Since the mode table is produced by the control module at the beginning of the application, the definition of the production rate of the mode values always depend on the behavior of the studied application. In this section, we consider that the control values come from the external environment of the application, which can be a human user (pressing a button, ...) or a physical process (changing temperature, ...).

One event, one execution mode, one output image

The first case is the simplest case for which we consider that the change of mode refers to the whole output image (figure 35). In this model, the system takes as input an infinity of 4×4 images and a control array, and produces as output an infinity of 4×2 images. In this example, only one control value corresponds to each 4×4 input image. An instance of the repetitive task $1image$ is performed for each input image. It takes as input a pattern of two elements and a control value to produce as output a pattern of one element as explained by figure 36. In this case, the same control value is used for all patterns of the same image. The degree of granularity chosen for this application thus corresponds to the calculation of a complete image.

Several events, several execution modes, one output image

Another possible situation consists in authorizing different running modes for each point of the output image. To do that, we modify the input and output data flow to adapt them to the control flow as shown by figure 37. In this case, each input pattern of two elements corresponds to one control value. The different patterns of the same image can have different control values

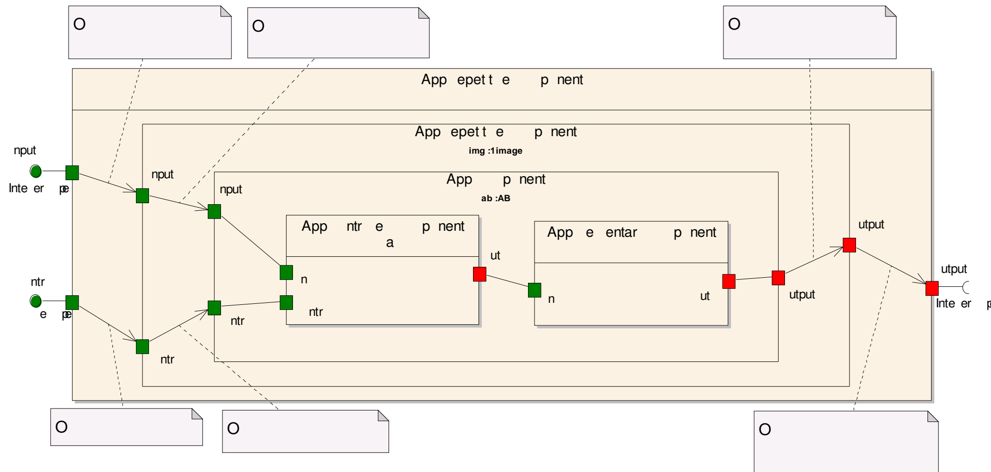


Figure 35: *Example of a control introduction for the whole output image: UML model*

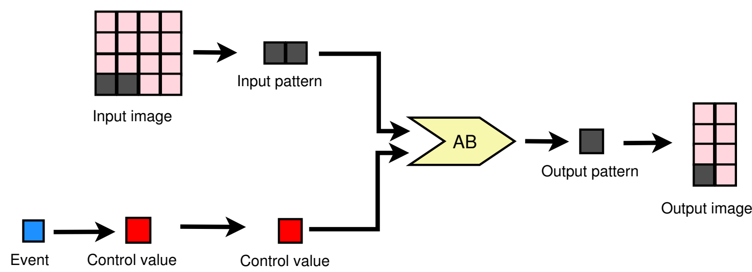


Figure 36: *Example of a control introduction for the whole output image: Global view*

as explained by figure 38. The degree of granularity corresponds to the calculation of only one point of the output image.

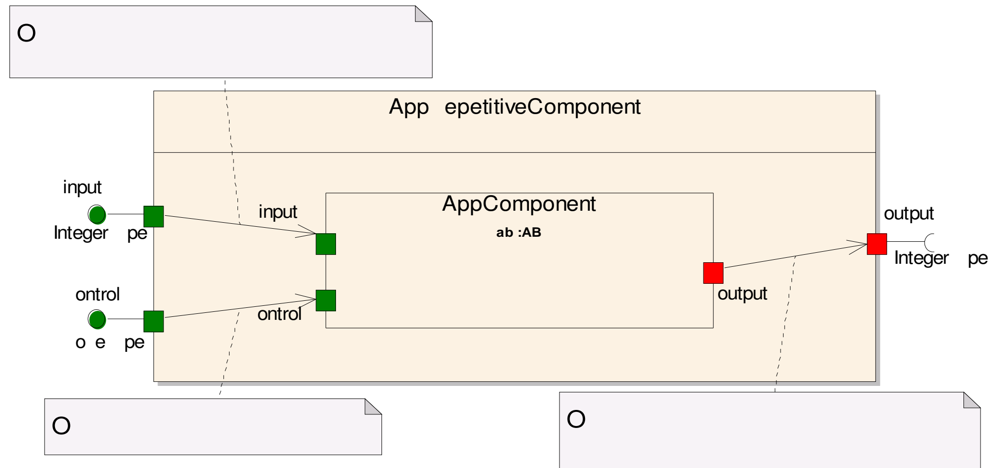


Figure 37: *Example of a control introduction for one point of the output image: UML model*

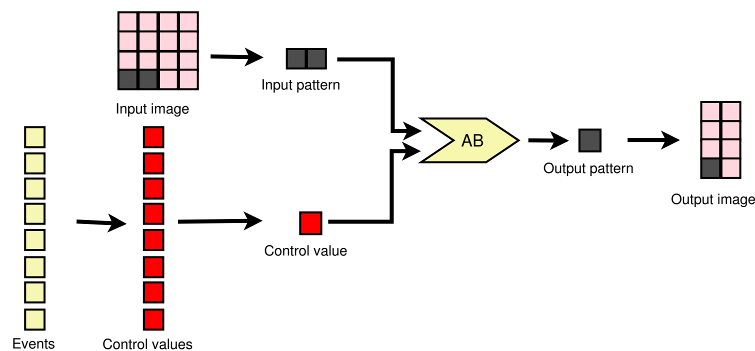


Figure 38: *Example of a control introduction for one point of the output image: Global view*

The main difference between the two presented examples is in the ratio $NbC/NbOP$ when NbC represents the number of control values, and NbP represents the number of output patterns. In the first example (figure 35), to each output image corresponds only one control value which is used for the calculation of the eight output patterns ($NbC/NbOP = 1/8$). While in the second example (figure 37), to each output image corresponds eight control values, one control value by one output pattern ($NbC/NbOP = 8/8 = 1$). Figure 39 shows the control/pattern ratio for the two studied examples.

It is also possible to consider the change of modes for only one line of the output image, several lines, a column, several columns, and so on ...

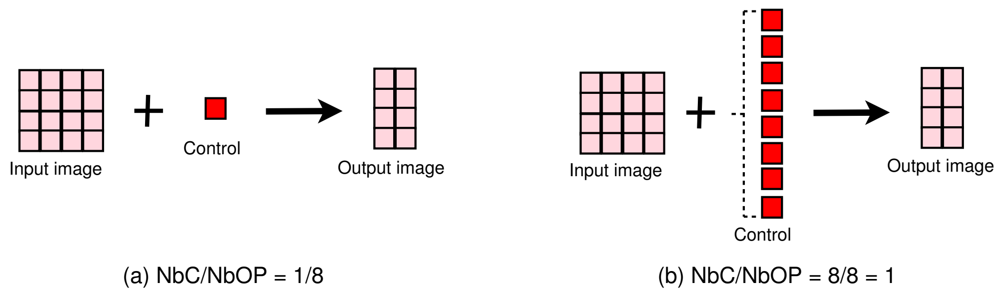


Figure 39: *Representation of the control/pattern ratio*

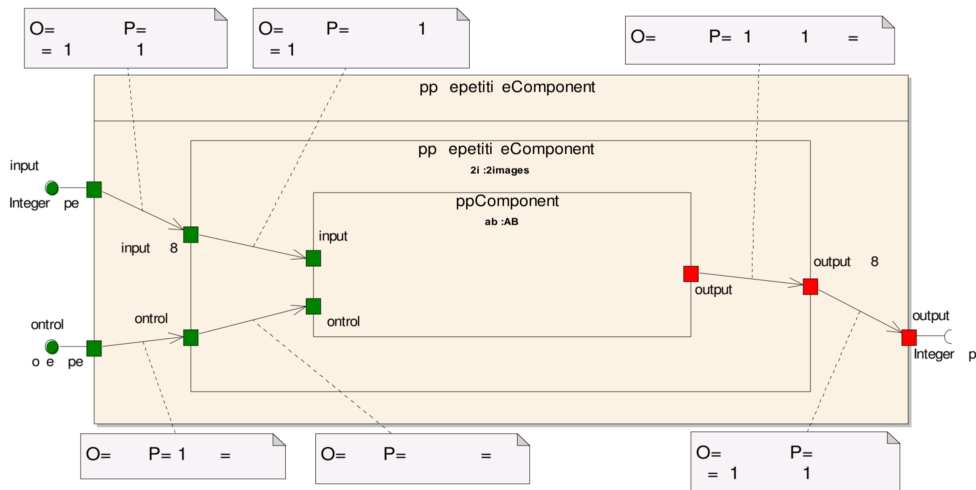


Figure 40: *Example of a parallel execution of two output images using the same control value: UML model*

One event, one execution mode, two output imges

The definition of the degree of granularity for an application can also depend on the implementation or on the mapping of this application on a particular architecture. For example, if we know that our system is able to process two images in parallel, we can consider that the application consumes and produces an infinity of two images as shown by figure 40. In this particular case, the same control value is used for the calculation of two outputs images as explained by figure 41.

One event, several execution modes, two output images

However, it is also possible to have different control values for each point of the output image. In this case, the mode table has a multidimensional structure as shown by figure 42. This example shows that depending on the event value, it is possible to calculate the mode values for each output pattern. This example shows the case of an application in which the input image elements can react differently to the external events. It is then possible to calculate the

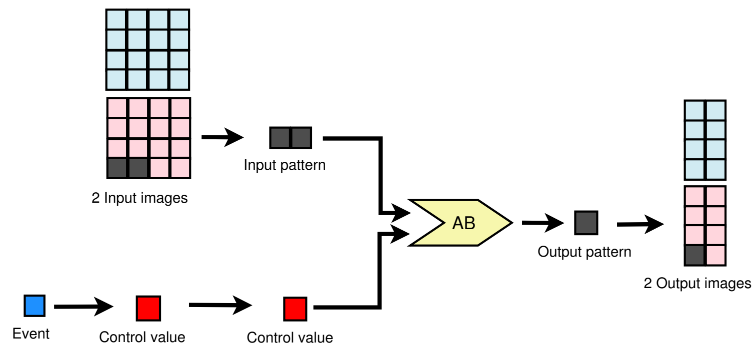


Figure 41: *Example of a parallel execution of two output images using the same control value: Global view*

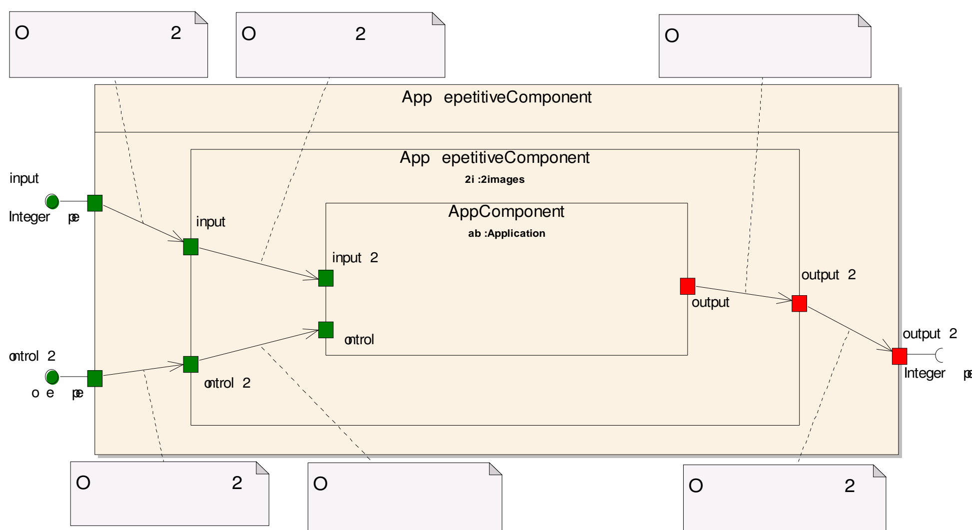


Figure 42: *Example of a parallel execution of two output images using different control values: UMl model*

output image elements in different modes according to the corresponding control value. In this particular case, the calculation of each two output images needs 16 mode values as explained by figure 43. In the studied example, we chose to represent the mode values in an array of 2×4 elements since the structure of the mode array has no particular significance.

The introduction of the degree of granularity concept into a parallel model allows to define the clock signal in which it becomes possible to take into account the various changes of modes in a parallel application. Our approach is a synchronous approach in which the mode tables are regarded as a simple input data, and the choice of the degree of granularity depends on the behavior of the studied application. Using this approach, users have a total freedom to define the degree of granularity for their applications according to the required behavior.

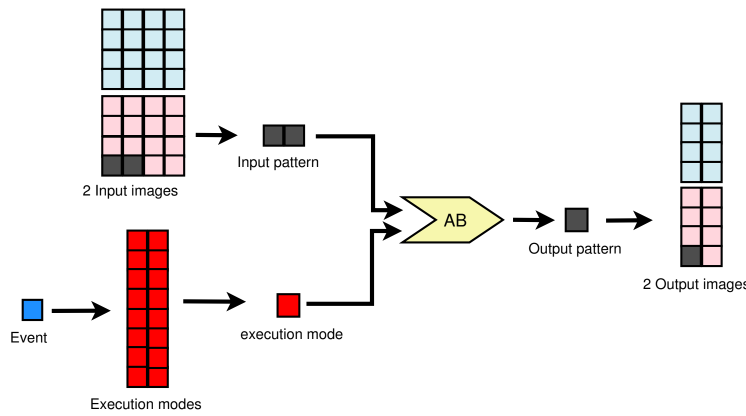


Figure 43: *Example of a parallel execution of two output images using different control values: Global view*

4.2 Internal Control: Changing Modes According to the Computation Results

In the previous examples, we have supposed that the events causing the change of modes come from the external environment and they do not have any relationship with the application. However, it is also possible that these events depend on an internal computation result. To illustrate this concept, we consider the example of figure 30, and we suppose that the elementary task *B* can change its calculation mode according to the result provided by the elementary task *A* as shown by figure 44.

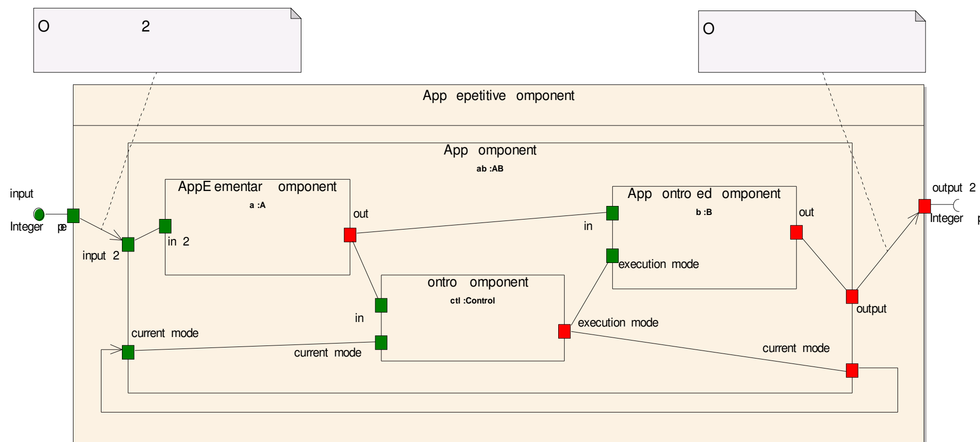


Figure 44: *Example of an internal control: UML model*

In this example, the calculation of the controlled task *B* depends on the result given by the elementary task *A*. To do that, we introduce a control component between the two tasks *A* and *B*. Its functionality consist in providing to the *B* task the adequate mode value according to the result gived by the *A* task. The different points of the output image can be then calculted

in different modes according to the intermediate result gived by the *A* task as explained by figure 45.

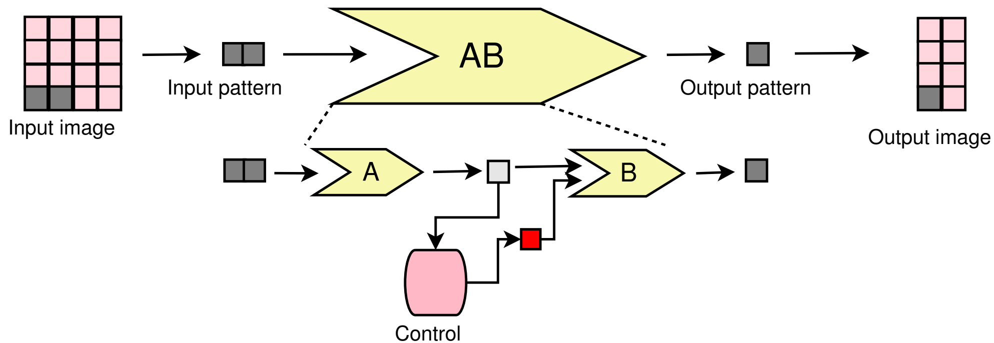


Figure 45: *Example of an internal control: Global view*

The internal control concept can be represented by the introduction of a data dependency between the two tasks *A* and *B* through a control component. For simplification reasons, we call the set containing the control component, the data dependency between the control depending task and the control component, and that between the control component and the controlled task, a *control dependency* (figure 46). This concept implicitly makes it possible to take var-

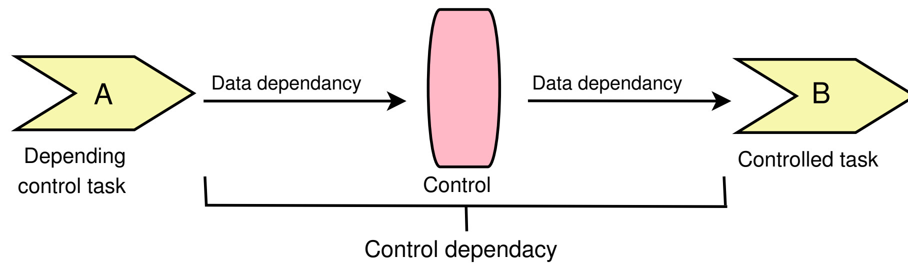


Figure 46: *Control dependency relation*

ious calculations modes for a given task into account by respecting the Gaspard2 application metamodel.

We can also imagine another situation in which, for the calculation of an output image, the application of the controlled task *B* only depends on one or some particular points of the result image provided by the task *A*. Figure 47 gives un example in which, for each output image, the application of the controlled task *B* depends on the origin point of the result image provided by the repeated application of the elementary task *A*. In this example, the different points of each output image are calculated in the same calculation mode according to the value of the origin point of the intermediate image as explained by figure 48. The degrees of granularity for this application is fixed then to the calculation of one output image since the change of mode is only authorized between the output images.

It is also possible to take various control levels into account, internal and external. This concept allows to study more system behavior. So, it is possible to model different behavior

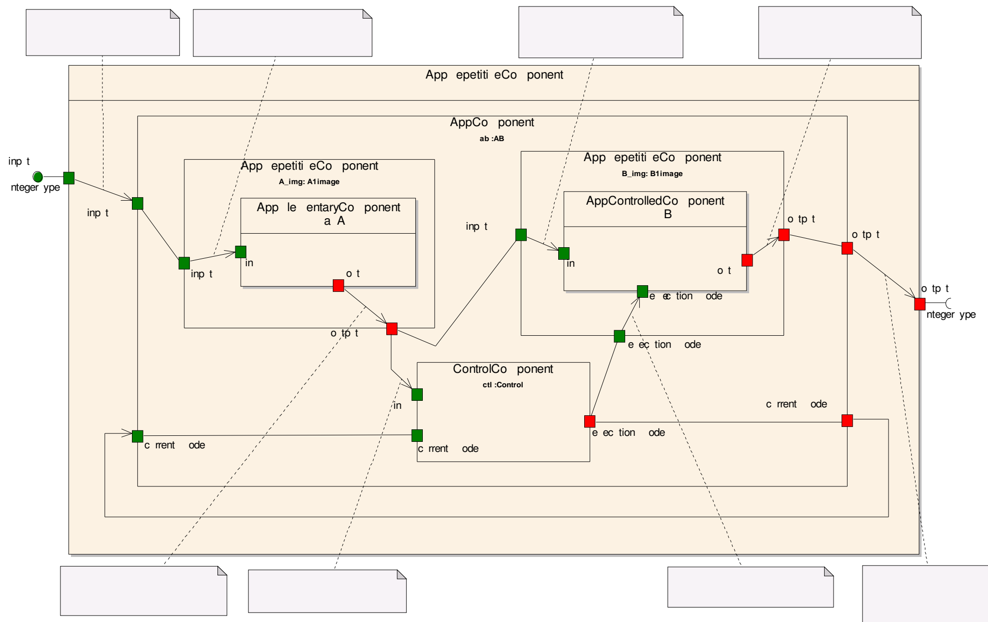


Figure 47: *Example of an internal control according the origin point: UML model*

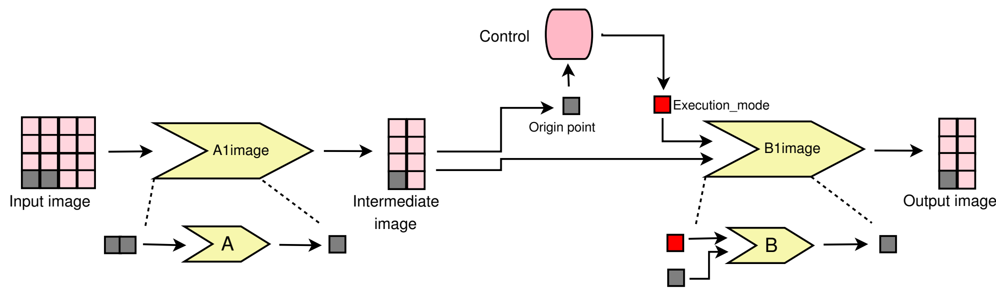


Figure 48: *Example of an internal control according the origin point: Global view*

of a parallel application by choosing the appropriated degrees of granularity and/or introducing the control dependency relation in these applications.

We notice that the introduction of the control into the Gaspard2 application metamodel takes the expected functionality of the application into account to fix the different moments at which it becomes possible to change modes. Since the basic Gaspard2 metamodel does not introduce any time or clock concepts, this approach can be seen as a possible solution for the introduction of the control according to the expected behavior of the application. Basing on these results, we can conclude that the proposed approach is a *functionality oriented* one.

The synchronous assumption, the degrees of granularity concept and the control dependency relation can impose a partial order on the calculation of the different parallel tasks since they make it possible to introduce the *flow* concept in the Gaspard2 application metamodel. However, the proposed model, introducing control in the data parallel applications, respects the

parallelism and the concurrency, it is deterministic, compositional and can be easily introduced in the Gaspard2 application metamodel as shown in section 3.

5 Conclusion and Future Work

In this document, we have studied the introduction of the control concepts in the Gaspard2 application metamodel. Our idea is mainly inspired by the synchronous reactive systems domain and in particular by the concept of Mode-Automata. The proposed metamodel is based on a clear separation between control and data parallel parts. It respects the concurrency, the parallelism, the determinism and the compositionality.

We have shown that the introduction of control into a data parallel domain requires to define the different instances allowing to take the various changes of modes into account. To do that, we have proposed to introduce the notion of degree of granularity in the parallel applications and the control dependency relation between the different instances of the control automata. The studied approach is a synchronous one, which supposes that data and control values must be present to be able to execute a computation function.

The main goal of our work consists in proposing a UML solution for the modeling of control automata, the different running modes of an application and the link between the control and the data parallel parts. Our metamodel allows to study more general parallel systems mixing control and data processing, and gives users more freedom to express the behavior of their applications.

In future work, we will propose the introduction of the control concepts into architecture and association Gaspard2 metamodels allowing to take the configurability concept into account for the architecture models and a better use of the mapping and scheduling algorithms. We will study other scenarios in which the different components in a parallel applications can be controlled by several and different events. This concept makes it possible to define different granularity levels (*multi-degrees of granularity*) in an application which allows the manipulation of different events at different moments. We will also study the relation between the parallel and hierarchical composition of the application model and the parallel and hierarchical automata structure, in particular for verification processes.

We want to transform or compile our control/data parallel metamodel into a synchronous language (like Lustre). This would make it possible to take advantage of the various existing tools for simulation, verification and automatic code generation. Thus, it can be interesting to compare the analysis results and the generated codes obtained using the Gaspard2 environment and those obtained using the corresponding synchronous model.

References

[1] David Harel and Amir Pnueli, *On the development of reactive systems*, In K. R. Apt, editor, Logics and Models of Concurrent Systems, **13**, NATO ASI Series,springer-Verlag, pages 477-498, New York, 1985

[2] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot, *STATEMATE: A working environment for the development of complex reactive systems*, IEEE Transactions on Software Engineering, **16 (4)**, pages 403-414, April, 1990

[3] Nicolas Halbwachs, *Synchronous programming of reactive systems, a tutorial and commented bibliography*, In Tenth International Conference on Computer-Aided Verification, CAV'98, Vancouver (B.C.), LNCS 1427, Springer Verlag, June, 1998

[4] Florence Maraninchi and Yann Rémond, *Mode-automata: About modes and states for reactive systems*, European Symposium On Programming, Springer Verlag, LNCS 1381, Lisbon, Portugal, March, 1998

[5] David Harel, *StateCharts: A visual Formalism for Complex Systems*, Science of Computer Programming, **8**, pages 231-274, 1987

[6] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, *The synchronous data-flow programming language LUSTRE*, Proceedings of the IEEE, **79(9)**, pages 1305-1320, September, 1991

[7] Gerard Berry and Georges Gonthier, *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*, Science of Computer Programming, **19(2)**, pages 87-152, Amsterdam, November, 1992

[8] P. Le Guernic, T. Gautier, M.Le Borgne and C. Le Maire, *Programming Real-Time applications with SIGNAL*, Another Look at Real-Time Programming, Proceedings of the IEEE, **79**, pages 1321-1336, September, 1991

[9] Bruce Powel Douglass, *Real Time UML Third Edition, Advences in the UML for Real-Time Systems*, Addison Weslay, Object Technology Series, ISBN: 0-321-16076-2, February, 2004

[10] Praveen K. Murthy and Edward A. Lee, *Multidimensional synchronous dataflow*, IEEE Transactions on Signal Processing, July, 2002

[11] A. Demeure and Y. Del Gallo, *An array approach for signal processing design*, In Sophia-Antipolis conference on Micro-Electronics, SAME'98, France, October, 1998

[12] Ouassila Labbani, Jean-Luc Dekeyser and Pierre Boulet, *Mode-automata based methodology for Scade*, In Springer, Hybrid Systems: Computation and Control, 8th International Workshop, LNCS series, pages 386-401, Zurich, Switzerland, March 2005

[13] Edward A. Lee, *Overview of the Ptolemy Project*. Technical Memorandum UCB/ERL M01/11, University of California, Berkeley, March, 2001 `http://ptolemy.eecs.berkeley.edu/publications/papers/01/overview`

[14] Irina Madalina Smarandache, *Conception conjointe des applications régulières et irrégulières en utilisant les langages Signal et Alpha*, PhD Thesis, Université de Rennes 1, October, 1998

[15] Julien Soula, *Principe de compilation d'un langage de traitement de signal*, PhD Thesis, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, December, 2001

[16] Philippe Dumont and Pierre Boulet, *Another multidimensional synchronous dataflow: Simulating Array-OL in ptolemy II*, Research Report RR-5516, INRIA, March, 2005

[17] Arnaud Cuccuru, Jean-Luc Dekeyser, Philippe Marquet and Pierre Boulet, *Towards UML 2 extensions for compact modeling of regular complex topologies - A partial answer to the MARTE RFP*, In MoDELS/UML 2005, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, October, 2005

[18] Arnaud Cuccuru, *Modélisation unifiée des aspects répétitifs dans la conception conjointe logicielle/matérielle des systèmes sur puce à haute performances*, PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, September, 2005

[19] N. Halbwachs, *Synchronous programming of reactive systems*, Kluwer Academic Publishers, ISBN: 0792393112, 1993

[20] Robert de Simone and Charles André, *Towards a "Synchronous Reactive" UML subprofile?*, Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems (SVERTS), San Francisco (CA), October, 2003

[21] Alain Demeure, Anne Lafage, Emmanuel Boutillon, Didier Rozzonelli, Jean-Claude, Dufourd and Jean-Louis Marro, *Array-OL : Proposition d'un Formalisme Tableau pour le Traitement de Signal Multi-Dimensionnel*, Gretsi, Juan-Les-Pins, France, September, 1995

[22] M. Jourdan and F. Lagnier and F. Maraninchi and P. Raymond, *A multiparadigm language for reactive systems*. IEEE International Conference on Computer Languages (ICCL), Toulouse, France, 1994

[23] Nicolas Pernet and Yves Sorel, *Optimized Implementation of Distributed Real-Time Embedded Systems Mixing Control and Data Processing*. International Conference: Computer Applications in Industry and Engineering, Las Vegas, USA, November, 2003

[24] Object Management Group, *UML 2.0: Superstructure Draft Adopted Specification*, Inc., Ed., `http://www.omg.org/cgi-bin/doc?ptc/03-07-06`, July, 2003

[25] Esterel Technologies, *SCADE Language Reference Manual*, 2004

[26] P.K. Murthy and E. A. Lee, *A generalization of multidimensional synchronous dataflow to arbitrary sampling lattices*, Rapport de recherche, electronics Research Laboratory, Mars, 1995

[27] Fabrice Popineau, *Introduction à la conception Orientée Objet et à UML*, Supélec, École Supérieur d'Électricité, Septembre, 2000, `http://www.site.uottawa.ca/~bochmann/SEG2501/Notes/UML_Presentation.pdf`

[28] Florence Maraninchi, *Modélisation et validation des systèmes réactifs: un langage synchrone à base d'automates*, Habilitation à Diriger des Recherches, Université Joseph Fourier - Grenoble I, Mai, 1997

[29] Charles Andre, Frederic Boulanger and Alain Girault, *Software Implementation of Synchronous Programs*, International Conference on Application of Concurrency to System Design (ICACSD 2001), Newcastle upon Tyne (UK), in Proceedings of the Second International Conference on Application of Concurrency to System Design, IEEE Computer Society, pp 133-142, June 25-29, 2001

[30] Axel Poigné, Matthew Morley, Olivier Maffeis, Leszek Holenderski and Reinhard Budde, *The Synchronous Approach to Designing Reactive Systems*, Formal Methods Syst. Design, vol. 12, no. 2, pp. 163-187, 1998

[31] L. Zaffalon and P. Breguet, *Conception de systèmes réactifs*, Vision" (Revue scientifique de l'école d'ingénieurs du canton de Vaud (EIVD)), 2001

[32] G. Berry and A. Benveniste, *The synchronous approach to reactive and real-time systems*, Proceedings of the IEEE, 1270-1282, **79**, September, 1991

[33] Nadine Richard, *Description de comportements d'agents autonomes évoluant dans des mondes virtuels*, PhD thesis, École Nationale Supérieure des Télécommunications, Paris, octobre, 2001

[34] Albert Benveniste and Grard Berry, *The synchronous approach to reactive and real-time systems*, Proceedings of the IEEE, 79(9):1270-1282, September, 1991

[35] P. Caspi, D. Pilaud, N. Halbwachs and J. A. Plaice, *Lustre, a declarative language for real time programming*, Proceedings ACM Conference on Principles of Programming Languages, 1987

[36] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, *The synchronous data-flow programming language LUSTRE*, Proceedings of the IEEE, 1305-1320, **79**, September, 1991

[37] Nicolas Halbwachs, Fabienne Lagnier and Christophe Ratel, *Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE*, IEEE Trans, Software Eng, 785-793, 18(9), 1992

[38] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier, *SIGNAL: a Data Flow Oriented Language for Signal Processing*, IEEE Trans. Acoust., Speech, Signal Proc., ASSP-34: 362-374, 1986

[39] Bernard Houssais, *Cours de Programmation en Langage Synchrone SIGNAL*, IRISA, Equipe ESPRESSO, Avril, 200 `http://www.irisa.fr/espresso/Polychrony/doc_V4.15.7/document/cours.pdf`

[40] P. Le Guernic and T. Gautier and M.Le Borgne and C. Le Maire, *Programming Real-Time applications with SIGNAL*, Another Look at Real-Time Programming. Proceedings of the IEEE. September. **79** (1991) 1321–1336

[41] Christophe Lavarenne, Omar Seghrouchni, Yves Sorel, and Michel Sorine, *The SynDEx software environment for real-time distributed systems design and implementation*, In ECC'91, 1991

[42] G. Moreaug, *Modélisation du comportement pour la simulation interactive: application au trafic routier multimodal*, PhD thesis, IFSIC/IRISA, Novembre, 1998

[43] Frédéric Boussinot and Robert De Simone, *The Esterel Language*, Another Look at Real-Time Programming, Proceedings of the IEEE, 1293-1304, **79**, September, 1991

[44] Gerard Berry and Georges Gonthier, *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*, Science of Computer Programming, 87-152, **19**, 1992

[45] Gérard Berry, *The Foundations of Esterel*, Proofs, Languages, and Interaction, Essays in Honour of Robin Milner, MIT Press., 2000

[46] F. Maraninchi and Y. Rémond, *Argos: an Automaton-Based Synchronous Language*, Computer Languages, Elsevier, 61-92, **27**, 2001

[47] C. Andre, *Representation and analysis of reactive behaviors: A synchronous approach*, In CESA'96, IEEE-SMC, Lille, France, July, 1996

[48] David Harel, *STATECHARTS: A visual approach to complex systems*, Science of Computer Programming, 8(3), 1987

[49] H. Boufaied, *Machines d'exécution pour langages synchrones*, PhD thesis, Université de Nice-Sophia Antipolis, Novembre, 1998

[50] Leszek Holenderski and Axel Poigné, *The Multi-Paradigm Synchronous Programming Language LEA*, In Proceedings of the Int. Workshop on Formal Techniques for Hardware and Hardware-like Systems, 1998

[51] M. Jourdan, F. Lagnier, F. Maraninchi and P. Raymond, *A Multiparadigm Language for Reactive Systems*, ICCL, 211-218, 1994

[52] Antoine Rauzy, *Mode automata and their Compilation into Fault Trees*, Reliability Engineering and System Safety, 1-12, **78**, 2002

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