The Knowledge Engineering Review, Vol. 29:4, 433–451. © Cambridge University Press 2014
doi:10.1017/S0269888914000150

Adaptivity in high-performance embedded systems: a reactive control model for reliable and flexible design

HUAFENG YU[1], ABDOULAYE GAMATIÉ[2], ÉRIC RUTTEN[3] and JEAN-LUC DEKEYSER[4]

[1]*INRIA Rennes/IRISA, Campus de Beaulieu, 263, avenue du Général Leclerc, 35042 Rennes, France;*
e-mail: huafeng.yu@us.toyota-itc.com;
[2]*CNRS/LIFL, INRIA Lille Nord Europe, 40 avenue Halley, 59650 Villeneuve d'Ascq, France;*
e-mail: abdoulaye.gamatie@lifl.fr;
[3]*INRIA Rhône-Alples, 655 avenue de l'Europe, Montbonnot, 38334 Saint-Ismier cedex, France;*
e-mail: eric.rutten@inria.fr;
[4]*USTL/LIFL, INRIA Lille Nord Europe, 40 avenue Halley, 59650 Villeneuve d'Ascq, France;*
e-mail: jean-luc.dekeyser@lifl.fr

Abstract

System adaptivity is increasingly demanded in high-performance embedded systems, particularly in multi-media system-on-chip (SoC), owing to growing quality-of-service requirements. This paper presents a reactive control model that has been introduced in Gaspard, our framework dedicated to SoC hardware/software co-design. This model aims at expressing adaptivity as well as reconfigurability in systems performing data-intensive computations. It is generic enough to be used for description in the different parts of an embedded system, for example, specification of how different data-intensive algorithms can be chosen according to some computation modes at the functional level; and expression of how hardware components can be selected via the usage of a library of intellectual properties according to execution performances. The transformation of this model toward synchronous languages is also presented, in order to allow an automatic code generation usable for formal verification, based on techniques such as model checking and controller synthesis, as illustrated in the paper. This work, based on Model-Driven Engineering and the standard UML MARTE profile, has been implemented in Gaspard.

1 Introduction

With the popularization of mobile multimedia devices such as PDA, multimedia cellular phone and MP3 player, high-performance embedded systems (HPESs) attract increasing interests in both industry and academia. These systems feature intensive data processing, including multimedia video codecs, software-based radio and radar signal processing systems. These applications generally require high-performance computing (HPC) resources, where parallel processing is a key feature. In addition, they concentrate on regular data partitioning, distribution and access.

1.1 Design complexity of high-performance system-on-chips

The previously mentioned applications are usually developed by using HPC programming languages that provide useful concepts to deal with parallel processing, in order to meet their real-time requirements. There exist several HPC programming languages and one of the most successful is High Performance Fortran (High Performance Fortran Forum, 1997). However, these languages tend to be very specific to

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

users and lack features, which are increasingly required for the design of modern HPESs: design of complex hardware topologies comprising multi-core architectures, of hardware/software allocations, of intellectual property (IP) deployment enabling reusability, etc. In addition, they do not necessarily allow system verification, which is important for guaranteeing the reliability of the systems.

On the other hand, the *system complexity* issue is another major obstacle faced by system-on-chip (SoC) designers. As computational power increases for SoCs, more functionality are expected to be integrated into these systems. Hence, more complex software applications and hardware architectures are involved. The resulting consequence leads to the disequilibrium in system design, particularly software design does not progress at the same pace as that of hardware. This has become a critical issue and has finally led to the *productivity gap* (Semiconductor Industry Association, 2004).

In order to address the above challenges, many efforts have been recently carried out to better address the SoC design (Sangiovanni-Vincentelli, 2007). Raising the levels of abstraction is considered as an effective solution to reduce the overall complexity in the design. High-level modeling approaches have been developed such as Model-Driven Engineering (MDE) (Object Management Group, 2007a). MDE also enables high-level system design (of both software and hardware) with the possibility of integrating heterogeneous components into the system. Furthermore, *model transformations* enable generation of executable code from high-level models. MDE is also supported by large number of existing standards and tools, for instance, UML-based modeling and transformation tools (UML tool list, 2009).

1.2 Adaptivity in multimedia system-on-chips

System *adaptivity* specification as well as *reconfigurable* computing description are considered to be critical in current embedded systems design, particularly in multimedia SoC design. Such systems must be able to cope with end user requirements and environments. Being adaptive and/or reconfigurable to the environment is highly demanded owing to current flexibility and quality-of-service (QoS) requirements. However, integration of adaptivity and reconfigurability into a system is expected not to affect system performance, especially regarding real-time constraints.

Mode-based control modeling, as one kind of behavioral specifications, plays an important role in multimedia systems, as it satisfies the following requirements that are becoming inevitable in mobile embedded systems: (1) mode changes specified in functionality, for example, color or monochrome mode for video effects; (2) mode changes owing to resource constraints of targeted hardware/platforms, for instance, switching from a high CPU load mode to a smaller one; or (3) mode changes owing to other environmental criteria such as energy consumption constraint. Mode-based control integrates flexibility in the design. Consequently, it offers better QoS choices to designers/end users. Thanks to the above characteristics, mode-based control mechanisms are good candidates for the integration of adaptivity and/or reconfigurability into the high-performance systems design seamlessly. First, modes of data processing can be switched owing to the requirements of users, platforms and environments. Second, data access, as a critical factor of performance, is kept unchanged so that mode changes have little influence on performance.

1.3 Contribution and outline

We present a high-level model allowing for the mixed description of data-intensive and control-oriented behaviors, as a solution for the design of HPESs. This model enables to:

- effectively handle issues such as adequate expression of inherent system parallelism both in functional specification and hardware architecture;
- express control behavior in order to characterize adaptivity and reconfigurability in a system in consideration of performance influence.

The control model is proposed in Gaspard, which is an MDE-based SoC co-design framework dedicated to HPESs (Gamatié *et al.*, 2010), and defined with the UML profile for MARTE (Object Management Group, 2008).

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

A transformation of our model toward synchronous reactive languages is also described, in order to obtain access to their formal validation technologies. It is formalized on the basis of an abstract syntax. Typical applications of our model for system behavior analysis and hardware synthesis are also briefly mentioned applications.

The rest of this paper is organized as follows. Some related works are discussed in Section 2. An overview of Gaspard is provided in Section 3. Section 4 concentrates on the extension of Gaspard with a control model, which includes its abstract syntax, while Section 5 presents the transformation of this control model into synchronous reactive programs. Section 6 presents the implementation of the control in the framework of MDE. The application of this control model in functional specification and IP deployment are briefly described in Section 7. Finally, Section 8 gives the conclusion.

2 Related works

Programming languages for the specification of high-performance systems has already been broadly studied, for example, Message Passing Interface (MPI) (MPI Forum, 2007) and OpenMP (OpenMP API, 2008). More recent languages include Stream It (Thies *et al.*, 2002) and the DARPA high-productivity languages Chapel (Callahan *et al.*, 2004), Fortress (Allen *et al.*, 2007) and X10 (Charles *et al.*, 2005). Improving productivity is one of their main objectives, such as reducing specification complexity by providing suitable macro constructs. Furthermore, they consider specific architecture models. However, they are not well adapted for SoC design in which one needs to program specialized architectures.

Alpha (Wilde, 1994) and Array-OL (Boulet, 2008) (core formalism adopted in Gaspard), which manipulates polyhedra and arrays, respectively, are interesting high-level languages for the data-parallel formalism. Alpha particularly suits for the specification of systolic architectures, while the aim of Gaspard is to cover embedded systems adopting a wide variety of architectures beyond systolic ones.

In several tools and development environments dedicated to dataflow applications, the *multi-paradigm* has been proposed to integrate languages in different styles, that is, dataflow and some imperative features based on finite state machines. This approach benefits from their different expressivity: dataflow specification for numerical computation and state machines for control behavior. Some examples are given below:

- Simulink and Stateflow (The MathWorks, 2009): the former is used for the specification of block diagrams where some operators of Stateflow are used to specify the computation on dataflow. The results of the computation serve to control the system.
- SCADE (Esterel Technologies, 2009): in the SCADE environment, state machines are embedded and used to activate dataflow processing specified in Lustre (Halbwachs *et al.*, 1991). Mode automata (Maraninchi & Rémond, 2003) and polychronous mode automata (Talpin *et al.*, 2006) derive from the concept of *combination of formalisms* in synchronous languages. It extends the dataflow language Lustre and Signal, respectively, with certain imperative style, but without many modifications of language style and structure. Applications specified in SCADE or with mode automata and polychronous mode automata can be formally validated using tools associated with synchronous languages. Although this approach is very similar to our proposition, however, Lustre and Signal are limited in the expression of large parallel data processing.
- Ptolemy: the composition of hierarchical finite state machines with some concurrency models has also been studied in Girault *et al.* (1999). Among them, the composition of state machines with synchronous dataflow is similar to our proposition, and the latter can be considered as a special case applied on parallel data-intensive processing specified with *repetitive structure modeling* (RSM) package of MARTE. The novelty is exhibited in the compositionality of state machines with data-parallelism and data access performance conservation in case of mode changes.

In comparison with the above-mentioned works, our work focuses on embedded systems with HPC, where the way data-parallelism is dealt with is very important. We adopt mode automata where the mode concept helps to reduce performance loss via mode switch during data-intensive computations: the interface of data processing is kept the same even if modes are changed, so data access are not changed allowing one to still benefit from the powerful way of expressing the data-parallelism in Gaspard.

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

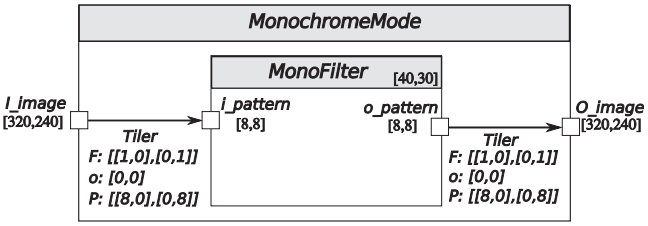


Figure 1 A monochrome effect filter

3 High-performance system design within MARTE

Gaspard (Gamatié *et al.*, 2010) is a MARTE compliant SoC design framework, that enables high-level modeling and automatic code generation by using UML graphical tools and technologies such as Papyrus[1] and Eclipse Modeling Framework[2]. Gaspard features hardware/software co-modeling using the MARTE profile, which enables to model *software applications*, *hardware architectures*, their *allocations* and IP *deployment* separately, but in a unique modeling environment. As high-level Gaspard models contain only domain-specific concepts, model transformations (implemented as Eclipse plugins) enable code generation for different execution platforms, such as synchronous domain for validation and analysis purposes (Gamatié *et al.*, 2008b); or FPGA synthesis (Quadri *et al.*, 2010). Thus, technological concepts are introduced seamlessly in the intermediate and low-level models.

The MARTE RSM is inspired from Gaspard. RSM is based on Array-OL (Boulet, 2007) that describes the *potential parallelism* in a system, and is dedicated to data-parallel multi-dimensional signal processing. Manipulated data in Gaspard and RSM are in the form of multi-dimensional arrays. RSM allows to describe the regularity of a system's structure (composed of repetitions of structural components interconnected in a regular connection pattern) and topology in a compact manner. Gaspard adopts RSM for the specification of regular hardware architectures (such as multi-processor architectures) and parallel processing applications. In addition, both data-parallelism and task parallelism can be, via RSM, expressed with regard to a functional specification.

An example specified with RSM is shown in Figure 1. It expresses data-parallelism in *MonochromeMode*, used for the processing of a [320, 240]-image. Because the filter, called *MonoFilter*, only works on small [8, 8]-pixel subsets, it should be repeated 40×30 times to cover the whole image. In RSM [40, 30] is referred to as the shape of repetition space associated with MonoFilter.

The repeated MonoFilter runs in a repetition context, defined by the MonochromeMode. All repetition instances run in parallel. A connector used in a repetition context is called *LinkTopology*. It adds a set of topological information to a connector. MonoFilter is connected to MonochromeMode via *Tiler* links. The repetitions of MonoFilter consume and produce identically shaped sub-arrays of pixels, which are, respectively, extracted from the input port *I_image* and stored in the output port *O_image*. These sub-arrays, referred to as patterns (shaped [8, 8] in the example), are tiled according to the Tilers, which are associated with each array (i.e. each edge in the graphical representation). A Tiler extracts (resp. stores) tiles from (resp. in) an array based on some information: F a *fitting* matrix (how array elements fill the tiles), $\mathbf{o}$ the *origin* of the *reference tile* (for the *reference repetition*) and P a *paving* matrix (how the tiles cover arrays).

The *repetition space* indicating the number of task instances is itself defined as a multi-dimensional array with a shape. Each dimension of the repetition space can be seen as a parallel loop and its shape space gives the bounds of the nested parallel loops. In Figure 1, the shape of repetition space is [40, 30].

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

$$\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < s_{\text{pattern}}, \mathbf{e_i} = \text{ref} + F \times \mathbf{i} \bmod s_{\text{array}} \tag{1}$$

where s_{pattern} is the shape of the pattern, s_{array} the shape of the array and F the fitting matrix.

[1] www.papyrusuml.org/
[2] www.eclipse.org/emf/

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

For each repetition instance, the reference elements of the input and output tiles are needed to be specified. The reference elements of the reference repetition are given by the *origin* vector, **o**, of each Tiler. The reference elements of the other repetitions are built relatively to this one. As above, their coordinates are built as a linear combination of the vectors of the *paving* matrix as

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

where $s_{\text{repetition}}$ is the shape of the repetition space, P the paving matrix and s_{array} the shape of the array.

Gaspard adopts a component-based approach. *Repetitive components* (MonochromeMode in Figure 1) are used to express data-parallelism in an application. Sets of input and output *patterns* (scope of involved data in array structure) consumed and produced by the repetitions of the interior *part* (a data processing task). Thus, a repetitive component provides the repetition context for its interior task. The repetitions are assumed to be independent and schedulable following any order, generally in parallel. In comparison, a *hierarchical component* contains several *parts*, and it allows to define complex functionality in a modular way and provides a structural aspect of the application. Specifically, task parallelism can be described using such a component. The shape of a pattern is described according to a *Tiler* connector, which specifies the tiling of produced and consumed arrays. An *inter-repetition dependency* (IRD) is used to specify an acyclic dependency among the repetitions of the same component. Particularly, an IRD connector leads to the sequential execution of repetitions. A *default link* provides a default value for repetitions linked with an IRD, on condition that the source of dependency is absent.

4 Definition of the reactive control model

This paper presents a control model for adaptive HPC in the framework of Gaspard: basic concepts are described, and both parallel and hierarchical composition are formally defined as well as synchronous reactive semantics is integrated in order to benefit from existing formal validation and synthesis tools (Gamatié *et al.*, 2008b, 2009). In addition to functional specification, this control model is also adopted in IP deployment for reconfigurable FPGAs, which is detailed in Quadri *et al.* (2010).

4.1 Mode-based control modeling

Control behavior in Gaspard is expressed through a mode-switch-based control model, which derives from mode automata (Maraninchi & Rémond, 2003). The notion of exclusion between modes helps to separate different computations in a modular way. As a result, programs are well structured and fault propagation from one mode to another is reduced. The control model is mainly composed of concepts such as *mode switch* and *state graphs*.

4.1.1 Mode switch and modes

A mode switch contains at least one mode, and offers a switch functionality that chooses one mode to execute, among several alternative present modes (Labbani *et al.*, 2005). The mode switch in Figure 2(a) illustrates such a component having a *window* with multiple tabs and interfaces. For instance, it has an m (mode value input) port as well as several input/output ports such as i_d (data input) and o_d (data output). The modes, M_1,..., M_n, in the mode switch are identified by the mode values: m_1,..., m_n. The switch between the modes M_i is carried out according to the mode value received through the port m. Each mode can be hierarchical or elementary in nature. All modes have the same interface (i.e. i_d and o_d ports).

4.1.2 State graphs

A Gaspard state graph (Figure 2(b)) is similar to Statecharts (Harel, 1987) and its synchronous variant SyncCharts (Andre, 2004) and mode automata (Maraninchi & Rémond, 2003), which are used to model the system behavior using a state-based approach. A Gaspard state graph is a sextuplet (Q, V_i, V_o, T, M, F) where:

- Q is the set of states defined in the state graph;
- V_i, V_o the sets of input and output variables, respectively. V_i and V_o are disjoint, that is, $V_i \cap V_o = \emptyset$. V the set of all variables, that is, $V_i \cup V_o$;

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

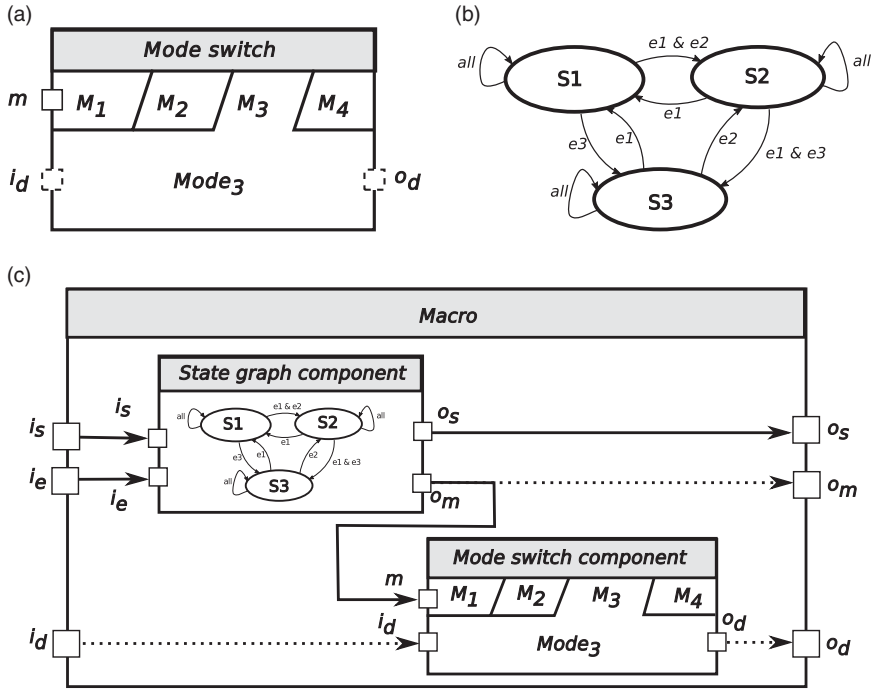


Figure 2 Basic concepts and their composition of Gaspard control model

- $T \subseteq Q \times C(V) \times Q$ the set of transitions, labeled by *conditions* on the variables of V. The conditions C the Boolean expressions on V;
- M the set of modes defined in the state graph;
- $F \subseteq Q \times M$ a set of surjective mappings between Q and M.

Gaspard state graph is considered as a graphical representation of transition functions as discussed in Gamatié *et al.* (2008a), hence there are no initial states defined. Each *state* is associated with some mode value specifications, and *Transitions* are conditioned by some events or Boolean expressions.

4.1.3 Composing mode switch and state graph

A component, whose behavior is exhibited by Gaspard state graphs, is called *state graph component*. A mode switch component indicates a component implementing a mode switch. Then a *macro* component can be used to compose state graph component and mode switch components together. The macro in Figure 2(c) illustrates one possible composition, that is, other compositions are also possible. For instance, one state graph component can be associated with several mode switch components. In the macro, the state graph component produces a mode value (or a set of mode values) and sends it (them) to the mode switch component. The latter switches the modes accordingly.

4.2 Composition definitions

Composition of the control model is detailed in this paper owing to the requirements of expressivity and verifiability. The parallel and hierarchical composition has been formally defined so that complex systems can be specified and system behavior is clear and verifiable by some formal verification tools. With the clear composition definition, it is also possible to abstract the control part of a system in order to make the verification more efficient.

4.2.1 Parallel composition

The parallel composition can be specified in two ways: composing state graphs directly or composing state graph components. The first one is similar to the parallel composition defined in SyncCharts and mode automata. The second one is considered as normal Gaspard component composition.

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

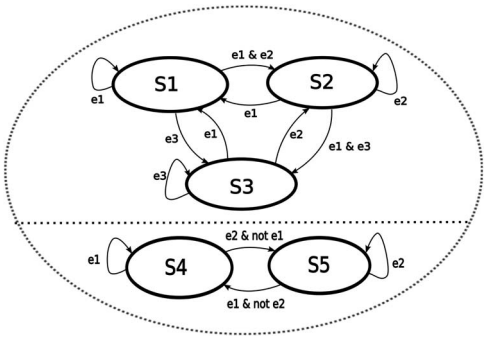


Figure 3 An example of the parallel composition of state graphs

A set of state graphs (Figure 3) can be composed together. The composition of state graphs in this manner is considered as *parallel composition*. Figure 3 illustrates a parallel composition example. The two state graphs are placed in the same ellipse, but separated by a dash line.

Two state graphs $G^1 := (Q^1, V_i^1, V_o^1, T^1, M^1, F^1)$ and $G^2 := (Q^2, V_i^2, V_o^2, T^2, M^2, F^2)$ are considered here to illustrate the composition. The composition operator is noted as: $\|$. The parallel composition is defined as:

$$(Q^1, V_i^1, V_o^1, T^1, M^1, F^1) \parallel (Q^2, V_i^2, V_o^2, T^2, M^2, F^2)$$
$$= ((Q^1 \times Q^2), (V_i^1 \cup V_i^2) \setminus (V_o^1 \cup V_o^2), (V_o^1 \cup V_o^2), T, (M^1 \times M^2), F)$$

where

$$F = \{((q^1, q^2), (m^1, m^2)) \mid (q^1, m^1) \in F^1 \wedge (q^2, m^2) \in F^2\}$$

and

$$T = \{((q^1, q^2), \mathrm{C}, (q^{1\prime}, q^{2\prime})) \mid (q^1, C^1, q^{1\prime}) \in T^1 \wedge (q^2, C^2, q^{2\prime}) \in T^2\}$$

where C is a new expression on C^1 and C^2: $C = C^1$ *and* C^2, and $(V_i^1 \cup V_i^2) \setminus (V_o^1 \cup V_o^2)$ implies any output variable is not considered as an input variable in the composed graphs, hence it should be removed from the input variable set.

These state graphs in a parallel composition can be triggered to carry out transitions at the same time, upon the presence of the source states and events of both two state graphs. Moreover, the number of transitions fired is supposed to be always the same for these state graphs, that is, the inputs of these state graphs have the same size.

As state graph components are considered as normal Gaspard components, the resulting component composed of several state graph components can be also considered as a normal Gaspard component. The composition of state graph components is illustrated with an example in Figure 4. State graph components can also be composed with other Gaspard components. But note that the interfaces of these state graph components should be coherent, that is, it ensures the same transition rate of state graphs. In addition, state graph components can also be composed with standard Gaspard components, which can specify the control out of capacity of state graphs, for example, some binary operations on events or conditions on numbers.

4.2.2 Hierarchical composition

Hierarchical composition of state graphs is defined similar to that of SyncCharts and mode automata. A state in the state graph A can be refined by another state graph B, where B is considered as sub-graph of A. Consider a state graph $G := (Q, V_i, V_o, T, M, F)$ where $Q = \{q_0, q_1, ..., q_n\}$, and a corresponding set of refining automata $\{G^k\}_{k\in[0,n]}$ where $G_k = (Q^k, V_i^k, V_o^k, T^k, M^k, F^k)$. The composition can be defined (inspired from Maraninchi & Rémond, 2003) as:

$$G \triangleright \{G^k\}_{k\in[0,n]} = (Q', V_i', V_o', T', M', F')$$

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

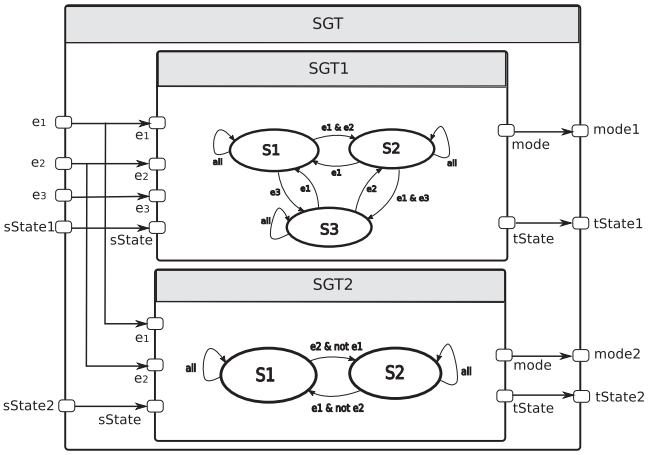


Figure 4 Parallel composition of state graph components in Gaspard

where

$$Q' = Q \triangleright \{Q^k\}_{k\in[0,n]} = \bigcup_{k=0}^{n} \{q^k \triangleright q_j^k \mid j \in [o, n^k]\}$$

$$V_o^1 = V_o \cup \bigcup_{k=0}^{n} V_o^k$$

$$V_i' = (V_i \cup \bigcup_{k=0}^{n} V_i^k) \setminus V_o'$$

$$\begin{aligned} T' =& \{(q^k \triangleright q_{j_1}^k, C, q^{d\prime} \mid (q^k, C, q^d) \in T \wedge j_1 \in [0, n^k]\} \\ & \cup \{(q^k \triangleright q_{j_1}^k, (C \wedge \neg \bigvee_{(q^k, C_m, q^d)\in T} C_m), q^k \triangleright q_{j_2}^k) \mid (q_{j_1}^k, C, q_{j_2}^k) \in T^k \wedge j_2 \in [0, n^k]\} \end{aligned}$$

$$M' = M \cup \bigcup_{k=0}^{n} M^k$$

and

$$F' = \{((q, q^1, \ldots, q^n), (m, m^1, ..., m^n)) \mid (q, m) \in F \wedge (q^1, m^1) \in F^1 \wedge \ \ldots \ \wedge (q^n, m^n) \in F^n\}$$

$q^k \triangleright q_j^k$ denotes the state q^k is refined by the state q_j^k. T' has two kinds of transitions: $\{(q^k \triangleright q_{j_1}^k, C, q^d \mid (q^k, C, q^d) \in T \wedge j_1 \in [0, n^k]\}$ implies the transitions of G, and q^k and q^d denotes the source and target state, respectively. As q^k is refined by states of G_k, $q^k \triangleright q_{j_1}^k$ is used instead of q^k. The target state is also a refined state of q^d, however, which state is entered is decided at run time. Hence, $q^{d'}$ is used instead. $\{(q^k \triangleright q_{j_1}^k, (C \wedge \neg \bigvee_{(q^k, C_m, q^d)\in T} C_m), q^k \triangleright q_{j_2}^k) \mid (q_{j_1}^k, C, q_{j_2}^k) \in T^k \wedge j_2 \in [0, n^k]\}$ denotes the transitions in G^k, that is, these transitions are fired when no transitions of G^k are fired. This condition is expressed by: $\neg \bigvee_{(q^k, C_m, q^d)\in T} C_m$. $q_{j_1}^k$ and $q_{j_2}^k$ denote the source and target states of a transition in G^k. Figure 5 shows an example of the hierarchical composition of state graphs. In this example, the state $S3$ is refined by the state graph composed of states $S4$ and $S5$, denoted by $S3 \triangleright S4$ and $S3 \triangleright S5$.

Hierarchical composition using both state graph components and mode switch components can be also achieved. In this case, a state graph component acts as a mode in a mode switch component, thus, it is only activated in this mode. The mode switch component therefore defines the collaboration between its corresponding state graph component and other state graph components acting as modes, where state graphs of the latter act as state refinements of state graphs of the former. An example is illustrated in Figure 6. The state graph associated with SGT_1 has three states: S_1, S_2 and S_3, which correspond to three modes M_1, M_2 and M_3, respectively. In the MST, SGT_2 is defined as the mode M_3, that is, SGT_2 is activated

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

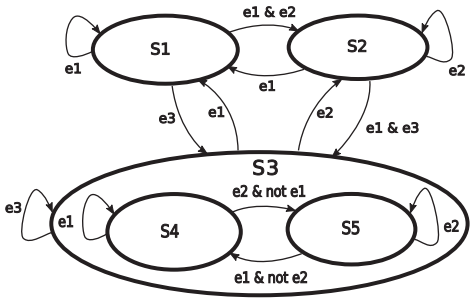


Figure 5 An example of hierarchical composition of state graphs

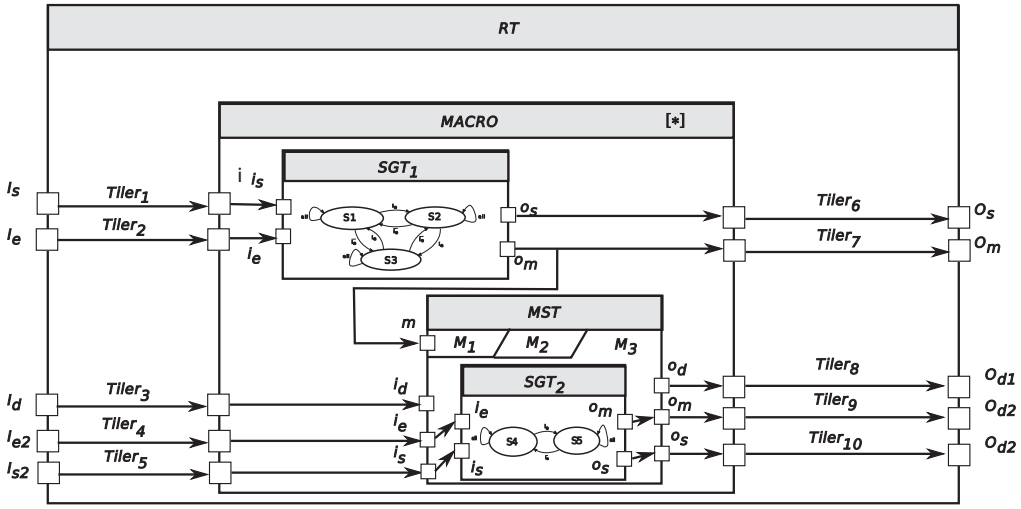


Figure 6 Hierarchical composition of state graph components in Gaspard

only when S_3 is active. Hence, the state graph of SGT_2 is considered as a refinement (sub-state graph) of SGT_1's S_3.

4.3 Reactive semantics

Basic Gaspard control constructs have been presented herein before, but its explicit semantics is still not given. We propose to integrate reactive semantics, inspired from mode automata (Maraninchi & Rémond, 2003), in order to confer certain expected properties related to safe design as well as enhance design correctness verifiability. The integration result of the control model is called Gaspard mode automata. The basic structure of Gaspard mode automata is presented by a composition of state graph component and mode switch component, that is, the *Macro* in Figure 2(c). The state graph component in this macro structure acts as a state-based controller and the mode switch component achieves the mode switch function. Compared with mode automata, where computations are set in the states, the computations in Gaspard mode automata is placed in the mode switch component, that is, outside of the Gaspard mode automata states. In Gaspard mode automata, a state graph component and its associated mode switch components are expected to be specified in the same repetition context in order to force the same executing cadence upon all the components.

There is also the incompatibility between specifications for parallelism in Gaspard and sequential trace semantics of automata. Hence, the parallel model is mapped onto a timed model through the time dimension defined in Gaspard. In addition, additional constructs (*IRD* and *default link*) are used to build a connecting link between the preceding and the following states of state graph components. More precisely, IRD specifications should be specified for the macro structure when it is placed in a repetition context.

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

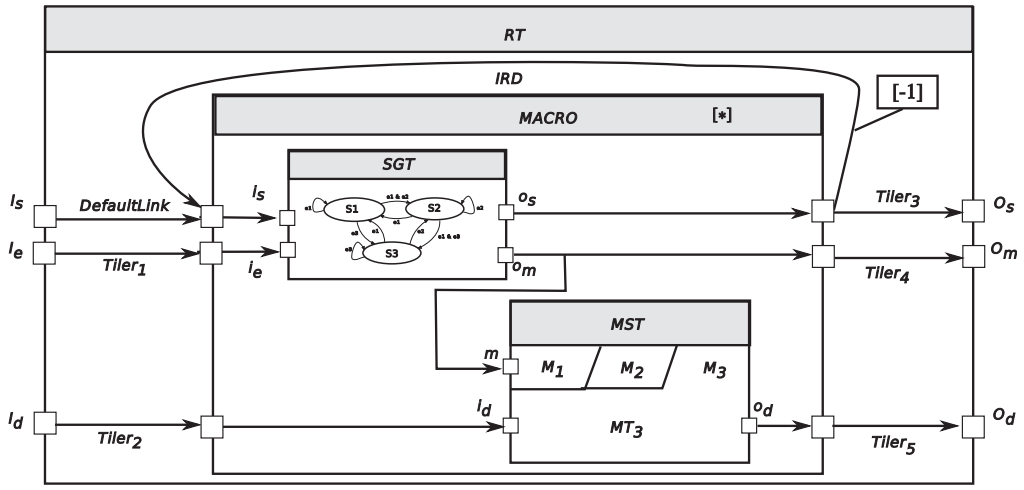


Figure 7 An example of Gaspard mode automata

$$
\begin{array}{lcll}
HTask & ::= & \{Task\};\ \{Connection\};\{Deployment\} & (r1)\\
Task & ::= & task_id\ ;\ Interface\ ;\ Body & (r2)\\
Interface & ::= & i, o\ :\ \{Port\} & (r3)\\
Port & ::= & datatype\ ;\ shape & (r4)\\
Body & ::= & Structure^{e} \mid Structure^{r} \mid Structure^{h} \mid Structure^{mt} \mid Structure^{sg} & (r5)\\
Structure^{e} & ::= & null & (r6)\\
Structure^{r} & ::= & t_i : \{Tiler\}\ ;\ (\mathbf{r}, Task)\ ;\ t_o : \{Tiler\} & \\
 & & \mid t_i : \{Tiler\}\ ;\ (\mathbf{r}, Task)\ ;\ t_o : \{Tiler\}\ ;\ \{IRD\} & (r7)\\
Tiler & ::= & Connection\ ;\ (F; \mathbf{o}; P) & (r8)\\
Structure^{h} & ::= & HTask & (r9)\\
Connection & ::= & p_i,\ p_o\ :\ Port & (r10)\\
Structure^{mt} & ::= & \{(m_k, T_k) : (mode_id, Task)\}, \forall i \neq j & \\
 & & \Rightarrow T_i.Interface = T_j.Interface & \\
 & & \& Task.Interface = T_k.Interface \cup p_m & (r11)\\
Structure^{sg} & ::= & sg_id\ ;\ S\ ;\ Tr\ ;\ s_c\ ;\ m_o\ ;\ reset & (r12)\\
S & ::= & \{(state_id, Mode) \mid (state_id; Structure^{sg}; Mode)\} & (r13)\\
Tr & ::= & \{(state_id; label; state_id)\} & (r14)\\
IRD & ::= & Connection\ ;\ dep_vector; default & (r15)\\
Deployment & ::= & \{ip_id\ ;\ task_id\ ;\ depl_info\} & (r16)
\end{array}
$$

Figure 8 An abstract syntax extract of Gaspard concepts for functionality specification

The reasons are as follows: the macro structure represents only one transition from source state on target state, whereas a Gaspard mode automata should have continuous transitions. Hence, the macro should be repeated to enable multiple transitions. Thus, the Gaspard mode automata can be built and executable.

4.4 A Gaspard mode automaton example

Figure 7 shows a Gaspard mode automata example, which can be transformed into synchronous mode automata eventually. *MACRO* is placed in a repetitive context, where each of its repetitions models one transition of mode automata. An *IRD* links the repetitions of *MACRO* and conveys the current state (sends the target state of one repetition as the source state to the next repetition) between these repetitions. The states and transitions of the automata are encapsulated in the *SGT*. The data computations inside the mode are set in the modes. *SGT* and its modes share the same repetition space, so they run at the same rate or clock. The detailed formal semantics of Gaspard mode automata can be found in Gamatié *et al.* (2008a).

4.5 Abstract syntax of the control model

Models and their transformations sometimes are big in size and have divers technologies or domains involved, which leads to difficult understanding, maintenance and verification (Chen *et al.*, 2005; Combemale *et al.*, 2006; Mohagheghi & Dehlen, 2008). The extended control model and its transformation is first described using abstract syntax (Figure 8), as abstract syntax helps to concentrate on

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

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

Figure 9 A simple example of Lustre code

domain-specific concepts, models, and their transformations are therefore simplified and easy to be verified, particularly the coherence between the models involved in the transformation.

A Gaspard *HTask* (rule (*r*1) in Figure 8, where {} denotes a set; we call a Gaspard component as a task from the viewpoint of programming language here) consists of a set of *Tasks*, *Deployment* and *Connections* (*r*10), which connect the tasks. These tasks share common features (*r*2): a *task_id*, an *interface* (*r*3) and a *Body* (*r*5). *Interface* specifies input/output *Ports* (typed by i or o in rule (*r*3) and *Port* is defined in rule (*r*4)) from which each task receives and sends multi-dimensional arrays. *Task* has many types, including elementary, repetitive, hierarchical, mode switch and state graph task. The type of a *Task* is identified by the structure in the *Body* of the task. These task types are described as follows:

- An *elementary task* (*r*6) corresponds to an atomic computation block. Typically, it represents a function or an IP. Elementary task can be associated with an IP through *Deployment*.
- A *repetitive task* (*r*7) expresses data-parallelism in a task. The attribute **r** (in the rule (*r*7)) denotes the *repetition space* for repetitions. In addition, patterns involved in each task repetition are defined via *Tilers* (*r*8). *IRD* can be also specified in a repetitive task (*r*15), which describes the dependency between repetitions of the repetitive task. *dep_vector* specifies which repetition relies on which repetitions and *default* gives a default value.
- A *hierarchical task* (*r*9) is actually an HTask. It is represented by a hierarchical acyclic task graph, in which each node consists of a task, and edges are labeled by arrays exchanged between task nodes.
- A *mode switch task* (*r*11) achieves mode switch function as presented in Section 4.1. It is composed of tasks as mode, which have the same interface. The interface of the mode switch task is the union of the interface of its internal tasks and the mode port, that is, p_m.
- A *state graph task* (*r*12) is associated with Gaspard state graphs that provide mode values for corresponding mode switch tasks. It is composed of a set of *states* (*r*13) and *transitions* (*r*14). A *reset* flag indicates reset of current state s_c for a state graph. m_o denotes the output mode.

Deployment (*r*16) indicates how to find and integrate an implementation, considered as an IP, of a specific elementary task. Each elementary task is associated with an IP, and *depl_info* describes necessary information for the integration of the IP into the system.

5 Transformation into the synchronous model

In Gaspard, the proposed control has been applied on functional specification, IP deployment, etc. The corresponding model transformations are also under development. Here, the transformation from Gaspard control integrated into functional specifications to synchronous languages (Benveniste *et al.*, 2003) is illustrated. The latter is used for the formal validation of Gaspard models. The transformation is described with the help of the abstract syntax of both the Gaspard model and the synchronous equational model.

A code segment of the synchronous dataflow language Lustre (Figure 9) is used for the introduction of some basic concepts here. A node is a basic functionality unit in Lustre. Each node gives the same results with the same inputs because of its determinism. Nodes have modular declarations that enable their reuse. Each node has an interface (input at line (l1) and output at (l2)), local definition (l3), and equations (lines (l5) and (l6)). Variables are called signals in Lustre. Equations are signal assignments. In these equations,

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

$$
\begin{array}{lcll}
Module & ::= & \{Node\} & (s1)\\
Node & ::= & nodename\ ;\ Interface\ ;\ EqSystem & (s2)\\
Interface & ::= & Interface^{i}\ ;\ Interface^{o} & (s3)\\
Interface^{i} & ::= & \{SignalDeclaration\} & (s4)\\
Interface^{o} & ::= & \{SignalDeclaration\} & (s5)\\
SignalDeclaration & ::= & signal\ ;\ DataType & (s6)\\
DataType & ::= & type\ ;\ shape & (s7)\\
EqSystem & ::= & \{Equation\} \mid CaseEquations \mid Autmata \mid extnodelink & (s8)\\
Equation & ::= & EqLeft\ ;\ EqRight & (s9)\\
EqLeft & ::= & null \mid signal & (s10)\\
EqRight & ::= & signal \mid SignalDelay \mid Invocation & (s11)\\
Invocation & ::= & nodename\ ;\ \{signal\} & (s12)\\
CaseEquations & ::= & case\ ;\ \{(modevalue, Equation)\} & (s13)\\
SignalDelay & ::= & signal\ ;\ delayinstant\ ;\ \{defaultvalue\} & (s14)\\
Automata & ::= & aut_id\ ;\ S\ ;\ Tr\ ;\ s_i\ ;\ reset & (s15)\\
S & ::= & \{(state_id \mid (state_id\ ;\ Automata\ )\} & (s16)\\
Tr & ::= & \{(state_id\ ;\ label\ ;\ state_id)\} & (s17)
\end{array}
$$

Figure 10 An abstract syntax extract of basic synchronous concepts

there are possibly node invocations (l5) that are declared outside this node. Obviously, in Lustre, modularity and hierarchy are inbuilt. The composition of these equations, denoted by ";", means their parallel execution w.r.t. data dependencies. The node has the same meaning independently of the equation order.

5.1 Synchronous equations abstract syntax

This abstract syntax is constructed based on common aspects of synchronous languages, which is intended to model three synchronous dataflow languages such as Lustre, Signal and Lucid synchrone. The syntax is illustrated in Figure 10.

Only a brief description of synchronous languages' syntax is given here. A more detailed description can be found in Yu (2008). A *Node* is defined as a basic functionality. All the nodes in an application are declared in a module ($s1$). A node ($s2$) is composed of *Interface* and *EqSystem*. The *Interface* has two families: $Interface^{i}$ and $Interface^{o}$ ($s3$). An *EqSystem* is the body of a node and defines the function of the node. An *EqSystem* ($s8$) can have at least one *Equation*, *CaseEquation*, *Automata* or an *extnodelink*, which indicate four implementation types of the *EqSystem*. An *Equation* is either an assignment of a *signal*, *SignalDelay* or an *Invocation* of another node ($s9$, $s10$ and $s11$). *Signal* and *SignalDelay* are variables used in the program. An *Invocation* indicates a function call to another node defined in the module. A *CaseEquations* contains a case statement where equations are activated according to some condition such as mode values ($s13$). A *SignalDelay* is similar to the *pre* operator (Benveniste *et al.*, 2003), which takes the value of the signal at the previous *delayinstant* instant ($s14$). *delayinstant* is defined as a positive integer. A *defaultvalue* is a default value of a signal when no value is provided for the previous instant. *Automata* ($s15$) have an S (set of states, $s16$), Tr (set of transitions, $s17$), an initial state s_i and a *reset* flag. The *reset* implies a restart of the automata, that is, the initial state s_i is taken as entering state. Finally, an *extnodelink* indicates an external implementation of the node, that is, a node is implemented by other languages.

5.2 Transformation between the two models

The first step of transformation is structural, and the second step involves semantic aspects. The correspondence in the transformation between Gaspard and synchronous concepts is indicated by $\overset{T}{\Rightarrow}$. In order to distinguish the concepts of Gaspard and the synchronous model, the number of the rule, in which the concept appears in the syntax, is also given in parentheses following the concepts. The rule numbers of Gaspard concepts begin with r, and those of synchronous model start with s.

5.2.1 Structural transformation

The synchronous *Module* ($s1$) is the container of all nodes. A Gaspard *HTask* ($r1$) is first translated into a *Node* ($s2$): $HTask \overset{T}{\Rightarrow} Node$. A *Task* ($r2$) is also translated into a *Node*: $Task \overset{T}{\Rightarrow} Node$. An *Interface* ($r3$) can be translated into *Interfaces* ($s3$): $Interface \overset{T}{\Rightarrow} Interface$. A *Port* ($r4$) in an *Interface* is a connection point of a Gaspard task, it is translated into a *signal*: $Port \overset{T}{\Rightarrow} signal$.

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

A *Body* (*r*5) represents the internal structure of a task. It can be considered as an *Eqsystem* (*s*8): $Body \overset{\mathcal{T}}{\Rightarrow} Eqsystem$. Five kinds of structures are involved in a body. A $Structure^{e}$ (*r*6) represents the structure of an elementary task, and the deployment will be used to integrate an IP for this task. A $Structure^{r}$ (*r*7) is translated into a set of *equations* (*s*9) (Yu *et al.*, 2008b). A $Structure^{h}$ (*r*8) is a structure that has a compound task, that is, *HTask*. A $Structure^{mt}$ (*r*11) corresponds to *CaseEquations* (*s*13): $Structure^{mt} \overset{\mathcal{T}}{\Rightarrow} CaseEquations$. A $Structure^{sg}$ (*r*12) can be translated into either *CaseEquations* (*s*13) or *Automata* (*s*15): $Structure^{sg} \overset{\mathcal{T}}{\Rightarrow} CaseEquations$ or $Structure^{sg} \overset{\mathcal{T}}{\Rightarrow} Automata$. Finally, *Deployment* (*r*10) is translated into an *extnodelink* (*s*8), which indicates the integration of an external implementation: $Deployment \overset{\mathcal{T}}{\Rightarrow} extnodelink$.

5.2.2 Transformation of behavioral aspects

In addition to structural aspects, transformation also involves behavioral aspects of Gaspard specifications. These aspects include the transformation of parallelism and control. The former, which includes elementary, repetitive and hierarchical task transformation, has been presented in Yu *et al.* (2008b). The latter, including mode switch and state graph tasks, will be presented here.

IRD (*r*15), which enforces serialized execution and passes values between repetitions, is translated by *SignalDelay* (*s*14): $IRD \overset{\mathcal{T}}{\Rightarrow} SignalDelay$. A *SignalDelay* is used to convey the previous value of a signal. *Depvector* is translated by *delayinstant*: $depvector \overset{\mathcal{T}}{\Rightarrow} delayinstant$. These two concepts are used to indicate which previous value to take in Gaspard and synchronous program.

A $Structure^{mt}$ (*r*11) corresponds to *CaseEquations* (*s*13). Each pair of (m_k, T_k), that is, a mode, is translated into (*modevalue, Equation*) in a case statement. The *modevalue* is the condition and *Equation* is invoked when the corresponding condition is evaluated as true. The *Equation* represents the computation to carry out in a mode.

A $Structure^{sg}$ (*r*12) is translated into either *CaseEquations* (*s*13) or *Automata* (*s*15) according to target languages. The former translate state graphs into pure equations with case statements, while the latter is a direct transformation which keeps explicit structure of states and transitions.

6 Implementation within Model-Driven Engineering

This section concentrates on the implementation of previously mentioned control modeling and transformation in the MDE framework, based on conceptual descriptions of models and their transformation.

6.1 Gaspard control metamodel

UML (Object Management Group, 2007b) is adopted in Gaspard to specify state graphs, which is actually a subset of UML State machines. It also helps to ensure the compatibility between Gaspard control with the MARTE profile. An extract of the Gaspard control metamodel is illustrated in Figure 11. This metamodel is proposed as a subset of the UML State machines metamodel in structure so that the compatibility between them simplifies the following transformation.

6.2 Model transformation

Based on the previously presented transformation on the foundation of abstract syntax, model transformation rules are built on OMG Query/Views/Transformations (Object Management Group, 2005) such as MOMOTE tool (MOdel to MOdel Transformation Engine) (INRIA DaRT Team, 2009). Most part of the transformation from Gaspard to synchronous languages has been developed as Eclipse plugins (Yu *et al.*, 2008b). An extension is expected to cover transformations to other platforms. Here, only an example is given in Figure 12 to illustrate the resulting automata (Lustre mode automata) obtained from Figure 2(b). Equations of data-parallel processing are not included in this example as Lustre mode automata is only used for model checking purpose.

7 Some applications

Gaspard is well fitted for applications with repetitive data-parallel computations, which include image processing, multimedia video codecs, software-based radio and radar signal processing, etc. A modern

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

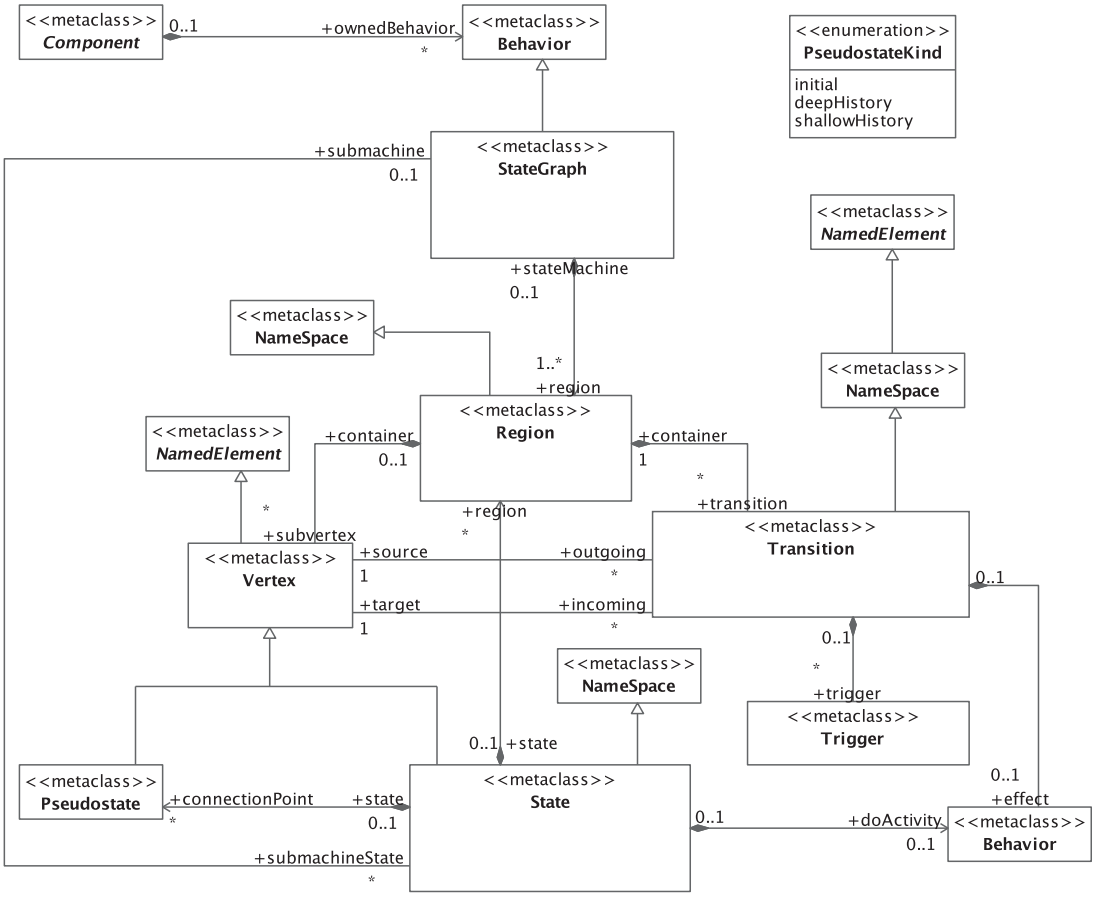


Figure 11 An extract of Gaspard *state graph*, which is proposed according to the metamodel of UML state machines

```
AUTOMATON auto
    STATES
       s1    init      [ state = s1; mode = M1; ]
       s2              [ state = s2; mode = M2; ]
       s3              [ state = s2; mode = M3; ]
    TRANS
       FROM s1 TO s2 WITH  [ e1 = true and e2 = true]
       FROM s2 TO s1 WITH  [ e1 = true ]
       FROM s1 TO s3 WITH  [ e3 = true ]
       FROM s3 TO s1 WITH  [ e1 = true ]
       FROM s2 TO s3 WITH  [ e1 = true and e3 = true]
       FROM s3 TO s2 WITH  [ e2 = true ]
    PROCESS auto [in(e1, e2, e3), out(state, mode)]
```

Figure 12 An extract of a Lustre mode automaton obtained by transformation

cellular phone is then taken as a typical example in this paper, which have complex multimedia functionalities: camera, games, MP3 music, video. Let us focus on the video part. A global model view of this multimedia module is illustrated in Figure 13. The played video clips are obtained from different *VideoSources*, either online library or local storage. There are different display *ImageStyle* such as *Black & White*, *Negative*, *Sepia* or *Normal*, meaning no effect. In addition, the *Resolution* of the video can be set to *High*, *Medium* and *Low*. Finally, the color can be in either *Color* or *Monochrome* for *ColorEffect* options. *ImageStyle*, *Resolution* and *ColorEffect* include both control and data processing parts.

Besides user commands, the video display modes are controlled by the system by the *Controller*, which validates mode change requests from other components according to current mode configuration and the

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

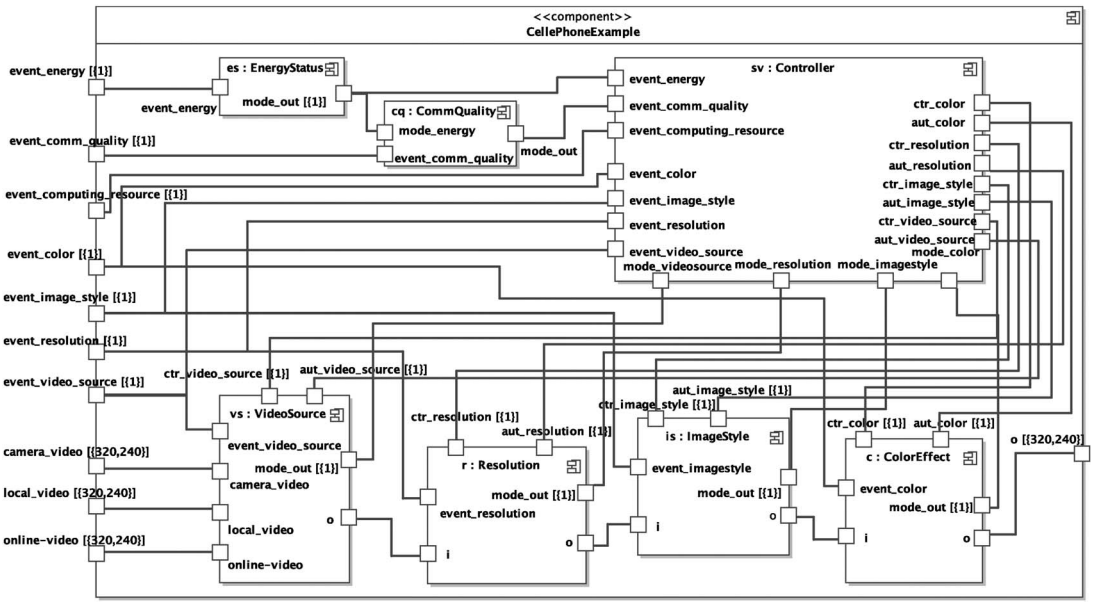


Figure 13 A global model view of the multimedia functionality module

available computational resources. This specific component is either coded manually, or generated automatically by *discrete controller synthesis* according to QoS requirements, including the status of energy level (*EnergyStatus*) and the communication quality (*CommQuality*). The former indicates the energy level according to events received from an energy monitor component, and the latter provides the communication quality level according to the energy level and the online transmission bandwidth of received data.

The above module has different configuration modes following which its components achieve algorithms for a suitable image display. Depending on the resource status, for example, energy level, communication bandwidth, the display quality of an online video varies. Hence, each mode is associated with non-functional properties, which must be satisfied in order to display images at a good quality level. The modes defined in the components are characterized by quantitative attributes representing the following non-functional properties: energy consumption (E), communication quality required (CQ), computing resource consumption (CR) and memory consumption (M). These non-functional requirements are considered as instantaneous consumption of quantitative resources that may vary from one system reaction to another. The values associated locally with the modes are combined additionally when components are composed in parallel, so as to obtain global costs for the whole system from the local costs of its components.

Possible behaviors involving the above characteristics are, for example, that the consumption of a resource must respect the bounds defined by its capacity. Therefore, if a new functionality is executed, then the other tasks that are already running should switch to lower consumption modes, possibly reducing their quality as well. Or, if the level of the battery goes down, then the control should switch task modes so that the lower energy capacity is respected. Such control strategies are defined by properties expressed in terms of the states and inputs of the system. The Sigali tool (Marchand *et al.*, 2000) allows one to express Boolean properties on states and inputs ($P : \mathbb{B}^n \rightarrow \mathbb{B}$), and to build cost functions, associating numerical values (here, assumed to be integer, without loss of generality) with Boolean functions of states and inputs ($f : \mathbb{B}^n \rightarrow \mathbb{N}$). We essentially consider *invariance*, by specifying a subset of system states, defined by a Boolean property P: P is invariant for the system if for all states in this subset, transitions from these states lead to states in the same subset. This invariance property of the system is noted $\forall P$: the Boolean property P is true at every instant of every trace of the system.

7.1 Model checking

Design validation through model checking of functional properties was first studied (Yu *et al.*, 2008a). One of the examples is an exclusion relation between two modes from two different components.

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

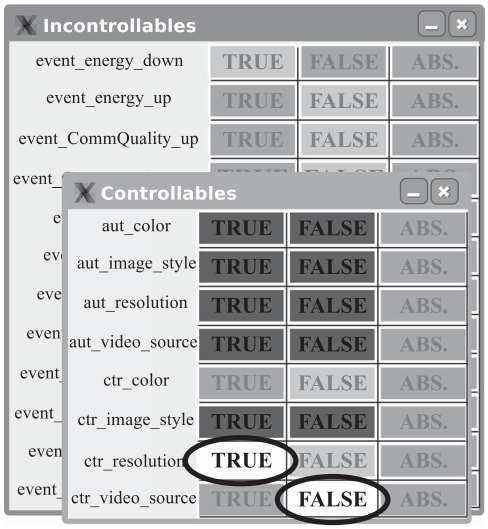


Figure 14 Simulation of the controlled system

For instance, in order to avoid waste of resources, it can be useful to specify that the modes B&W (*ImageStyle* component) and Color (*ColorEffect* component) are never active at the same instant. This invariance property is denoted: $\forall(\overline{\text{B \& W} \wedge \text{Color}})$. Non-functional properties, for instance, resources provided by the environment, platform or hardware, can also be checked. By associating each state of the automata with cost functions, a global cost function has been computed when composing these automata in parallel. This global function defines, for each resource, the sum of its cost functions computed from all possible combination of active states in the global automaton. A resource bound is then specified in the system. Finally, the reachability of certain states is checked under this configuration of resource costs. This example shows a fast verification in consideration of functional and/or non-functional properties in order to to evaluate complex systems at a low cost in a fast manner, compared with other approaches that need to specify the system in a precise way (Yu *et al.*, 2008a).

7.2 Discrete controller synthesis

Another example involves discrete controller synthesis (Gamatié *et al.*, 2009): for a cost function where the global cost is defined by the sum of the local costs of the components, for example, for memory footprint, there is a bound defined by the size of the memory. Thus, if we note by $f_M = f_{IS} + f_{VS} + f_R + f_{CE}$ the cost function associating with each global state of the system, the memory usage in this state, we can enforce the fact that this usage will always be bounded by the memory available (here, 90 units), by the invariance synthesis objective: $\forall(f_M \leq 90)$. The bound itself can vary in time: it is actually the case for the available energy. In this case, we add to our model an automaton that represents the environment, namely here the energy resource available. We associate a cost function f_{EA} with this automaton, associating with its states the energy quantity instantaneously available. Then, we can bound the energy consumption f_E by the available energy: $\forall(f_E \leq f_{EA})$.

With these synthesis operations, different policies or strategies can be obtained *automatically* by changing the objectives, hence providing for separation of concerns and making the models easy to reuse.

The controller computed by Sigali is extracted, and co-simulated with the system with the SigalSimu tool. Figure 14 shows a particular simulation step, where the controller enforces the values of two controllable inputs so as to keep the properties satisfied. At this step, the system is in high energy, high resolution and color state. We then simulate the discharge of the battery by the occurrence of the uncontrollable input `event_energy_down`. On the controllable inputs panel, the clearer inputs shown with ellipsis are those whose values have been forced by the controller. It is here the case of the input `ctr_resolution`, meaning that the controller has triggered the transition from high to medium resolution state.

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

7.3 Hardware synthesis

The control model has also been applied to move from high-level MARTE specifications to reconfigurable architectures such as FPGAs, and specifically those supporting partial dynamic reconfiguration. The continuation of this work is in progress in the project FAMOUS, in which the OMG MARTE profile will be extended into RecoMARTE, for the design of reconfigurable architectures, implemented on FPGAs. In addition, slight extensions have been made to the existing control/dataflow concepts and the deployment level in our framework to integrate the partial dynamic reconfiguration aspects. An initial version of generating a complete IP core from model transformations has also been developed, along with a solution to avoid de-synchronization related to task parallelism in the modeled applications (Quadri *et al.*, 2010). Currently, the Xilinx-based partial dynamic reconfiguration design flow is adopted owing to its availability and extensible nature.

8 Conclusions

This paper presents a reactive control model and its transformation in a MDE-based SoC co-design framework for high-performance systems, which is compliant with the MARTE standard. The control model is based on mode automata in order to enable the specification of adaptivity for high-performance systems. Our contribution is the integration of composition and formal semantics into this model to enrich its expressivity and verifiability. Model transformation toward synchronous languages has also been studied in order to benefit from validation tools associated with these languages. Furthermore, the extended control is also integrated into Gaspard IP deployment for reconfigurable FPGAs, besides functional specifications. The model is illustrated with its abstract syntax, based on which a conceptual transformation is given. An implementation of the metamodel and model transformation in Eclipse has been carried out, and it is partially accomplished.

One perspective of this work is the application of the control model toward other target technologies such as SystemC. Reconfigurability is also a very interesting research topic in these platforms. Our control proposition is one of possible solutions, which take HPC into account. Another perspective is related to the implementation of the control model, its transformation and formal verification in the unique framework of MDE. The backstage technologies should be made transparent to users and the only interface to users would be an integrated development and simulation environment, such as Eclipse.

Acknowledgements

The authors acknowledge all the reviewers and editors of this paper for their valuable advices and comments.

References

Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessn, J.-W., Ryu, S., Steele, G. L. Jr. & Tobin-Hochstadt, S. 2007. The Fortress language specification version 1.0 beta. Technical report. SunMicrosystems, Inc., March.

Andre, C. 2004. Computing SyncCharts reactions. *Electronic Notes in Theoretical Computer Science* **88**, 3–19.

Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Guernic, P. L. & Simone, R. D. 2003. The synchronous languages twelve years later. *Proceedings of the IEEE* **91**(1):64–83.

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

Boulet, P. 2008. Formal semantics of Array-OL, a domain specific language for intensive multidimensional signal processing. Research Report RR-6467, INRIA, March.

Callahan, D., Chamberlain, B. L. & Zima, H. P. 2004. The cascade high productivity language. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*. IEEE Computer Society, April, 52–60.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. & Sarkar, V. 2005. X10: an object-oriented approach to nonuniform cluster computing. In *20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*. ACM Press, 519–538.

Chen, K., Sztipanovits, J., Abdelwahed, S. & Jackson, E. K. 2005. Semantic anchoring with model transformations. In *European Conference on Model Driven Architecture Foundations and Applications (ECMDA–FA'05)*, 115–129.

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

Combemale, B., Rougemaille, S., Crégut, X., Migeon, F., Pantel, M., Maurel, C. & Coulette, B. 2006. Towards rigorous metamodeling. In *MDEIS*, 5–14.

Esterel Technologies 2009. SCADE. http://www.esterel-technologies.com

Gamatié, A., Beux, S. L., Piel, É., Atitallah, R. B., Etien, A., Marquet, P. & Dekeyser, J.-L. 2010. A model driven design framework for massively parallel embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)* (to appear).

Gamatié, A., Rutten, É. & Yu, H. 2008a. A model for the mixed-design of data-intensive and control-oriented embedded systems. Research Report RR-6589, INRIA, July. http://hal.inria.fr/inria-00293909/fr

Gamatié, A., Rutten, É., Yu, H., Boulet, P. & Dekeyser, J.-L. 2008b. Synchronous modeling and analysis of data intensive applications. *EURASIP Journal on Embedded Systems*. http://dx.doi. org/10.1155/2008/561863

Gamatié, A., Yu, H., Delaval, G. & Rutten, E. 2009. A case study on controller synthesis for data-intensive embedded systems. In *Second International Conference on Embedded Software and Systems (ICESS09)*, ISBN: 978-0-7695-3678-1, IEEE Computer Society, 75–82.

Girault, A., Lee, B. & Lee, E. 1999. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **18**(6):742–760.

Halbwachs, N., Caspi, P., Raymond, P. & Pilaud, D. 1991. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE* **79**(9):1305–1320.

Harel, D. 1987. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* **8**(3):231–274.

High Performance Fortran Forum 1997. High Performance Fortran language specification, January. http://hpff. rice.edu/versions/hpf2/index.htm

INRIA DaRT Team 2009. Gaspard SoC framework. http://www.gaspard2.org/

Labbani, O., Dekeyser, J.-L., Boulet, P. & Rutten, É. 2005. Introducing control in the Gaspard2 data-parallel MetaModel: synchronous approach. In *Proceedings of the International Workshop MARTES*.

Maraninchi, F. & Rémond, Y. 2003. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming* **46**(3):219–254.

Marchand, H., Bournai, P., Borgne, M. L. & Guernic, P. L. 2000. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System: Theory and Applications* **10**(4):325–346.

Mohagheghi, P. & Dehlen, V. 2008. Where is the proof? A review of experiences from applying MDE in industry. In *Fourth European Conference on Model Driven Architecture Foundations and Applications (ECMDA-FA)*, Schieferdecker, I. & Hartman, A. (eds), LNCS **5095** 432–443. Springer.

MPI Forum 2007. Message Passing Interface forum. http://www.mpi-forum.org/docs/docs.html

Object Management Group 2005. MOF query/views/transformations, November. http://www.omg.org/spec/QVT/

Object Management Group 2007a. Portal of the Model Driven Engineering community. http://www.planetmde.org

Object Management Group 2007b. OMG unified modeling language (OMG UML), superstructure, V2.1.2, November. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/

Object Management Group 2008. Modeling and analysis of real-time and embedded systems (MARTE). http://www.omgmarte.org/

OpenMP API 2008. OpenMP 3.0 specifications. http://www.openmp.org/mp-documents/spec30.pdf

Quadri, I., Yu, H., Gamatié, A., Rutten, E., Meftali, S. & Dekeyser, J.-L. 2010. Targeting reconfigurable FPGA based SoCs using the MARTE UML profile: from high abstraction levels to code generation. *International Journal of Embedded Systems (IJES)*, Special Issue on Reconfigurable and Multicore Embedded Systems (to appear).

Sangiovanni-Vincentelli, A. 2007. Quo Vadis SLD: reasoning about trends and challenges of system-level design. *Proceedings of the IEEE*, **95**(3), 467–506. http://chess.eecs. berkeley.edu/pubs/263.html

Semiconductor Industry Association 2004. International technology roadmap for semiconductors update (design). http://www.itrs.net

Talpin, J.-P., Brunette, C., Gautier, T. & Gamatié, A. 2006. Polychronous mode automata. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*. ISBN: 1-59593-542-883–92. ACM.

The MathWorks 2009. Simulink. http://www.mathworks.com/products/simulink

Thies, W., Karczmarek, M. & Amarasinghe, S. 2002. Streamit: a language for streaming applications. In *Compiler Construction. 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002*, April, Lecture Notes in Computer Science, **2304/2002**, 49–84. Springer.

UML tool list 2009. Unified modeling language (UML) tools. http://en.wikipedia.org/wiki/List_of_UML_tools

Wilde, D. K. 1994. The ALPHA language. Technical Report 827, IRISA.

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press

Yu, H. 2008. *A MARTE-Based Reactive Model for Data-Parallel Intensive Processing: Transformation Toward the Synchronous Model*. PhD thesis, Université des Sciences et Technologie de Lille.

Yu, H., Gamatié, A., Rutten, E. & Dekeyser, J.-L. 2008a. Safe design of high-performance embedded systems in a MDE framework. *Innovations in Systems and Software Engineering (ISSE)* **4**(3):215–222.

Yu, H., Gamatié, A., Rutten, E. & Dekeyser, J.-L. 2008b. Model transformations from a data parallel formalism towards synchronous languages. In *Embedded Systems Specification and Design Languages, Selected Contributions from FDL'07*, chapter 13, ISBN: 978-1-4020-8296-2, Lecture Notes Electrical Engineering, **10**. Springer Verlag.

https://doi.org/10.1017/S0269888914000150 Published online by Cambridge University Press