2010 Ninth International Symposium on Parallel and Distributed Computing

Operational Semantics of the Marte Repetitive Structure Modeling Concepts for Data-Parallel Applications Design

Abdoulaye Gamatié
CNRS/LIFL, INRIA LNE
Villeneuve d'Ascq, France
abdoulaye.gamatie@lifl.fr

Vlad Rusu
INRIA Rennes - Bretagne Atlantique
Rennes, France
vlad.rusu@inria.fr

Éric Rutten
INRIA Grenoble - Rhône Alpes
Saint Ismier, France
eric.rutten@inria.fr

***Abstract*—This paper presents an operational semantics of the repetitive model of computation, which is the basis for the repetitive structure modeling (RSM) package defined in the standard UML Marte profile. It also deals with the semantics of an RSM extension for control-oriented design. The goal of this semantics is to serve as a formal support for *i)* reasoning about the behavioral properties of models specified in Marte with RSM, and *ii)* defining correct-by-construction model transformations for the production of executable code in a model-driven engineering framework.**

***Keywords*-Marte Repetitive Structure Modeling, data-parallelism, operational semantics, correctness**

I. Introduction

Modern embedded systems are increasingly complex and sophisticated. An example of domain concerned by this observation is consumer electronics, which includes cellular phones, audio equipments, high-definition televisions, digital cameras, etc. The designers of these systems must particularly face several challenges about the way to overcome the design complexity and to produce high-quality products, while reducing the overall development cost and time-to-market. Model-based design approaches have been strongly promoted as useful solutions to address these challenges. They propose the necessary notions and concepts enabling one to capture the relevant information about systems, according to the problem to address. Since models are often executable and verifiable, they are an interesting costless support for both behavioral simulation and property analysis before the production of the actual system.

A. High-level modeling

The general purpose modeling language UML is very popular in both industry and academia thanks to its attractive graphical representation. However, because of its generality, UML has to be often refined via the notion of *profile* in order to make it precise enough to address domain-specific problems. Roughly speaking, a profile is a set of stereotypes that specialize UML concepts. There are currently several profiles such as SysML [1] for system design in general, or UML SPT [2], UML-RT [3] and Embedded UML [4] for embedded system design in particular. Among these profiles, only the former two have been standardized by the Object Management Group (OMG). SysML is a general-purpose modeling language for system design, while UML SPT is dedicated to the modeling of time, schedulability, and performance-related aspects of real-time systems. The UML-RT profile targets real-time systems, but is less rich in terms of concepts than UML SPT. The Embedded UML profile has been defined as an experimental proposal that goes beyond the real-time field by including concepts for hardware/software co-design.

Because all above profiles dedicated to embedded and real-time systems may potentially overlap, significant standardization efforts have been recently realized by the OMG, resulting in the single unified and effective Marte standard profile [5]. Marte stands for Modeling and Analysis of Real-time and Embedded systems. It is an evolution of the UML SPT profile and borrows some concepts from the more general SysML profile. It is composed of several packages for application and hardware architecture design as well as their mapping, for non functional properties specification, etc. It also includes a specific package named *Repetitive Structure Modeling* or RSM, which is used to describe repetitive computations and topologies (e.g., data-parallel algorithms, grid of processing units) in a system.

B. Contribution

In this paper, we mainly consider RSM, for which we propose an operational semantics (Section III) through the repetitive model of computation [6], which served as a basis for the definition of this package. Our proposition aims at providing a formal semantics for this subset of Marte in order to specify the meaning of each basic concept and to enable the verification and execution of models specified with these concepts. This will be exploited to define correct-by-construction model transformations in our model-driven engineering framework, called Gaspard [7], dedicated to data-intensive applications.

Operational semantic descriptions are not usually taken into account in the definition of UML profiles. This raises several serious correctness issues about the manipulation of models defined with profiles. First, when multiple profiles

978-0-7695-4120-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ISPDC.2010.30

are combined to model a system, the operational semantics of each profile is clearly needed in order to exactly characterize how the resulting "heterogeneous" model works. Second, when transforming a system model, one also needs to know its precise semantics so as to be able to unambiguously check the preservation of its behavioral property in the transformation results, e.g. in executable programs. This is highly required to define correct-by-construction transformations. Surprisingly, there are only a few works that deal with the definition of operational semantics for UML-related models. A major initiative from OMG aims at studying an operational semantics of a foundational subset for executable UML models (FUML) [8]. In connection to this initiative, authors in [9] also propose to encapsulate operational semantics into UML stereotype definitions.

Outline. In the rest of the paper, Section II gives an informal overview of the concepts defined in the repetitive structure modeling package. Then, Section III defines a formal operational semantics for these concepts in order to be able to reason about the behavioral properties of models defined with RSM. In addition, the semantics of a control-based extension of the RSM is given in Section IV. Finally, Section V provides the concluding remarks.

II. Repetitive structure modeling

The RSM package of Marte allows to describe regular parallelism in a system: functional algorithms, hardware architectures topologies, allocation of functionality on execution platforms. A major RSM concepts is that they enable a factorized and compact representation of potentially large regular structures. This is very helpful for the design scalability, typically when considering massively parallel systems. An overview of the RSM package is shown in Figure 1. All its basic stereotypes inherit from the LinkTopology stereotype, which generalizes the nature of the different kinds of links in a regular structure. These links are Tiler, InterRepetition, DefaultLink and Reshape. We describe the Tiler link and also briefly present the rest of the package.

A. Tiler

A Tiler connector expresses how a multidimensional structure of elements, e.g. an array, is tiled by subsets of elements. We refer to such subsets as *patterns*. In Figure 2, a repetitive task R in which a task T describing some algorithm is repeated in a regular manner. Each instance of T takes as input a pattern p_i extracted from a multidimensional array of data i, and produces a pattern p_o stored in another multidimensional array of data o. The vector $\mathbf{s}_\mathrm{r}$ denotes the (typically, multi-dimensional) *repetition space*, that is, the number of instances of the task T executed in R.

The tiler connectors t_i and t_o contain the useful information that enable to extract and store the patterns respectively from i and in o: F is a *fitting* matrix describing how array elements fill patterns; $\mathbf{o}$ is an *origin* of the *reference pattern*;

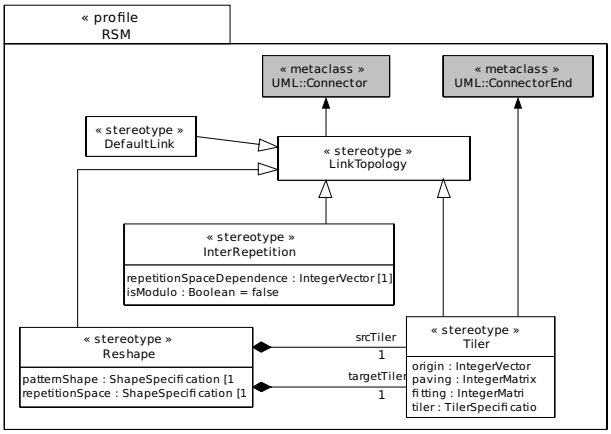

Figure 1. The RSM package of Marte.

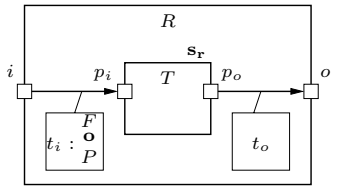

Figure 2. Model of task repetition with tilers.

and P is a *paving* matrix specifying how patterns tile an array. We briefly recall below the basic principles for pattern fitting and array paving (for further details, see also [6]).

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

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

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

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

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

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

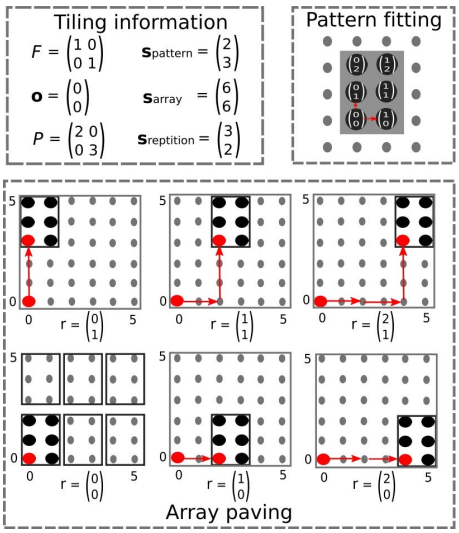

Figure 3. Paving and fitting scenarios.

The above formulas (1) and (2) that respectively define the fitting and paving operations can be combined into one general formula expressing the contents of the patterns $\pi_{\mathbf{k}}$:

$$\pi_{\mathbf{k}} = \{(\mathbf{o} + (P\ F) \times \binom{\mathbf{k}}{\mathbf{i}}) \bmod \mathbf{s}_{\text{array}} | \forall \mathbf{k}, \mathbf{0} \leq \mathbf{k} < \mathbf{s}_{\text{repetition}}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}\} \quad (3)$$

We denote by $\alpha = \biguplus_t \{\pi_{\mathbf{k}} \mid \mathbf{0} \leq \mathbf{k} < \mathbf{s}_{\text{repetition}}\}$ the tiling operation of the array α according to tiler t and repetition space $\mathbf{s}_{\text{repetition}}$, by which the array α is partitioned into a set of patterns according to the general formula (3).

B. Inter-repetition dependency and default link

An InterRepetition link specifies dependencies between the repetitions of a given repeated structural element. This link connects a pattern of a repeated structural element with another pattern of the same repeated structural element.

An example of situation where task instances may depend on other task instances is the computation of the sum of array elements by considering the partial sum previously calculated at each step, until all elements are taken into account. Such an algorithm therefore induces an execution order between task instances. Figure 4 illustrates a repetitive task with an inter-repetition dependency connecting the output ports p_o of executing instances to the input ports p of other instances to execute later. The correspondence between these instances in specified via a repetitionSpaceDependence (see Figure 1) attribute, represented by the dependency vector $\mathbf{d}$ in Figure 4. Finally, a DefaultLink connector specifies the initial value, denoted in Figure 4 by def, of the input port p involved in the inter-repetition dependency.

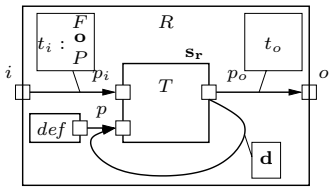

Figure 4. Inter-repetition dependency.

C. Reshape

A Reshape connection, illustrated in Figure 5, expresses link topologies in which elements with shape $\mathbf{s}_{\text{p}}$, from a multidimensional array o_1, are redistributed in another multidimensional array i_2 according to a repetition space characterized by $\mathbf{s}_{\text{r}}$. Actually, a Reshape can be expressed as combination of two Tilers and one repetitive task (cf. Definition 8 of the operational semantics of Reshape).

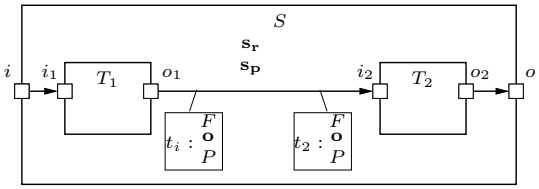

Figure 5. Reshape connection.

The above concepts of the Marte RSM package originate from the repetitive model of computation (MoC) for which we propose an operational semantics in the next section.

III. An operational semantics for RSM

The repetitive MoC relies on the Array-OL domain-specific language [10], which is dedicated to the specification of intensive multidimensional signal processing applications. It extends this language with further constructs that make it expressive enough to design high-performance embedded systems [7]. Among the basic characteristics of Array-OL are the following: *true data dependency expressions, determinism, absence of dependency cycles* and *single assignment* in specifications. In this section, we first give a syntactic overview of the repetitive MoC. Then, we define an associated formal semantics in terms of sequences of computational steps.

A. An overview of the concepts

The main data type manipulated in RSM is *multidimensional array*. Different kinds of tasks are distinguished: *elementary, composed, hierarchical, repetitive* and *reshape* tasks. The grammar presented in Figure 6 describes the basic specification concepts of RSM. By convention, the notation

$x : X$ in the grammar means that X is the type of x, and $\{X\}$ denotes a set of elements typed X.

$$
\begin{array}{rcll}
Task & ::= & Interface; Body & (4)\\
Interface & ::= & i, o : \{Port\} & (5)\\
Port & ::= & id; type; shape & (6)\\
Body & ::= & Body^e \mid Body^c \mid Body^h \mid Body^r \mid Body^s & (7)\\
Body^e & ::= & some\ function\ \phi & (8)\\
Body^c & ::= & Task_1; Task_2; \{Cnx\} & (9)\\
Cnx & ::= & i, o : Port & (10)\\
Body^h & ::= & Task; \{Cnx\} & (11)\\
Body^r & ::= & t_i, t_o : \{Tiler\}; \mathbf{s};\ Task; \{Ird\} & (12)\\
Tiler & ::= & Cnx; (F; \mathbf{o}\ ; P) & (13)\\
Ird & ::= & Cnx; \mathbf{d} & (14)\\
Body^s & ::= & Task_1; Task_2; Reshape & (15)\\
Reshape & ::= & t_1, t_2 : Tiler; \mathbf{s}; \mathbf{p} & (16)
\end{array}
$$

Figure 6. A grammar of RSM concepts.

All tasks share common features, as stated by Rule (4):

- an *interface* defined in Rule (5), which specifies input and output *ports*, respectively represented by i and o. Ports are characterized in Rule (6) by their *identifier*, by the *type* of the array elements they transmit, and by the *shape* (i.e. dimension) of those arrays.
- a *body*, defined in Rule (7), describing the function defined by the task.

In order to explain the remaining rules, we introduce some preliminary definitions used to define the operational semantics of RSM. Here, the main data structure is that of multi-dimensional arrays over basic types (integers, Booleans, ...). Let $\mathcal{V}$ denote the set of such arrays.

Definition 1 (Environment): For a set of ports P, an environment for P is a function $\varepsilon : P \rightarrow \mathcal{V}$. □

The set of environments associated with P is noted ε_P. The fact that a port (or a set of ports) p takes a value v in the environment ε is denoted by $\varepsilon(p) = v$.

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

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

□

For syntactical convenience, we use a "dot" notation to designate parts of a concept according to the grammar of Figure 6, e.g., if T is a task, then $T.Interface$ denotes the set of input and output ports of T.

A task T_2 is a *sub-task* of a task T_1 if either $T_1 = T_2$, or T_2 is a sub-task of a task occurring in the body of T_1 (cf. Rules (9) – (15) in our grammar). We say that two tasks T_1, T_2 are *disjoint* if the set of sub-tasks of T_1 is disjoint from the set of sub-tasks of T_2, or, equivalently, if the sets of the respective *elementary sub-tasks* of T_1, T_2 are disjoint.

As a general assumption, we require that for all disjoint tasks T_1 and T_2, or s.t. T_2 is a *strict* sub-task of T_1, the interfaces $T_1.Interface$ and $T_2.Interface$ are disjoint, hence, all environments $\varepsilon_1 \in \varepsilon_{T_1.Interface}$ and $\varepsilon_2 \in \varepsilon_{T_2.Interface}$ are compatible. Other such "semantical" constraints, not expressed in the grammar shown in Figure 6, are associated to some of the semantical rules defined below.

Our semantics consists of rules of the form

$$\frac{C}{\varepsilon \xrightarrow{T} \varepsilon'}$$

where $T \in \mathcal{T}$, $\varepsilon, \varepsilon' \in \varepsilon_{T.Interface}$, and C is a condition on T, ε and ε'. The environment ε gives the *current* values to the ports of the task T, and, the environment ε' gives the *next* values for these ports, i.e., after the task T is executed. The condition C must be satisfied in order to perform the transition between ε and ε' according to T. We sometimes denote by $[\![T]\!]$ the semantics of a task T as follows: let $T.Interface = (i, o)$; then, for all environments $\varepsilon, \varepsilon'$: $\varepsilon \xrightarrow{T} \varepsilon'$ iff $(\varepsilon'(o) = [\![T]\!](\varepsilon(i)))$.

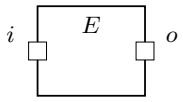

Figure 7. Elementary task.

B. Elementary task

An elementary task E - Rule (8) - consists of a body $E.Body$ performing some function ϕ and an interface $E.Interface = (i, o)$.

Definition 3 (Elementary task): Let E be an elementary task. Its operational semantics is given by the rule:

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

where $(i, o) = E.Interface$. □

C. Composed task (task parallelism)

A composed task K - Rule (9) - has a body that consists in two tasks T_1, T_2, and a set of connections C, where each connection is a pair of ports, as specified by Rule (10).

We assume that the *interface* of the composed task K is equal to $T_1.Interface \cup T_2.Interface$. This ensures that the graph induced by the connections C is acyclic. A simple inductive argument shows that by iterating the composition operator, the resulting graph of connection remains acyclic. This is consistent with the *absence of dependency cycles*, which is required from Marte RSM specifications.

Definition 4 (Composed task): The semantics of a composed task K, whose body is $T_1; T_2; C$, is as follows:

$$\frac{\varepsilon_1 \xrightarrow{T_1} \varepsilon_1', \quad \varepsilon_2 \xrightarrow{T_2} \varepsilon_2' \quad \forall (p_1, p_2) \in C, \varepsilon_1'(p_1) = \varepsilon_2(p_2)}{\varepsilon_1 \oplus \varepsilon_2 \xrightarrow{K} \varepsilon_1' \oplus \varepsilon_2'}$$

□

A natural question that arises is whether the composition is associative, in the following sense. Let T_1, T_2, T_3 be three tasks. Their composition can be obtained by first composing T_1 and T_2 and then composing the result with T_3, or by first composing T_2 and T_3 and then composing T_1 with the result. Are the tasks semantically the same?

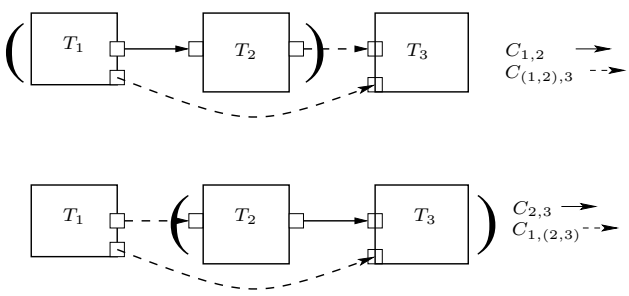

Figure 8. Associativity of composition.

More precisely, let $C_{1,2} \subseteq T_1.Interface \times T_2.Interface$ be a connection between T_1 and T_2, let $T_{1,2}$ be the composition of T_1 and T_2 with respect to the connection $C_{1,2}$, and let $C_{(1,2),3} \subseteq T_{1,2}.Interface \times T_3.Interface$ be a connection between $T_{1,2}$ and T_3. We can build the task $T_{(1,2),3}$ as the composition of $T_{1,2}$ and T_3 with respect to the connection $C_{(1,2),3}$ – cf. top of Figure 8. We can also first compose the tasks T_2 and T_3 with respect to a connection $C_{2,3} \subseteq T_2.Interface \times T_3.Interface$, which produces a task $T_{2,3}$, and then compose the task T_1 with $T_{2,3}$ with respect to a connection $C_{1,(2,3)} \subseteq T_1.Interface \times T_{2,3}.Interface$, resulting in a task $T_{1,(2,3)}$ – cf. bottom of Figure 8.

We have proved that the tasks $T_{(1,2),3}$ and $T_{1,(2,3)}$ are semantically equivalent, that is, their semantics, obtained by repeatedly applying Definition 4, is the same. This result holds provided $C_{1,2} \cup C_{(1,2),3} = C_{2,3} \cup C_{1,(2,3)}$. This condition says that the "links" between the components T_1, T_3 and T_3 are the "same" in both versions of the composition.

D. Hierarchical task

A hierarchical task H - Rule (11) - has a body that consists of a task T and a set of connections C. We assume that C connects interfaces of H and T, i.e., for all $(p_1, p_2) \in C$, $(p_1, p_2) \in (H.Interface \times T.Interface) \cup (T.Interface \times H.Interface)$. We also assume that H "hides" T from the outside, i.e., $\forall p \in T.Interface, \exists (p_1, p_2) \in C$ such that $p = p_1$ or $p = p_2$.

Definition 5 (Hierachical task): The semantics of a hierarchical task H, whose body is $T; C$, is given by the rule:

$$\frac{\tilde{\varepsilon} \xrightarrow{T} \tilde{\varepsilon}', \quad \forall (p_1, p_2) \in C, \varepsilon(p_1) = \tilde{\varepsilon}(p_2) \wedge \tilde{\varepsilon}'(p_1) = \varepsilon'(p_2)}{\varepsilon \xrightarrow{H} \varepsilon'}$$

□

Figure 9 illustrates a hierarchical task H including a task T. The above semantical rule just says that the ports of H "linked" by a connection with a port of T have the same value at the same time, and that all the "actual" computation $\tilde{\varepsilon} \xrightarrow{T} \tilde{\varepsilon}'$ is performed by T.

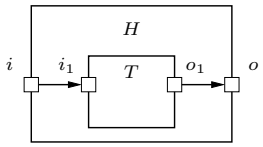

Figure 9. Hierarchical task.

E. Repetitive tasks (data parallelism)

A repetitive task R - Rule (12) - expresses data-parallelism. In Figure 2, T denotes the basic functionality to be replicated on different subsets of data obtained from the input arrays of task R. The resulting *instances* of T are assumed to be independent and schedulable following any order, even in parallel. In Rule (12), T is denoted by $Task$ in the task body. The attribute $\mathbf{s}$ is the possibly multi-dimensional *repetition space*. Each dimension of the repetition space can be seen as a parallel loop and its shape gives the bounds of the loop.

Each task instance consumes and produces *patterns* constructed via *tilers* - Rule (13), which are associated with each pair of ports, called connections *Cnx* in Rule (10).

Definition 6 (Repetitive task): Let R be a repetitive task, with $R.Interface = (\{i_1, \ldots i_n\}, \{o_1, \ldots, o_m\})$, $R.Body.Tiler = (\{ti_1, \ldots ti_n\}, \{to_1, \ldots, to_m\})$, $\phi = [\![R.Body.Task]\!]$, and $R.Body.Task.Interface = (i, o)$: The semantics of the task R is defined as follows:

$$\frac{\begin{array}{c}\forall \mathbf{k} \in \mathbf{0}..\mathbf{s},\ \varepsilon'_{\mathbf{k}}(o) = \phi(\varepsilon_{\mathbf{k}}(i)), \\ \bigwedge_{j=1}^{n} \varepsilon(i_j) = \biguplus_{ti_j} \{\varepsilon_{\mathbf{k}}(i'_j) | \mathbf{0} \leq \mathbf{k} < \mathbf{s}, (i_j, i'_j) = ti_j.Cnx\}, \\ \bigwedge_{l=1}^{m} \varepsilon'(o_l) = \biguplus_{to_l} \{\varepsilon'_{\mathbf{k}}(o'_l) | \mathbf{0} \leq \mathbf{k} < \mathbf{s}, (o_l, o'_l) = to_l.Cnx\}\end{array}}{\varepsilon \xrightarrow{R} \varepsilon'}$$

where $\mathbf{s} = R.Body.\mathbf{s}$. □

The above definition requires some explanations. The condition of the transition specified in Definition 6 has three parts. The first part requires that all the repetitions (indexed by the index-vector $\mathbf{k}$) of the tasks in the repetitive task's body complete their execution, with the effect that their output ports o are valuated according to some next-state environments $\varepsilon'_{\mathbf{k}}$; these values depend on the values of the input ports i according to some current environment $\varepsilon_{\mathbf{k}}$.

The second part of the transition condition in Definition 6 describes how the current environment ε for the repetitive task R is related to the current environments $\varepsilon_{\mathbf{k}}$ of the repetitions. Essentially, the condition says that for each input port i_j, for $j = 1, \ldots n$, $\varepsilon(i_j)$ is "tiled" by the corresponding input tiler ti_j, into the set of tiles: $\{\varepsilon_{\mathbf{k}}(i'_j) | \mathbf{0} \leq \mathbf{k} < R.Body.\mathbf{s}\}$, where i'_j is the port connected to i_j by the input tiler ti_j. The third part of the condition is similar to the second part - it describes how the next environment ε' for the repetitive task R is related to the next environments $\varepsilon'_{\mathbf{k}}$ according to the output tilers to_l $(l = 1, ..., m)$.

Note that the condition of the transition in Definition 6 may not be satisfiable. This happens when more than one tiler attempts to assign the same array element. When this is the case, the transition simply does not "fire", i.e., there is a deadlock in the operational semantics. Conversely, the absence of deadlocks guarantees the required "single-assignment" property for RSM specifications.

Repetitive tasks with an *inter-repetition dependency* are characterized by Rule (14). Cnx represents the pair of ports connected by the dependency link: one is an input to the repeated task T e.g., i', and the other is one of its outputs e.g., p'. The vector $\mathbf{d}$ specifies the coordinates of the inter-repetition dependency link on the repetition space. Initially, i' holds a default value, given by def. There can be several inter-repetition dependencies within a task, since an instance may require values from more than one instances to compute its outputs. This is why Rule (12) allows for a set of dependency link vectors $\{Ird\}$.

Definition 7 (Inter-repetition dependency): Let R be a repetitive task, with $R.Interface = (\{i_1, \ldots i_n\}, \{o_1, \ldots, o_m\})$, $R.Body.Task.Interface = (i, o)$, $R.Body.Tiler = (\{ti_1, \ldots ti_n\}, \{to_1, \ldots, to_m\})$, $\phi = [\![R.Body.Task]\!]$, $R.Body.Ird = \{\langle i_{j_l}, o_{j_l}, \mathbf{d}_{j_l}\rangle | l = 1, p\}$. The semantics of R is defined as follows:

$$\frac{\begin{array}{c}\forall \mathbf{k} \in \mathbf{0}..\mathbf{s}, \varepsilon'_{\mathbf{k}}(o) = \phi(\varepsilon_{\mathbf{k}}(i)) \ \wedge \ \bigwedge_{l=1}^{p} \varepsilon'_{\mathbf{k}}(i_{j_l}) = \varepsilon_{\mathbf{k}+\mathbf{d}_{j_l}}(o_{j_l}), \\ \bigwedge_{j=1}^{n} \varepsilon(i_j) = \biguplus_{ti_j} \{\varepsilon_{\mathbf{k}}(i'_j) | \mathbf{0} \leq \mathbf{k} < \mathbf{s}, (i_j, i'_j) = ti_j.Cnx\}, \\ \bigwedge_{l=1}^{m} \varepsilon'(o_l) = \biguplus_{to_l} \{\varepsilon'_{\mathbf{k}}(o'_l) | \mathbf{0} \leq \mathbf{k} < \mathbf{s}, (o_l, o'_l) = to_l.Cnx\}\end{array}}{\varepsilon \xrightarrow{R} \varepsilon'}$$

where $\mathbf{s} = R.Body.\mathbf{s}$. □

The transition rule specified in Definition 7 is quite similar to the one given in Definition 6. The difference is in the first part of the condition: now, certain inputs: i_{j_l} of the repeated task instances, take their next values from the current values of the outputs o_{j_l}, of other task instances, which are the "neighbors" of a current task instance $\mathbf{k}$ at "distances" specified by the vectors $\mathbf{d}_{j_l}$ - where, for $l = 1, \ldots, p$, $\langle i_{j_l}, o_{j_l}, \mathbf{d}_{j_l}\rangle$ are the members of the inter-repetition dependency set $R.Body.Ird$.

The rule given in Definition 7 may also fail to apply when its condition is unsatisfiable. In addition to multiple assignment problems, "inherited" from repetitive tasks (Definition 6), another reason for deadlocks is in Definition 7 that of *dependency cycles*. Conversely, the absence of deadlocks guarantees the absence of dependency cycles, also required from RSM specifications.

F. Reshape

A reshape task S – Rule (15) – has a body that consists of two tasks T_1, T_2 and a reshape connection Rsp. We assume that T_1 and T_2 have one input port and one output port each, i.e., $T_1.Interface = (\{i_1\}, \{o_1\})$ and $T_1.Interface = (\{i_2\}, \{o_2\})$, where i_1, o_1, i_2, o_2 are distinct ports (see also Figure 5). For the reshape connection, $Rsp \equiv \langle t_1, t_2, \mathbf{s}, \mathbf{p}\rangle$, – Rule (16), where t_1 and t_2 are tilers, $\mathbf{s}$ is a (typically, multi-dimensional) repetition space, and $\mathbf{p}$ is the shape of patterns circulating between the two tilers.

Definition 8 (Reshape task): A reshape task $S = T_1, T_2, Rsp$ is the composition between the tasks T_1, a new repetitive task Rid, defined below, and the task T_2. The task Rid consists of:

- $Rid.Interface = (\{t_1.Cnx.o\}, \{t_2.Cnx.i\})$,
- $Rid.Body^r = t_1, t_2; Rsp.\mathbf{s}; I; \{\}$, where I is an elementary task (see Definition 3) with interface $I.Interface = (\{i_I\}, \{o_I\})$, s.t. $i_I.shape = o_I.shape = Rsp.\mathbf{p}$, and $i_I.type = o_I.type = t_1.Cnx.o.type = t_2.Cnx.i.type$. The body of I is the identity function on $\varepsilon_{I.Interface}$.

The composition of the tasks T_1, Rid, and T_2 is performed with respect to the connections $(o_2, t_1.Cnx.o)$ between T_1 and Rid and $(t_2.Cnx.o, i_2)$ between Rid and T_2. □

Since a reshape task is defined in terms of existing constructions (compositions, repetitive tasks, *etc.*) its semantics can derived from the semantics of its components.

IV. Extending RSM for control-based design

In the repetitive MoC the same computation is repeated forever. For some applications such as systematic signal processing this model is expressive enough. However, for other applications, this MoC is too limited; one wishes at least for the possibility to switch several computations, that is, to extend the repetitive MoC with some *control* - just like Mode Automata [11] propose a control-oriented extension for dataflow languages. The combination of control-oriented and parallel computation concepts has been previously studied in languages, such as Mentat [12], PSather [13] and MasPar programming language [14]. A *control parallelism* is considered in the form of a concurrent execution of different instruction streams. Control is described by using mostly the usual system-level scheduling and synchronization mechanisms, e.g., *fork/join*, master/slave model or monitors. For control-oriented design with RSM, we have introduced in [15], [16] a construction called a *mode task*,

presented below. Similar control-oriented concepts are introduced in other dataflow models to express dynamic changes or reconfiguration in streaming applications [17] [18].

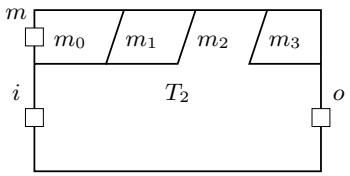

Figure 10. A mode task.

A. Mode tasks

A mode task M expresses a choice among several possible alternative computations denoted by tasks T_j, also called *modes*. Figure 10 illustrates a mode task, in a notation inspired by windows with multiple tabs. Here, the mode task M is composed of four modes $T_0 \dots T_3$, each identified by a corresponding mode value: $m_0 \dots m_3$. The mode task M has a distinguished input port m, called the *task selector*. When the port m holds the value m_k, the computation performed by M is that performed by T_k. The modes run exclusively, meaning that whenever a mode task executes, only the task T_k associated with the selected mode m_k is computed. This is particularly useful when analyzing the behavior of the mode task since it eliminates by construction the risk for possible interaction between faulty and non-faulty modes, hence favoring safe designs.

All the modes T_k of M have the same interface, satisfying $T_k.Interface = M.Interface \backslash \{m\}$. This also defines the *interface* of M. The *body* of M is the set $\{T_k \mid k \in I\}$ of its modes, where I is a finite set of indices.

Definition 9 (Mode task): The semantics of a mode task M is described by the following rule:

$$\frac{M.Body = \{T_k \mid k \in I\} \quad \varepsilon(m) = j \in I, \quad \varepsilon \xrightarrow{T_j} \varepsilon'}{\varepsilon \xrightarrow{M} \varepsilon'}$$

where m is the Selector of M. □

The rule says that the computation performed by M is that performed by the mode $T_j : \varepsilon \xrightarrow{T_j} \varepsilon'$, when, in the current environment ϵ, the value $\epsilon(m)$ of the selector port m is j.

B. Example: a color effect switch

To illustrate the use of mode tasks in data-intensive applications, let us consider a scenario of adaptation in a multimedia phone integrating streaming applications that provide the user with video-on-demand programs, or television broadcast. Such applications are data-intensive and often perform in different modes in order to fulfill their functionality according to various criteria. Among important aspects to be taken into account for guaranteeing quality of service regarding image display, are video effects, resolution and compression level.

The macro-component *ColorEffect* depicted in Figure 11 represents a composition of two sub-components: *ColorControl* denoting a transition function and a mode task composed of two image style modes. It is expected to execute in the context of a repetitive task with inter-repetition dependency as the task R shown in Figure 4. Typically, it replaces the repeated task T in R. The inter-repetition dependency link within R therefore connects the port m to c_m. The resulting R task has a similar semantics as synchronous Mode Automata [11] where the repeated component consists of the hierarchical task `ColorEffect`, in which a mode task executes a data-intensive algorithm to define the color effect of received images, depending on selected color mode.

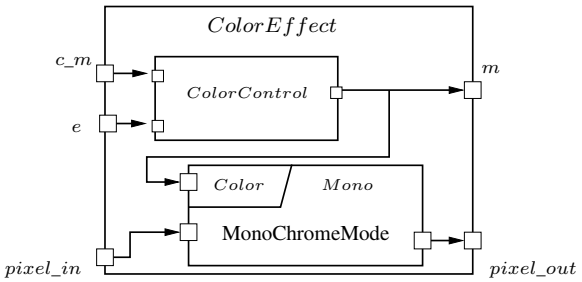

Figure 11. Composition of a transition function and a mode switch.

The *ColorControl* sub-component produces mode values consumed by the mode task in order to transform the pixels of an image according to the selected mode. The input data e is a Boolean-valued event that enables to chose among the two possible mode values in *ColorControl*. The input data c_m denotes the previous mode value m computed by *ColorControl*. It is used to determine the next mode w.r.t. the selected mode in the mode task. In practice, the *ColorControl* component is implemented by a transition function in which each state is associated with a mode, e.g., a one-to-one mapping between states and modes.

The specified mode task has two image effect computation modes: *ColorMode* and *MonochromeMode*, associated with adequate image filters. Its input m denotes at any time the selected mode that applies to input pixels to produce output ones. The two modes induce different resource requirements in terms of energy, communication quality, computing resource and memory usage. When considering such a component in an embedded system, e.g., a mobile phone, it is important to be able to switch between these modes for optimizing the usage of system resources [19].

Now that we have defined an operational semantics of RSM, used as input model in our co-modeling framework GASPARD [7] for data-parallel system-on-chip, we can exploit it to establish the behavioral correctness of model transformations in GASPARD. In particular, we will consider the

refinement of RSM into the synchronous language SIGNAL. This refinement has been defined for the formal verification of functional properties of RSM-based application models. SIGNAL already has a formal semantics. We can consider a technique such as the one presented in [20] to check the semantic consistency of initial and final models, respectively expressed in RSM and SIGNAL.

V. CONCLUSIONS

This paper presented an operational semantics for the repetitive structure modeling (RSM) package of the Marte profile. RSM proposes useful mechanisms allowing one to express, in a compact way, the potential parallelism available in a system: task and data parallelism in applications, multi-processor architectures, etc. A major goal of this semantics is clarify the meaning RSM concepts and to enable the verification and execution of models specified with these concepts. This operational semantics will be considered as a foundational formal basis to reason RSM modeling in our model-driven engineering framework GASPARD [7]. As a short-term perspective, we are interested in the correctness of model transformations towards various target languages in GASPARD. We will consider first the transformation from RSM to the synchronous language SIGNAL, which also has a formal operational semantics. Its correctness issue can be treated as a behavioral equivalence problem between a RSM model and its corresponding SIGNAL program, *w.r.t.* their respective operational semantics.

REFERENCES

[1] OMG, Ed., *Final Adopted OMG SysML Specification*. http://www.omg.org/cgi-bin/doc?ptc/06-0504, May 2006.

[2] ——, *(UML) Profile for Schedulability, Performance, and Time Version 1.1*. http://www.omg.org/technology/documents/formal/schedulability.htm, Jan. 2005.

[3] B. Selic, "Using uml for modeling complex real-time systems," in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*. London, UK: Springer, 1998, pp. 250–260.

[4] G. Martin, L. Lavagno, and J. Louis-Guerin, "Embedded uml: a merger of real-time uml and co-design," in *Proceedings of the 9th International Symposium on Hardware/Software Codesign (CODES'01)*, 2001, pp. 23–28.

[5] Object Management Group, "A UML profile for MARTE," 2007, http://www.omgmarte.org.

[6] P. Boulet, "Formal semantics of Array-OL, a domain specific language for intensive multidimensional signal processing," Inria, INRIA Research report RR-6467, Mar. 2008. [Online]. Available: http://hal.inria.fr/inria-00261178/en/

[7] A. Gamatié, S. Le Beux, E. Piel, A. Etien, R. Ben Atitallah, P. Marquet, and J.-L. Dekeyser, "A Model Driven Design Framework for High Performance Embedded Systems," INRIA, Research report 6614, Aug. 2008. [Online]. Available: http://hal.inria.fr/inria-00311115/fr/

[8] Object Management Group, "Semantics of a foundational subset for executable uml models (fuml)," 2009, http://www.omg.org/spec/FUML/1.0/Beta1.

[9] A. Cuccuru, C. Mraidha, F. Terrier, and S. Gérard, "Enhancing uml extensions with operational semantics," in *Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'07)*, 2007.

[10] A. Demeure and Y. Del Gallo, "An array approach for signal processing design," in *Proceedings of Sophia-Antipolis conference on Micro-Electronics (SAME'98), System-on-Chip Session, France*, Oct. 1998.

[11] F. Maraninchi and Y. Rémond, "Mode-automata: a new domain-specific construct for the development of safe critical systems," *Science of Comp. Prog.*, vol. 46, no. 1-2, pp. 219–254, 2003.

[12] E. A. West, "Combining Control and Data Parallelism: Data Parallel Extensions to the Mentat Programming Language," University of Virginia, Charlottesville, VA, USA, Tech. Rep. CS-94-16, 18, 1994.

[13] C.-C. Lim, J. Feldman, and S. Murer, "Unifying control- and data-parallelism in an object-oriented language," in *Joint Symposium on Parallel Processing*, Waseda Univ., Tokyo, 1993, pp. 261–268.

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

[15] O. Labbani, J.-L. Dekeyser, P. Boulet, and E. Rutten, "Introducing Control in the Gaspard2 Data-Parallel Metamodel: Synchronous Approach," *Workshop on Modeling and Analysis of Real-Time and Embedded Systems*, Oct. 2005.

[16] A. Gamatié, E. Rutten, and H. Yu, "A model for the mixed-design of data-intensive and control-oriented embedded systems," INRIA, France, Research report 6589, Jul. 2008. [Online]. Available: http://hal.inria.fr/inria-00293909/en

[17] B. Theelen, M. Geilen, T. Basten, J. Voeten, S. Gheorghita, and S. Stuijk, "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," in *4th Int'l Conf. on Formal Methods and Models for Co-Design*, july 2006, pp. 185–194.

[18] S. Neuendorffer and E. Lee, "Hierarchical reconfiguration of dataflow models," in *2nd Int'l Conf. on Formal Methods and Models for Co-Design*, june 2004, pp. 179–188.

[19] A. Gamatié, H. Yu, G. Delaval, and E. Rutten, "A case study on controller synthesis for data-intensive embedded systems," in *Proceedings of the 6th IEEE International Conference on Embedded Software and Systems, ICESS'09,* HangZhou, Zhejiang, China, May 25–27, 2009, 2009.

[20] V. Rusu and D. Lucanu, "Checking semantical consistency based on observational simulations," in *Journées IDM'2010 (Ingénierie Dirigée par le Modèles)*, Pau, France, March 2010.