A Generic Execution Framework for Models of Computation

Cécile Hardebolle Frédéric Boulanger Dominique Marcadet Guy Vidal-Naquet

Supélec – Computer Science Department
3 rue Joliot-Curie, 91192 Gif-sur-Yvette cedex, France
{firstname.name}@supelec.fr

Abstract

The Model Driven Engineering approach has had an important impact on the methods used for the conception of systems. However, some important difficult points remain in this domain. In this paper, we focus on problems related to the heterogeneity of the computation models (and therefore of the modeling techniques) used for the different aspects of a system and to the validation and the execution of a model. We present here a language for describing computation models, coupled with a generic execution platform where different computation models as well as their composition can be interpreted. Our goal is to be able to describe precisely the semantics of the computation models underlying Domain Specific Languages, and to allow the interpretation of these models within our platform. This provides for a non ambiguous definition of the behavior of heterogeneous models of a system, which is essential for validation, simulation and code generation.

1. Introduction

From the Model Driven Engineering (MDE) point of view, the design of a system can be considered as a sequence of modeling stages. The process starts with high level specifications and then refines and transforms the models until they are executable on the target platform, leading to an implementation. These models describe the properties and the possible behaviors of the designed system. They can be used for analysis, simulation, execution or validation of the system.

We consider the design of complex systems, that calls for several technical fields. Each technical field has its own design techniques and modeling languages, that are suitable for its specific needs. For instance, the design of the control of an industrial plant and the design of an image processing algorithm may use different modeling languages. However, if the plant uses video cameras to inspect and sort out defective products, such image processing algorithms are part of the global model of the plant. Therefore, several models of the system, which cover different technical aspects and use different modeling languages, can be used at the different stages of the design process.

When changes are made to these models without automatic synchronization, different problems arise. First of all, it is difficult to keep these models coherent, mainly because they use different formats and are managed in different platforms. Furthermore, if these models are unconnected at the level of abstraction to which they belong, their integration into the global model of the system can occur only at a lower level that does not capture the design choices that have been made in these models. This integration is typically done using a programming language instead of a modeling language. The problem is that programming languages tell how things are done, not what they are supposed to do and why we chose to do them that particular way. Integration of high level models in a low level language deprives us of useful alternatives that should have been preserved until the implementation stage.

Moreover, when problems are detected in the most refined system model, the identification of what should be changed in the models of the subsystems to solve these problems can be a very complex task. *Heterogeneous modeling techniques* allow to describe the whole system as a composition of subsystems described with different languages, thus avoiding that kind of issues.

Component oriented modeling languages, which are increasingly used for reusability and testability reasons, naturally allow heterogeneous modeling. In this context, we studied Lee's work on the association of component based techniques and hierarchical heterogeneity [7]. Lee defines a concept that is fundamental in our approach: the concept of *Model of Computation (MoC)*.

This work is partially financed by the SYSTEM@TIC PARIS-REGION Competitiveness Cluster.

A model of computation is a formal description of the behavioral aspect of a modeling method. In the context of component oriented modeling languages, the model of computation allows to define the behavior of the model of a system by composing the behaviors of the models of its components. In other words, it is the set of rules that allows to compute the behavior resulting from the composition of the individual behaviors of the components.

Due to the variety of technical fields and related modeling techniques, there are plenty of specialized models of computations, such as Synchronous Dataflow (SDF), Communicating Sequential Processes (CSP), Petri-Nets or Finite State Machines (FSM) for instance.

Natural language is generally used to define such models of computation. This method is simple but leads to ambiguities, especially on the meaning of the MoC for modelers, but also for programmers or for analysts who use formal tools. For instance, several programmers in charge of the implementation of a component may not understand the model the same way. Similarly, lack of precision in the understanding of the MoC that underlies a model of a system may lead to an inaccurate analysis. When described in natural language, MoCs cannot be executable in a reliable way. This makes any computation of the behavior of the system impossible, thus preventing simulation or validation for example.

We call *model of execution* an operational specialization of a model of computation. Several models of execution may exist for the same model of computation. For instance, there are different ways of solving differential equations representing the behavior of a system: one may use either Euler's method or Runge-Kuta's (even though these methods only approximate the behavior defined by the differential equations). Describing the execution of a model of computation with a well designed language therefore removes possible ambiguities.

In previous works ([2], [3]), we studied hierarchy and component encapsulation in the perspective of handling heterogeneity. In this paper, we are interested in the heterogeneity of models of computation in the domain of *predictive modeling*, which aims at predicting and validating the behavior of a system. The approach we present here targets the execution of models that use different models of computation at different levels of their hierarchy. It comprises two complementary parts: a vocabulary for describing models of systems and a set of primitive execution operations for models of computation. The semantics of these generic operations is described using our vocabulary.

The remainder of this paper is structured as follows. Section 2 summarizes some of the related work in this area. We expose the major concepts of our approach in section 3. Section 4 describes the semantics of our primitive execution model. We detail all the operations of our execution model in section 5. Section 6 presents an example of use of our framework on a modal system. We conclude in section 7.

2. Related work

Many specific models of computation have been individually studied and formalized. More recently, a few attempts have been made in order to find unified (and sometimes formal) approaches that allow to describe different models of computation with a unique set of concepts. These works all take the internal mechanisms of components into account when defining the composition of their behaviors.

As a foundation of these approaches, Willems provides in his Behavioral Approach [15] a formal framework for modeling and analyzing dynamic systems.

But the very first general framework allowing to study and compare different models of computation is the Tagged Signal Model [11] of Lee and Sangiovanni-Vincentelli. The Tagged Signal Model is a formal semantic framework that uses a mathematical and denotational formalism based on the domain theory.

The work on Trace Algebras begun by Burch [5] in his thesis at Stanford brings a powerful general environment in which it is possible to guarantee the properties of the combination of heterogeneous models of computation. In the context of this approach, a trace represents a behavior among all the possible behaviors of a system.

The latest approach we have seen is Rosetta [8], a system specification language created by Kong and Alexander. Rosetta has a formal semantics. It allows to combine models of computation either in order to assemble components or to model different aspects of one component using facets. A facet models a particular aspect of a component, e.g. its behavior or its energy consumption.

But in spite of the potential of the facet concept, Rosetta cannot be used for predictive modeling since a Rosetta specification is not executable. This is a common feature of the approaches we have examined. This lack of executability is mainly due to the fact that all these approaches use the description of the internal mechanisms of components in order to describe their composition. Our work differs from these approaches by considering that it is neither necessary nor desirable to take the internal mechanisms of components into account for the description of a model of computation. Indeed, as detailed in section 3, our approach is based on the *observation* of the behaviors of components, not on the description of the internal working of components.

The study of these approaches allowed us to identify important concepts that underlie models of computation. One of the most important concepts we listed is the concept of *time*. Time is a key notion in most of the models of computation but it is used in various ways. The Discrete Events MoC (DE) and the Synchronous Data Flow MoC (SDF) il-

lustrate our point: in the DE model every piece of information exchanged between components is stamped with a date given by a global clock, whereas in the SDF model, data is strictly ordered only between pairs of communicating components. Communication modes, synchronization modes and concurrency are other important concepts we pointed out.

Regarding the executability issue, we studied the way heterogeneous modeling tools address it. In particular, we studied Ptolemy II [4], which is developed within a project led by E. A. Lee and conducted at the University of Berkeley. It is one of the richest tools in terms of number of implemented models of computation. Ptolemy II is an actor oriented experimental framework coded in Java. In Ptolemy II, multiple models of computation (which are implemented by the notion of "Domain") can be used in a same model thanks to the concept of composite actors. Ptolemy models are executable. However, Ptolemy II is not yet mathematically founded and that makes formal analysis or validation quite impossible. Furthermore, the only tool that can be used to add a model of computation to the framework is a programming language (Java) and the reference definition of a model of computation is its implementation. Similarly, the semantics of the composition of models of computation and of inter-domain interactions are implicitly coded in the core of Ptolemy.

Other teams in the meta-modeling community tried to build a general meta-model that would cover most of the models of computation while preserving the executable aspect. The Triskell team from INRIA chose such an approach to create the Kermeta language [13]. Kermeta can be seen as a "meta-programming language". It offers a generic mechanism based on the weaving of models to allow the extension of existing meta-modeling languages (MOF and Essential MOF – EMOF – in particular) with the ability to manage actions. Other similar approaches exist: for instance Xactium and the eXecutable Modelling Facility (XMF) framework, implemented in the XMF-Mosaic tool [6].

These latter works are important in the context of our study because they address Domain Specific Languages (DSLs) executability and consequently MoCs executability (since the semantics of any DSL is described by a MoC). However, both Kermeta and XMF are described as executable meta-languages that are meant to help the creation of new executable DSLs while our goal is not to create but to combine DSLs in a same executable model of a system.

3. Major concepts of our framework

Executing a model involves the computation of the evolution of observable parameters of the model in response to the evolution of input data representing the system environment. In our context, the execution of a model is a digital operation. It is thus discrete and composed of a sequence of computation steps.

Generally speaking, the computation operations underlying a model of computation can be put in two categories: generic computation operations and computation operations specific to the semantics of the MoC. In our approach, we determine and factorize the generic computation operations (that is, operations that are shared by most models of computation) and we provide support for the definition of the specific operations. In order to guarantee the genericity and the modularity of our approach, we based our work on two paradigms: the black box paradigm and the stroboscope paradigm.

The black box principle relies on two key ideas. First of all, a component of a system is considered as a closed element: its functioning is independent from its environment (i.e. from the system). Secondly, we do not care about how it works, it is sufficient to observe what it generates for its environment. This principle ensures the modularity of our execution platform: the replacement of a component by another with the same interface does not change the way its behavior is composed with the behavior of its siblings.

The stroboscope paradigm is used in physics for the study of movements and in the study of distributed algorithms (see [14]). The mechanism is the following: first, one takes a series of instantaneous photographs of the moving object at chosen times, then, the superimposition of these different photographs makes it possible to analyze the movement. In our approach, we use a similar principle for the execution of a model. We call our instantaneous photographs *snapshots*. The exclusive use of snapshots, i.e. pure observations, ensures the genericity of our approach since any system can be observed.

In the context of a component-based approach, the observation of a system at a chosen instant is made from the observation of each of its components at the same instant. The notion of "same instant" depends on the models of computation involved in the model of the system and of its components. Moreover, without knowledge of the machine that will compute the execution of the model and in order to be as generic as possible, we assume that a scheduling of the observations of the different components is necessary. Now, just as taking the picture of a landscape is different from taking the picture of a moving person, the scheduling of the snapshots of the system's components depends on the model of computation used in the model of the system. An advantage of our approach is that it provides a generic framework to define precisely how to compute the snapshots.

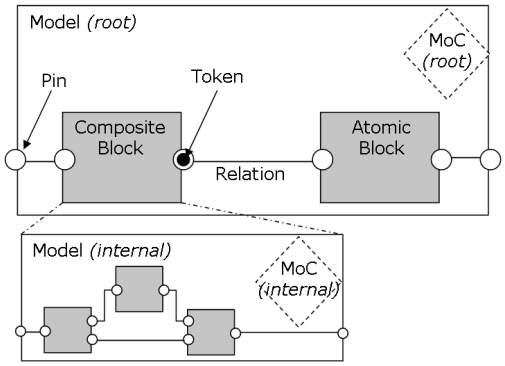

Figure 1. Roles of our modeling objects

4. Modeling vocabulary

To represent a model of a system, we use a set of elementary objects. These objects compose what we call our modeling vocabulary. They support basic operations allowing, for instance, to read/write their parameters, to access objects they contain and most importantly to execute them. The role of these objects is illustrated on figure 1. We use the following objects:

Model What we call a model is the description of the behavior of a system. In our point of view, a model is both structural and functional: it includes a structural description of the system in the form of a set of blocks (or components) and, since each block realizes a particular function, the whole is a functional description of the system. A model contains objects called blocks and has pins for interaction with its environment.

Model of Computation A model of computation is associated to every model in order to provide semantics for the combination of the behaviors of the blocks of the model. The model of computation implements execution primitives (see section 5).

Block Blocks are elementary elements used in the description of the behavior of a system. They can represent components (in the general meaning of the word) but also other basic concepts such as the concept of state. Blocks interact through their pins. The behavior of a block may be described using a model (hierarchical composition). In this case, the block is said to be composite. Otherwise, its behavior may be described using a formalism outside of our framework. It is then said to be atomic. In both cases, a block must have the same set of execution operations defined in section 5.3 below.

Pin Pins can represent the notion of port, of interface or even of control nodes. The direction of pins can be specified (in, out or both). Other attributes may qualify them according to the MoC, for instance to be able to distinguish data pins from control pins.

Relation Relations connect block pins. The semantics of relations depends on the model of computation according to which the model is interpreted.

Token A token represents a piece of information exchanged between blocks. The elementary amount of information as well as its type and its properties depends on the model of computation involved.

The UML diagram of figure 2 shows the relations between the different objects we introduced along with the operations that they support. The operations related to the platform, the models of computation and the blocks are described in section 5. Notice that pins have specific operations allowing to put tokens, to read and to consume available tokens.

The set of objects we have presented in this section along with their operations is the basis of our model of computation description language.

5. Execution operations

The various existing models of computation have different rules for transporting data between components, specifying causal dependencies between inputs and outputs and specifying how data is produced and when it is available. For instance, in the Synchronous Data Flow [10] model of computation, data is produced a fixed number of tokens at a time and instantaneously propagated between components. The causal dependencies are therefore statically deduced from the connections between components. On the contrary, in the Discrete Events [12] model of computation, the production of a data token by a component denotes the occurrence of an event. Such an event has a time-stamp that specifies when the data will be available. Data propagation is done according to a global clock that determines when an event has to be delivered to its target. Therefore, causal dependencies are dynamical because they depend both on the connections between components and on the timestamps of the events.

Our generic model of computation has been designed in order to allow the detailed specification of such rules. We have identified three sets of primitive operations that we detail below.

5.1. Platform operations

The execution platform operations form the backbone of our framework. The platform is in charge of the overall

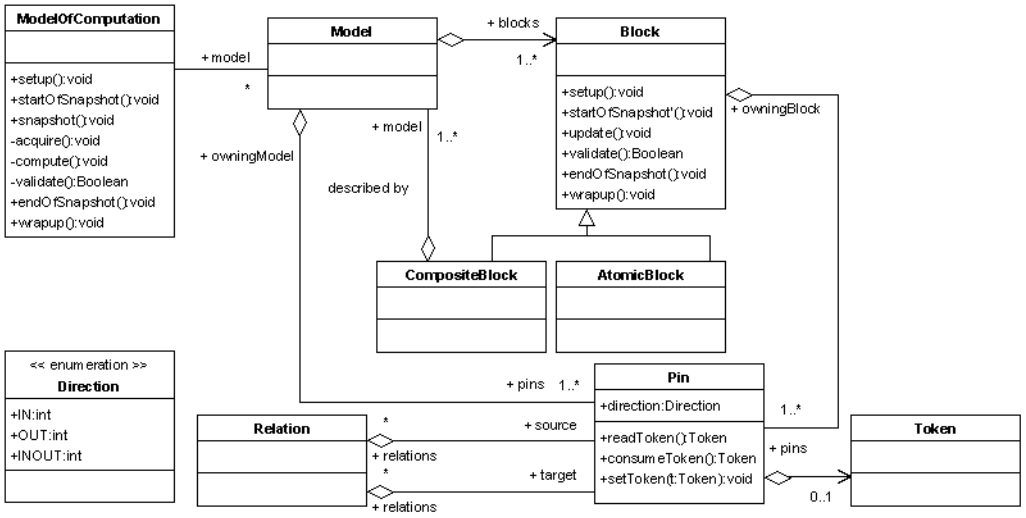

Figure 2. UML diagram representing our modeling concepts

execution of the model. It implements the loop that computes successive snapshots in order to obtain a complete execution. The number of snapshots to build is specified, explicitly or implicitly (with a constraint for example), by the modeler who wants to simulate the system.

The operations that structures the platform engine, represented on figure 3, are the followings:

`setup()` The `setup` operation is a pre-initialization operation that includes checking the structure of the model which has to be correct with respect to the model of computation it involves.

`startOfSnapshot()` This operation prepares the model for execution. That may include, for instance, initialization of variables.

`snapshot()` The `snapshot` operation builds one observation of the system behavior. The computation is delegated to the `snapshot` operation of each model of computation.

`endOfSnapshot()` The observation ends in the `endOfSnapshot` operation. The state of the model is then considered as stable.

`newSnapshot()` The `newSnapshot` operation checks if, with respect to the parameters given by the modeler for execution, it is necessary to proceed with a new snapshot. If so, the platform loops on the `startOfSnapshot` operation, else it carries on to the `wrapup` operation.

`wrapup()` The `wrapup` operation is the final step of execution. During this step, simulation results are displayed and objects remaining in the memory are destroyed.

5.2. Model of computation operations

The MoC operation set is depicted on figure 4. This set is generic since it is absolutely the same for all models of computation. By contrast, the content of each execution operation depends on the model of computation.

The `setup`, the `sartOfSnapshot`, the `endOfSnapshot` and the `wrapup` operations are the MoC counterparts of the homonym platform operations. For instance, the static check of the model structure that is realized in the `setup` operation differs totally according to the model of computation.

The `snapshot` operation builds the observation of the system behavior. Its successive steps form two loops that are repeated until the observation is considered to be complete:

`acquire()` The acquisition of the environment of the model is realized during operation `acquire`. Then, acquired data is considered as stable until the end of the snapshot and is present on the model pins: the goal of the snapshot is to determine the behavior of the model based on the acquired data. Consequently, if the data change during the computation, the snapshot would be inconsistent.

`compute()` The `compute` operation is the most important operation: it computes the observation of the model. It is composed of different sub-operations described below, which can be roughly classified in two classes: scheduling operations and data propagation operations.

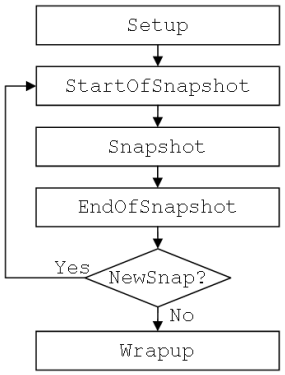

Figure 3. Platform operations

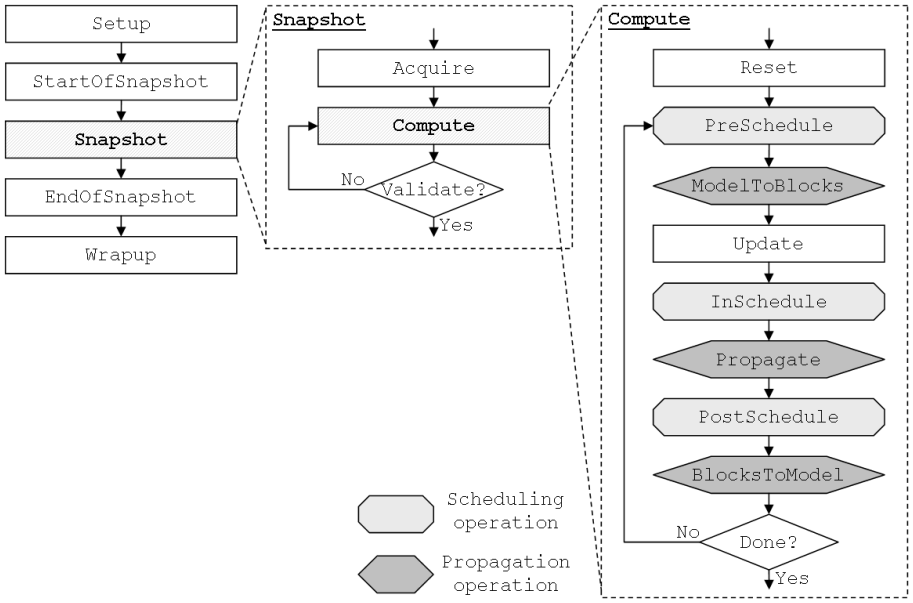

Figure 4. MoC operations

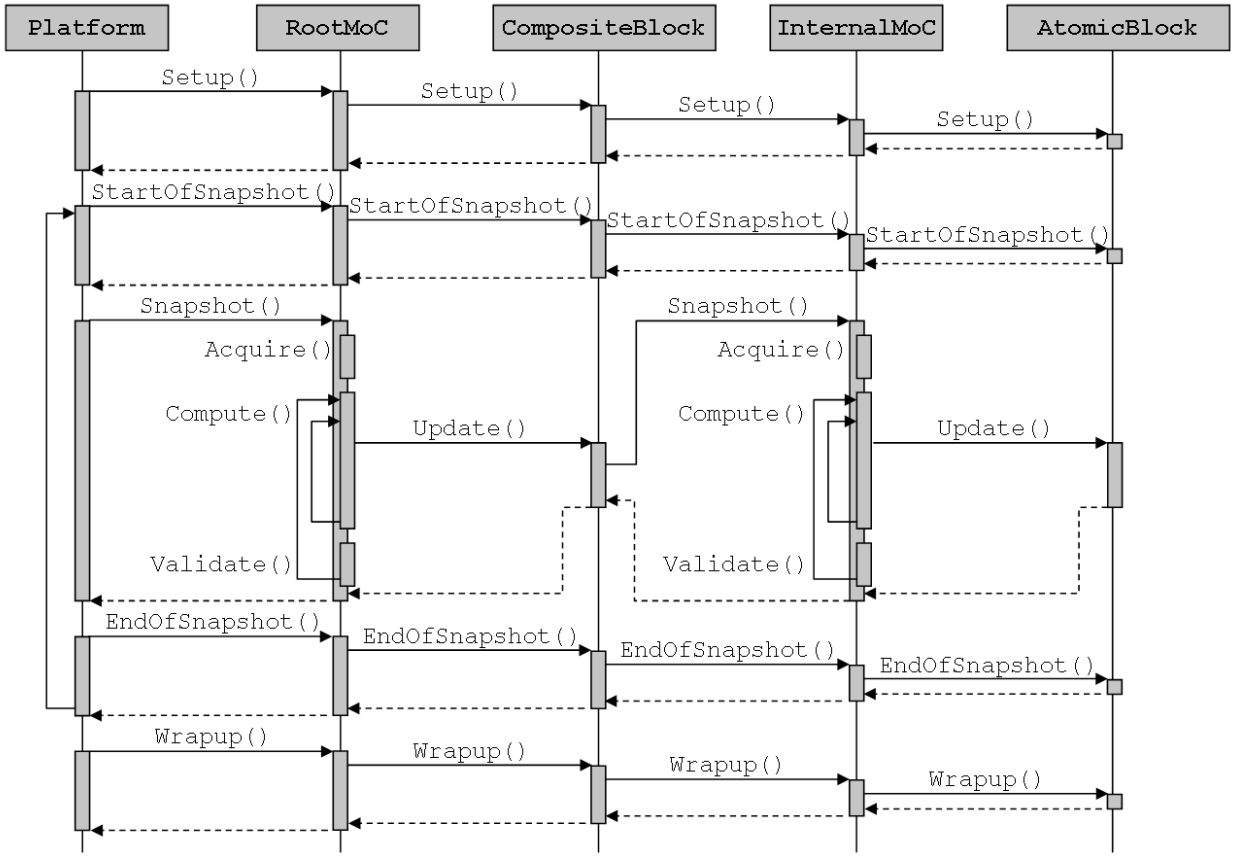

Figure 5. Hierarchical execution sequence

reset() The goal of the `reset` operation is to reset the observable state of the objects. For example, outputs may be reset to an "unknown" status, etc. The way of performing the reset depends on the model of computation.

preSchedule() The `preSchedule` operation is part of the scheduling operations. Its role is to determine the order in which blocks have to be observed depending on the status of the model and the data available on its pins.

modelToBlocks() This operation propagates data over relations connecting model input pins to block input pins. The way it operates may depend on the `preSchedule` operation which may have restricted the number of blocks concerned by the propagation.

update() During this operation, blocks are asked to update their output pins.

inSchedule() `inSchedule` is a scheduling operation. It takes into account new data on output pins.

propagate() `propagate` propagates observable output data from blocks to the input pins of blocks they are connected to.

postSchedule() `postSchedule` is a scheduling operation. It takes into account new data on input pins.

blocksToModel() `blocksToModel` propagates observable output data from blocks to the model output pins they are connected to.

done() The `done` operation checks if the observation is complete (what depends on the model of computation). If it is the case, it carries on with the `validate` operation. Else, it loops on the `preSchedule` operation.

validate() During the `validate` operation, blocks have to validate the computed observation. Thanks to this operation, blocks can influence the course of the computation. This is particularly useful in the case of systems depending on models of computation that require a progressive readjustment of parameters in order to obtain the most precise observation possible. A signal crossing detector is a perfect example of that kind of systems. The detector samples an input signal at periodic times. If it finds out that the signal has crossed the fixed threshold but is not able to determine the date of that event with enough precision, it can require a new computation of the observation with a smaller sampling step until the date of the event is precise enough.

The semantics of each of these operations is described in our modeling language (see section 4) for every model of computation one wants to work with. As soon as the platform has such a description of a model of computation, it is able to execute any model that uses it.

5.3. Block operations

Every block must support the execution operations required by the platform. That means that if the behavior of a block is defined using a formalism different from ours (using C++ or Esterel for example), its code has to be wrapped in an interface that maps its behavior to our operations. If some of the required operations are not defined in the code of the block, they can be added using our language.

First of all, every block must have the `setup`, the `startOfSnapshot`, the `endOfSnapshot` and the `wrapup` operations which have the same meaning in the platform. Every block also must have an `update` operation. It is the most important operation because it triggers the update of the output pins of the block. Note that a block may be active at any time, even when we do not observe it. However, when we invoke its `update` operation, it must provide us with a coherent view of its outputs, that will be taken into account by other blocks. Last, every block must have a `validate` operation that allows it to act on the overall execution of the model.

5.4. Hierarchy

In accordance with our observation principle, we designed the three sets of operations we described above so as to allow the execution of complex hierarchical models while ensuring a minimum dependency between a model and the models of its components. Let us consider the simple hierarchical model example of figure 1: its behavior is described by a root model using a root MoC and composed of at least one composite block whose behavior is described by an internal model using an internal MoC (possibly different from the root MoC) and composed of atomic blocks. Figure 5 shows the sequence of execution operations on this example.

6. Example: a simple modal system

We have executed different kinds of models using our framework in order to test it. The example presented in this section is the model of a simple modal system. A system is modal if its behavior can be decomposed into several modes. Mode changes happen when conditions on the values of the inputs, the outputs or the parameters of the system are satisfied. A modal model can be seen as a more general kind of hybrid model, in which modes define a behavior that

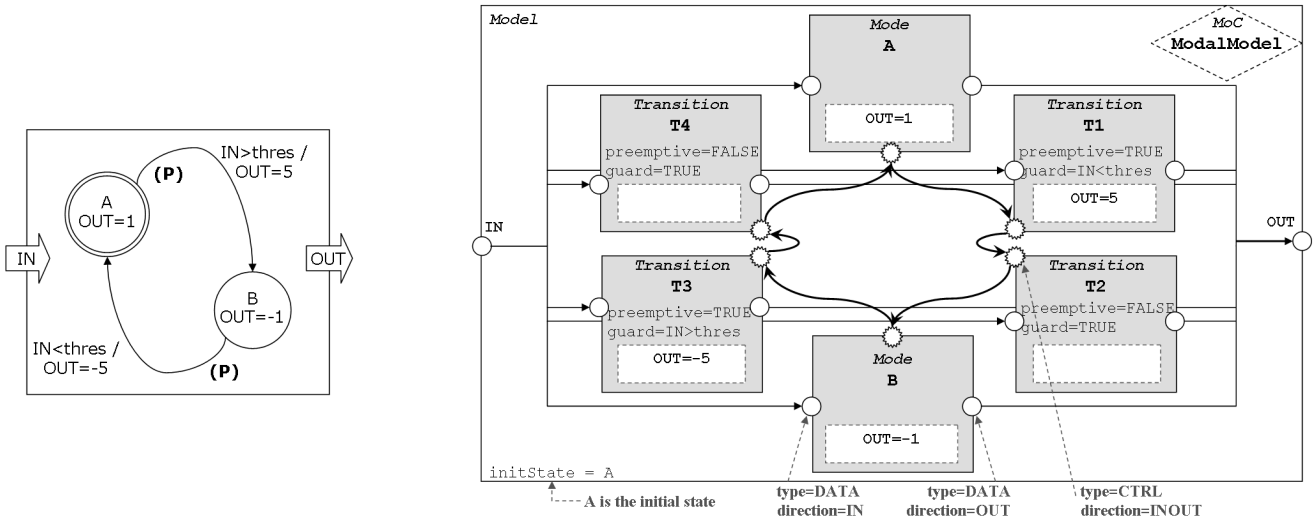

Figure 6. Example modal model and its representation in our framework

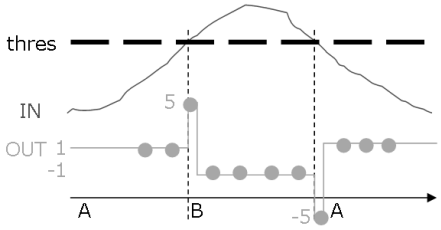

Figure 7. Expected behavior of the modal system

is continuous in time. Many systems are modal. Satellites, for example, are modal systems that generally have at least three different modes: a "launch" mode used for moving the satellite to its orbit; a "nominal" mode used when the satellite is ready to accomplish its mission, like taking pictures of the Earth; and a "safe" mode that allows the satellite to wait for new orders safely. These modes are generally refined into sub-modes and the conditions of change are strictly defined and checked so that the behavior of the satellite is predictable in any situation. Any malfunction would lead very likely to the loss of the satellite...

In this example, we are interested in the model of computation of the *charts approach [9] because it is particularly suitable for representing modal systems. In *charts, a modal model is hierarchical. The root model is a finite state machine with states that represent modes and describe different behaviors of the system. The behavior of the system in each mode is described by a sub-model that can be based on any model of computation. For instance, a sub-model can be a finite state machine and in this case the overall model is a hierarchical finite state machine, or it can be a continuous timed model and the overall model is then a hybrid model.

Our modal model example is shown on figure 6.The behavior of the system in each mode is the emission of a constant valued signal. Notice that even if the behavior of the system had been more complex, the global way the execution of the model is made would not have changed. Indeed, the computation is based exclusively on observations and is not modified by the way observations are computed.

The modal system receives a signal on its input, called IN, and emits another signal on its output, called OUT. It has two modes: in mode A it emits the value 1 on its output and in mode B it emits -1 on its output. The system swaps modes when the received signal crosses a given threshold. Transitions between modes are preemptive: they specify a behavior (just like a mode does) that has to be executed instead of the behavior specified by the target mode. The initial mode of the system is mode A. The figure 7 shows how the system should behave in response to an example of input signal.

Figure 6 shows the internal structure of the modal model in our framework. Modes are represented with blocks. Each block is connected to the input and the output of the model because only one block at a time can represent the behavior of the system. It is as if the model of one of the blocks was substituted to the model of the system at each instant. Blocks have data pins for input and output signals and control pins that allow relations to represent the sequence of mode changes. Transitions are represented by specific blocks with attributes that indicate if they are preemptive and what is their guard. Preemptive transitions have to be broken in two distinct blocks: one representing the behavior of the system when taking the transition and another, of

which the guard is always true, representing the behavior of the system right after the transition has been taken.

The execution operations for the *charts model of computation are detailed below. The algorithm requires three global variables and one constant. The constant, represented by the `initMode` attribute of the model, is used to save the initial mode of the system. The three global variables are attributes of the model of computation. The `currentMode` variable holds the current mode of the system, the `nextMode` variable is used to store the future mode of the system and the `hist` variable is used to remember if the system has to begin with the initial mode or with the current mode. Below, we only detail the sub-operations of the `compute` operation. We describe briefly the role of the others here. The `setup` operation has to check if the finite state machine is deterministic. The `startOfSnapshot` operation initializes the variable representing the current mode. If `hist` is false, that means that the model has never been executed and `currentMode` is initialized with the value of `initMode`. Else, it is initialized with the value of `nextMode` (which has been determined in a previous snapshot). The `validate` operation is quite simple to implement since only one block can be active at a time. Thus, only the active one can validate the computation. Last, in the `endOfSnapshot` operation, the value of `hist` is forced to false since the model may not be in the initial mode in the next snapshot and the model pins are made observable. In the following algorithms, pins have a "known" attribute that indicates if the value of the token of the pin has been determined in the current snapshot or not. We also use a special Expression type with an `eval` operation that evaluates any expression in the context of a model.

```
reset()
  // Reset outputs (blocks and model)
  foreach Pin p in self.model.pins such that p.direction = OUT and
  p.type = DATA do
  |   p.known ← FALSE;
  endfch
  foreach Block b in self.model.blocks do
  |   foreach Pin p in b.pins do
  |   |   p.known ← FALSE;
  |   endfch
  endfch
```

```
preSchedule()
  // Take enabled preemptive transitions
  foreach Pin p in self.currentMode.pins such that p.type = CTRL do
  |   foreach Relation r in p.relations do
  |   |   foreach Block t in r.target.owningBlock such that
  |   |   t.preemptive = TRUE and Expression.eval(t.guard,
  |   |   self.model) = TRUE do
  |   |   |   self.currentMode ← r.target.owningBlock;
  |   |   endfch
  |   endfch
  endfch
```

```
modelToBlocks()
  // Propagate data from model to current mode
  foreach Pin pmodel in self.model.pins such that pmodel.direction
  = IN and pmodel.type = DATA do
  |   foreach Relation r in pmodel.relations such that
  |   r.target.owningBlock = currentMode do
  |   |   foreach Pin pblock in r.target.owningBlock.pins such that
  |   |   pblock.direction = IN and pblock.type = DATA do
  |   |   |   pblock.setToken(pmodel.readToken());
  |   |   endfch
  |   endfch
  endfch
```

```
update()
  // Update on current mode
  self.currentMode.update();
```

```
inSchedule()
  // No purpose for this MoC
```

```
propagate()
  // No purpose for this MoC
```

```
postSchedule()
  // Take enabled transitions
  self.nextMode ← self.currentMode;
  foreach Pin p in self.currentMode.pins such that p.type = CTRL do
  |   foreach Relation r in p.relations do
  |   |   foreach Block t in r.target.owningBlock such that
  |   |   Expression.eval(t.guard, self.model) = TRUE do
  |   |   |   self.nextMode ← r.target.owningBlock;
  |   |   endfch
  |   endfch
  endfch
```

```
blocksToModel()
  // Propagate data from current mode to model
  foreach Pin pblock in self.currentMode.pins such that
  pblock.direction = OUT and pblock.type = DATA do
  |   foreach Relation r in pblock.relations do
  |   |   foreach Pin pmodel in r.target.owningBlock.pins such
  |   |   that pmodel.direction = OUT and pmodel.type = DATA
  |   |   do
  |   |   |   pmodel.setToken(pblock.readToken());
  |   |   endfch
  |   endfch
  endfch
```

```
done() : Boolean
  // Check if model outputs are known
  Boolean done ← TRUE;
  foreach Pin p in self.model.pins such that p.direction = OUT and
  p.type = DATA do
  |   if p.known = FALSE then
  |   |   done ← FALSE;
  |   endif
  endfch
  return done;
```

7. Conclusion

We have presented a generic and modular approach for heterogeneous and executable modeling of systems. Our previous experience shows that it is particularly useful for embedded systems where problems are related not only to the design process but also to the very nature of these systems that must be reliable, concurrent and reactive, to cite only a few properties. Our approach includes a language for describing models of computation and a generic execution platform that computes the behavior of heterogeneous hierarchical models according to the description of their models of computation.

Currently, we are improving our execution model and verifying its ability to handle any model of computation. We are defining the semantics of our language and the execution operations precisely, the next step being a mathematical formalization in a framework that has not been chosen yet. One of our goals is to be able to integrate our platform with analysis and model-checking tools.

The concrete syntax of our language is not chosen yet, but we will probably use an existing language such as OCL — which allows to navigate models easily and to work efficiently with sets — and extend it with the primitive operations of our execution model. We are also studying how to specify with precision what happens at the interface between heterogeneous models. For example, when a discrete time model and a continuous time model exchange data, we must tell how continuous signals are sampled and how discrete samples are integrated into continuous signals. Another important work in progress relates to the concept of "Time". It is a key notion in the domain of embedded systems, in particular for real time systems. In our framework, the only built-in notion of time is the sequence of snapshots that constitute the observation of a system. Each model of computation can define its own notion of time by associating a date to each snapshot (and eventually duration between dates). We are working on the definition of relations between the times of different models of computation based on concepts that should be available in the MARTE profile [1].

We foresee numerous extensions to our approach. For example, since it is possible to associate several models to the same block, we could interpret these models as different facets of the system, just like in Rosetta. This way, a block could not only have a behavioral model but also an energy consumption model, a mechanical model, etc. Another perspective is to integrate model refinement techniques into our approach in order to execute under-specified models, for instance to validate architecture choices. We are considering the use of abstract data types, as well as symbolic execution techniques which allow to determine the tree of all possible executions of a model. Eventually, our language will come with a set of transformations allowing to turn a model described in a given formalism (with a known meta-model) into a model described in our language that will be executable by the platform.

References

[1] UML profile for modeling and analysis of real-time and embedded systems (MARTE) RFP, january 2005. http://www.omg.org/cgi-bin/doc?realtime/2005-2-6.

[2] F. Boulanger, M. Mbobi, and M. Feredj. An approach for domain polymorph component design. In *IEEE International Conference on Information Reuse and Integration 2004 (IRI 2004)*, 2004.

[3] F. Boulanger, M. Mbobi, and M. Feredj. Flat heterogeneous modeling. In *IPSI 2004 conference*, 2004.

[4] C. Brooks, E. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java, (Volume 1, introduction to Ptolemy II). Technical Report 21, EECS Dept., UC Berkeley, July 2005.

[5] J. Burch, R. Passerone, and A. L. Sangiovanni-Vincentelli. Overcoming heterophobia: Modeling concurrency in heterogeneous systems. *acsd*, 00:13, 2001.

[6] T. Clark, A. Evans, P. Sammut, and J. Willans. *Applied metamodelling: A foundation for language driven development*. Xactium, september 1978.

[7] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. In *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, volume 91, pages 127–144, January 2003.

[8] C. Kong and P. Alexander. The Rosetta meta-model framework. In *Proceedings of the IEEE Engineering of Computer-Based Systems Symposium and Workshop (ECBS'03)*, april 2003.

[9] B. Lee. Specification and design of reactive systems. Technical Report UCB/ERL M00/29, EECS Department, University of California, Berkeley, 2000.

[10] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, volume 75, september 1987.

[11] E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[12] L. Muliadi. Discrete event modeling in Ptolemy II. Technical Report UCB/ERL M99/29, EECS Department, University of California, Berkeley, may 1999.

[13] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML 2005*, 2005.

[14] G. Tel. *Introduction to Distributed Algorithms, 2nd edition*. Cambridge University Press, 2000.

[15] J. C. Willems. Models for dynamics. *Dynamics Reported*, vol.2, 1989.