Simulation of Multi-Formalism Models with ModHel'X

Frédéric Boulanger Cécile Hardebolle
SUPELEC – Département Informatique
frederic.boulanger@supelec.fr *cecile.hardebolle@supelec.fr*

Abstract

A step in the validation of a system is to check its behavior by simulation. Simulation is also used to validate the behavior of the model of the system against test patterns. For complex systems, models are made of parts which use different modeling formalisms. The main issues in the simulation of such systems are the specification of the semantics of each modeling formalism, and of the interactions between heterogeneous parts of a model. ModHel'X relies on component based modeling and models of computation to address these problems and focuses on the computation of one possible behavior of a model. This includes simulation and code generation. ModHel'X defines a MOF meta-model for describing the structural elements of a modeling language. The semantics of modeling languages is expressed in an imperative style and addresses three aspects: control, data and time. ModHel'X is supported by a simulator of multi-formalism models.

1. Introduction

Complex systems are inherently heterogeneous because of the diverse nature of their numerous parts: hardware, software, digital, analog, internal or external IPs (Intellectual Properties), etc. Modeling such systems requires multiple modeling formalisms, adapted to the nature of each part of the system, the aspect on which the model focuses (functionality, time, power consumption…) and to the level of abstraction at which the system, or one of its parts, is studied. As emphasized by [1], having a global model of such a system all along the design process is necessary in order to answer questions about properties of the whole system, and in particular about its behavior. Each formalism can be based on a specific paradigm — process networks, discrete events, automata, continuous time, differential equations, etc. [2] — therefore such a model is said to be *multi-formalism* [3].

This work has been performed in the context of the Usine Logicielle project (http://www.usine-logicielle.org) and is partially financed by the System@tic Paris-Région Competitiveness Cluster (http://www.systematic-paris-region.org).

Multi-formalism modeling (or heterogeneous modeling) is an emerging field and different aspects of it have been studied: mathematical foundations [2], tools for validation [4] or simulation [5]. A central problem is to establish the meaning of the composition of heterogeneous parts of a model and to ensure their correct inter-operation when using the model to answer questions about the designed system [6].

We believe that the first step to be taken for solving this problem is to provide means for the precise specification of the semantics of the modeling formalisms that we want to use. Indeed, except for a few mathematically founded languages, the semantics of a modeling language is often described using natural language, what may lead to ambiguities and to diverse interpretations by different tools along the design chain. When combining different modeling languages in a model, ambiguities in the semantics of one of them make it impossible to define the overall semantics of the model. In this context, semantic variations as found in UML are acceptable only if the variation used is explicitly stated. In ModHel'X, we propose a set of tools for allowing the precise specification of the semantics of a modeling formalism without referring to any model instance (i.e. at the meta-modeling level [3]). In order to facilitate the combination of multiple modeling languages in models, our approach relies on concepts of component-oriented and hierarchical modeling [7]. The encapsulation principle in component-oriented modeling is a major advantage for heterogeneous modeling since its very purpose is to hide the internal mechanisms of the components. In this context, hierarchy is a structural way of combining the heterogeneous parts of a model, as well as a simple abstraction mechanism.

The second step to obtain a meaningful multi-formalism model of a system is to provide support for the specification of the semantic adaptation between model parts that use different modeling formalisms. An important constraint is that no model part should be modified to become compatible with the other parts of the multi-formalism model. This is particularly important when the model parts come from different technical teams or from suppliers for instance. In ModHel'X, the adaptation mechanism is decoupled from the model parts which are being integrated. A second issue is that the semantic adjustment between heterogeneous parts of a model depends not only on the formalisms at stake but also on the system which is modeled. Usual adaptation patterns between modeling formalisms often exist, but they are not unique and may need parameter adjustements. Such patterns may represent default adaptations which may not fit directly a particular context. For example, when integrating model parts that have different notions of time, it may be necessary to customize the way the different times are synchronized in order to have a coherent behavior of the system. ModHel'X permits the description of adaptation patterns and allows the designer to choose the most suitable one in a given model. These descriptions may be reused in different contexts, and parameters allow their adaptation to specific applications.

Finally, we have developed for ModHel'X an execution engine which is able to interpret multi-formalism models and to simulate their behavior. This execution engine is deterministic, so if every modeling formalism used in a model is deterministic, the whole simulation is deterministic — the same input sequence will always produce the same simulation result. This allows ModHel'X to be used to test heterogeneous models. It is also possible to rely on the same precise definition of the semantics of the modeling formalisms and of their interactions to generate implementations that behave the same as the model.

The remainder of the paper is organized as follows. In Section 2 we review some of the related work and motivate our approach. Section 3 details and illustrates the main principles of ModHel'X. We discuss some specific aspects of our approach in Section 4, before concluding.

2. Other multi-formalism approaches

In meta-modeling approaches such as Kermeta [8], the abstract syntax of a modeling language is described as a MOF meta-model. The elements of this meta-model have methods whose semantics is defined in an imperative language. Each modeling language has a different meta-model in Kermeta. In the context of heterogeneous modeling, the definition of the combination of several modeling languages using such approaches implies either the definition of a meta-model which is the union of all the meta-models of the involved languages, or the definition of transformations from each meta-model to a meta-model chosen among them. Defining a union meta-model seems neither reasonable nor scalable since it implies modification of the meta-model and the associated model transformations when an additional modeling language is taken into consideration. The second method is much more interesting since it is more flexible: the target meta-model can be chosen according to the question to be answered about the system. Such an approach is implemented in the $ATOM^3$ tool [9]. However, the way the different heterogeneous parts of the model are "glued" together does not seem to be addressed by this approach.

Other approaches [10, 11] are also based on model transformation. In particular, [11] states that it is possible to formally define the semantics of a modeling language by defining a mapping to an already existing formally defined modeling language.

Another approach for defining the semantics of a modeling language, is to define the constructs of the language in a fixed abstract syntax — or meta-model — which is component oriented (as in [7]) and to consider that the semantics of a modeling language is given by its "Model of Computation (MoC)". Such an approach is implemented in Ptolemy [1]. A model of computation (called "domain" in Ptolemy) is a set of rules for interpreting the relations between the components of a model. In this approach, the meta-model is the same for each language, and what defines the semantics of the language is the way the elements of this meta-model are interpreted by the corresponding MoC. Heterogeneous models are organized into hierarchical layers, each one involving only one MoC. Thanks to this architecture, MoCs (i.e. modeling languages) are combined in pairs at the boundary between two hierarchical levels. The main drawback of the Ptolemy approach is that the way MoCs are combined at a boundary between two hierarchical levels is fixed and coded into the Ptolemy kernel. This implies that a modeler has either to rely on the default adaptation performed by the tool, or to modify the design of parts of its model (by adding adaptation components) in order to obtain the behavior he expects.

Let us consider, for example, the model of a system which computes routes for a car according to known traffic conditions. This system is composed of an algorithm which computes a route from the destination and the current position of the car, and of a subsystem which retrieves traffic information from a network. The traffic information, when available, is used by the routing algorithm to minimize the duration of the trip. The routing algorithm regularly receives the position of the car and updates the route. A synchronous data-flow formalism (SDF Ptolemy domain) is particularly adapted for modeling such signal processing systems. The traffic information retrieving system is provided by a supplier, who used a discrete events formalism (DE Ptolemy domain) in order to model the response delay of the network. Embedding the DE model directly into the SDF model is not possible because SDF requires an immediate response to inputs, while the traffic information retrieving system will produce data only when the network answers its request. In Ptolemy II, the modeler will have to modify the DE model by adding a sampler component which will deliver data synchronously by repeating the previous data sample when no new data is available. The problems that arise with this approach are the following:

- The altered model of the information retrieving system does not represent any longer the behavior of the component which will be delivered by the supplier. This may lead to implementation incoherences at the end of the development cycle.
- Since semantic adaptation is done by the modeler in the model of the component, it is not protected from changes made by the supplier to the model.
- If the formalism used in one of the models changes — e.g. to refine the model in order to take finer details into account — the adaptation must be expresed again in the new formalism. This is incompatible with modularity and reuse.

It is therefore important to allow the designer to specify the semantic adaptation outside the models of the parts he assembles.

The approach we propose is based on the concept of *model of computation (MoC)* as defined in [1]. Our MOF meta-model, which is inspired by the abstract syntax of Ptolemy, contains special constructs for making the interactions between heterogeneous MoCs explicit and easy to define. In order to interpret a model in ModHel'X, it is necessary to describe its structure using our meta-model. Then, we define an interpretation of the elements of our meta-model which matches the semantics of the original language. Such an interpretation is what we call a Model of Computation. The interpretation of a model according to a MoC gives the same behavior as the interpretation of the original model according to the semantics of its modeling language. The same concepts used to define MoCs are used to define how different MoCs are "glued" together in heterogeneous models, at the boundary between two hierarchical layers. The execution engine of ModHel'X relies on the precise specification of the models of computation and of their interactions to determine without ambiguity the behavior of multi-formalism models.

Other approaches of heterogeneous modeling are also based on a hierarchical and component oriented abstract syntax. BIP (Behavior, Interaction, Priority) [4] provides formally defined mechanisms for describing combinations of components in a model using heterogeneous interactions. BIP does not consider components as black boxes and has access to the description of their behavior. This allows the formal verification of properties on the model. It is important to note that, in BIP, the description of the interactions between components is made at the M1 level.

The "42" approach [12] seems closer to ours. Based on the synchronous paradigm, 42 generates the code of the MoCs (called "controllers") from the contracts of the components (described using automata), the relations between their ports and additional information related to activation scheduling. The strength of this approach relies on the description of the behavioral contract of components. However, such a description may not be available (in the case of an external IP for instance) or may not be easy to establish, in the case of continuous time behaviors for example.

Metropolis [13] also relies on the concept of model of computation, but it focuses on MoCs related to process networks. It originates from trace algebras [14] and is closely related to the tag semantics approaches [2, 15], which provide mathematical frameworks for the formalization of MoCs and their interactions but are very far from model execution.

3. Modeling heterogeneous systems with ModHel'X

3.1. Black boxes and snapshots

We adopt a component-oriented approach in which we consider components as black boxes, called *blocks*, in order to decouple the internal model of a component from the model of the system in which it is used. Therefore, the behavior of a block is observable only at its interface: nothing is known about what is happening inside the block, and in particular whether the block is even computing something.

In addition, instead of "triggering" the behavior of a block, we only *observe* its interface. When we need to observe a block, we ask it to provide us with a coherent view of its interface at this moment. A block can therefore be active even when we do not observe it. This is a key point in our approach because it allows us to embed asynchronous processes in a model without synchronizing their activity: we simply observe them at instants suitable for the embedding model, and the embedded model provides us with views of its interface at these instants. The behavior of a block or a model is therefore a sequence of observations, without consideration for the internal behavior which produces these observations. An observation of a model is defined as the combination of the observations of its blocks according to a MoC. This definition holds at all the levels of a hierarchical model. The observation of the top-level model, i.e. the model of the overall system, is a *snapshot* [16] which defines the exact state of the interface of each block at a given instant (such a notion is also defined in the context of UML [17]). We detail the way a snapshot is obtained using the rules expressed by a MoC in Section 3.4.

3.2. Time

The notions of time used in different models of computation are varied (real time, logical clocks, partial order on signal samples, etc.), and ModHel'X must support all of them. Moreover, in an heterogeneous model, different notions of time are combined and each part of the model may have its own time stamp in a given snapshot. Therefore, the succession of snapshots is the only notion of time which is shared by all MoCs and which is predefined in ModHel'X. On this sequence of instants, each MoC can define its own notion of time.

A snapshot of a model is made whenever its environment (i.e. the input data) changes, but also as soon as any block at any level of the hierarchy needs to be observed because its state has changed. To this end, each component of an heterogeneous model can give constraints on its time stamp at the next snapshot. For instance, in a timed automaton, a time out transition leaving the current state must be fired even if no input is available. This can be achieved by requiring, when entering this state, that the next snapshot occurs before the timeout expires. This feature is a major departure from the Ptolemy approach, where the root model drives the execution of the other layers of the hierarchy.

Times in two MoCs may be synchronized by the interaction pattern at the boundary of two hierarchical levels. Thus, time constraints can propagate through the hierarchy up to the top level model.

3.3. A generic meta-model for representing the structure of models

The generic meta-model that we propose, shown on Figure 1, defines abstract concepts for representing the structural elements of models. Each of these concepts can be specialized in order to represent notions that are specific to a given modeling language, but their semantics is given by the MoCs which interpret them.

In the *structure* of a *model*, *blocks* are the basic units of behavior. *Pins* define the interface of models and blocks. The interactions between blocks are represented by *relations* between their pins. Relations are unidirectional and do not have any behavior: they are interpreted according to the MoC in order to determine how to combine the behaviors of the blocks they connect. For instance, a relation can represent a causal order between two blocks as well as a communication channel.

In Modhel'X, data is represented by *tokens*. The concept of token can be specialized for each model of computation. For instance, in a discrete event model, tokens may have a value and a time stamp, while in a data-flow model, they carry a value only. The type of the value which is carried by a token is not taken into account by the MoC, which is only in charge of delivering the tokens by interpreting the relations between the blocks.

The behavior of a block can be described either using a formalism which is external to our framework (for instance in C or Java), yielding an *atomic block*, or by a ModHel'X model. To handle the latter case, we have introduced a special type of block called an *interface block*, which implements hierarchical heterogeneity: the internal model of an interface block may obey a MoC which is different from the MoC of the external model in which the block is used. Interface blocks are a key notion in our framework since they are in charge of adapting the semantics of their inner and outer models of computation. They allow the explicit specification of the interactions between different MoCs.

3.4. An imperative semantics for MoCs and their interactions

Computing a snapshot of an heterogeneous model requires to compute the observation of all its parts, which may use different MoCs i.e. different notions of time, control or data. The issue of the consistency of such an observation is similar to the definition of the state of a distributed system [16]. In ModHel'X, we have chosen to define a model of computation as an algorithm for computing observations of the model to which it is associated. For each observation, the algorithm asks the blocks of the model to *update* the state of their interface. The results of the update (output data) are propagated to other blocks by *propagation* operations. We want our execution engine to be deterministic, therefore we observe the blocks sequentially. To ensure the consistency of the computed behavior with the control and concurrency notions of the original model, the MoC must include *scheduling* operations which determine the order in which to update the blocks.

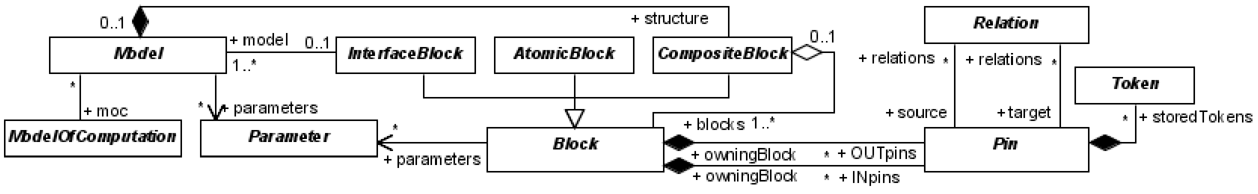

Figure 1. Generic meta-model for representing the structure of models

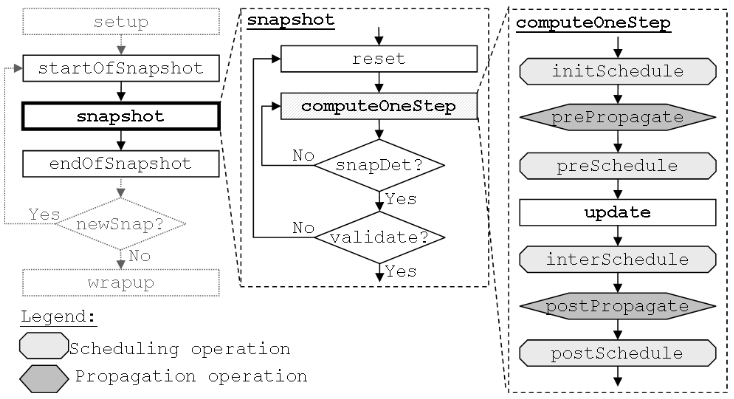

Figure 2. Generic execution algorithm

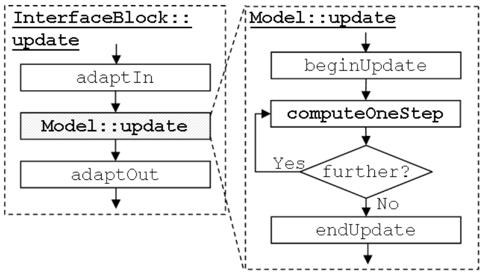

Figure 3. Update on an interface block and its internal model

Figure 2 represents the generic structure of our algorithm. This structure is a fixed frame which "standardizes" the way MoCs can be expressed in ModHel'X, but the contents of each element is left free. Therefore, for each MoC, the semantics of the operations of this algorithm has to be described, using an imperative syntax, in order to define the scheduling and propagation "policies" specific to the MoC (non necessary operations can be left empty). The left part of the figure shows the loop which computes the succession of snapshots of the execution of the model. In the computation of a snapshot, the observation of one block brings into play the scheduling and propagation operations mentioned above and is called a *step* (represented on the right part of Figure 2 under the name `computeOneStep`). The algorithm loops on successive steps until the snapshot is entirely determined (i.e., for most MoCs, when the state of all the outputs of the executed model is known). A given block may be updated several times in this loop, what allows the use of non-strict [18] blocks for the computation of fixed point behaviors. Therefore, ModHel'X supports MoCs in which cyclic dependencies are allowed.

The basic sequence for performing a computation step is to choose a component according to the state of the model and the available inputs (`init-Schedule`), propagate input data to this component (`prePropagate`), then choose a component to observe (`preSchedule`), ask it to update its interface (`update`), choose a component according to the state of the model and the data produced during the update (`interSchedule`), propagate the data according to the chosen component (`postPropagate`), and finally, chose a component according to the data which has just been propagated (`postSchedule`). This sequence is built so that a component may be scheduled as soon as something new happens in the model (new inputs, new outputs, propagation of data), and the propagation of data may depend on which component is scheduled.

The scheduling operations of a model of computation are responsible for ensuring the causality of the observations. They are used both for choosing the component to which data will be routed and for choosing the component which will be observed next. The order in which the components of a model are observed may influence the result of the observation. For instance, if the outputs of component B depend on the outputs of component A, but B is observed before A, B won't be able to take the outputs of A into account and won't produce the same outputs as if it were observed after A. Scheduling operations are therefore among the most important operations in the description of a model of computation. When implementing models of computation where components run concurrently, the scheduling operations model the nature of the concurrency and the synchronization mechanisms of the model of computation. Since the execution engine invokes the operations sequentially, the computation of a snapshot is deterministic. However, it is always possible to call non-deterministic functions like `random` in the scheduling operations in order to model non-deterministic models of computation. Such MoCs may be useful for simulating a system, but their use diminishes the value of tests since the same test pattern may succeed or fail depending on non deterministic choices during the simulation.

The execution of a model traverses the hierarchy thanks to the delegation of the operations of interface blocks to their internal model. Snapshots are realized only at the top level, which represents the whole system. An internal model is only asked to provide a coherent view of its behavior when its interface block is updated. The `update` operations of interface blocks and models are shown on Figure 3. The `adaptIn` and `adaptOut` operations of an interface block allow the modeler to specify explicitly how the semantics of the internal and the external MoCs are adapted before and after the update of its internal model, i.e. to specify the meaning he gives to the joint use of two models of computation. In `adaptIn`, data from the embedding model is interpreted and translated into the formalism of the embedded model. This may include more than changing the representation of data. For instance, if the internal model expects that two of its inputs are always available simultaneously, `adaptIn` may store the first occurrence of one of these inputs and wait for the second before delivering the two inputs to the embedded model. On the contrary, if two inputs are exclusive, `adaptIn` may be used to deliver them to the embedded model in two separate snapshots if they happen simultaneously in the embedding model. `adaptIn` can therefore change the data that is passed to the embedded model, but can also change control, i.e. when the embedded model will be able to react to new data. The last point which is controlled by `adaptIn` is time. Since each model of computation may have its own notion of time, `adaptIn` can be used to compute the time stamp of the current snapshot for the embedded model of computation from the time stamp of the embedding model of computation, from the time stamps of the input data, or from any other suitable parameter.

After the input data has been adapted, the internal model of the interface block must be updated. The `beginUpdate` operation is used to take new adapted inputs from the interface block into account, and the `endUpdate` operation is used to provide outputs determined during the update of the model to the interface block. The observation of a model may be partial (if it models a non-strict component). The loop which computes the observation must stop when the `further` operation indicates that no more outputs can be determined according to the current state of the model and the currently available inputs.

Then, the `adaptOut` operation interprets the data produced by the embedded model and translates it so that it is meaningful to the embedding model. The same kinds of transformations as used in `adaptIn` may be performed here: change of representation, computation of time stamps, holding data until a later snapshot, delivering default or previously produced data and so on.

3.5. Implementation and validation

We have experimented our approach in a prototype of ModHel'X based on the Eclipse EMF framework [19]. We use the ImperativeOCL [20] language, an imperative extension of OCL, for describing the semantics of the operations of our algorithm. No interpreter being available for the moment, we translate it into Java. We have successfully implemented several MoCs, such as Finite State Machines (FSM), Discrete Events (DE) and *charts [21]. We are developing a library of MoCs in order to further the validation of our approach. In particular, we are currently working on the UML Statecharts and the Synchronous Dataflow (SDF) MoCs.

3.6. Example multi-formalism model

To illustrate our approach, and in particular the semantic adaptation between a timed and an untimed MoC, we consider a simple hierarchical and heterogeneous model of a coffee machine which works as follows: first the user inserts a coin, then he presses the "coffee" button to get his coffee after some preparation time.

In this model, we take into account the date of the interactions between the user and the machine: insert a coin, push a button, deliver coffee. Therefore, we use the Discrete Events (DE) MoC, which is implemented by SimEvents (The MathWorks), VHDL or Verilog for instance. We represent our user by an atomic block, whose behavior is written in Java. We model the coffee machine as an automaton (with UML Statecharts for instance), because at this stage of the design process, we focus on the logic of its behavior. We consider here a simple version of this MoC called FSM (Finite State Machines), which is similar to the one presented in [8]. Figure 4 shows the global model resulting from the combination of the DE and FSM models. Such a combination is a classical example, which is well addressed by tools like Ptolemy. However, we will see that it is pos-

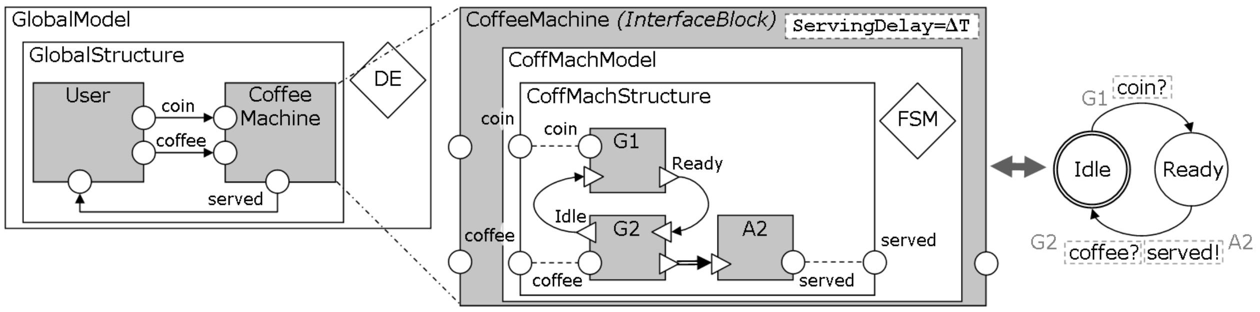

Figure 4. Global model of the coffee machine and coffee machine automaton

```
// Search for blocks with a constraint at the current time
OrderedSet(Block) blocklist := self.constraints
  →select(c:Constraint|c.constraintTime=self.currentTime)
  →collect(c:Constraint|c.author);
// If blocks have constraints at the current time...
if ( blocklist →notEmpty()) {
  // Topological sort on these blocks
  self.topologicalSort( blocklist , m.structure);
  // Choose the first one to update
  self.currentBlock := blocklist →first();
  // Then remove the corresponding constraint
  self.constraints := self.constraints→reject(b:Block|b=self.currentBlock);
} else {
  // else, search for blocks that have to receive events
  blocklist := self.activeEventList→collect(e:Event|e.destinationPin.isInputForBlock)
  // If there are blocks to update
  if ( blocklist →notEmpty()) {
    // Topological sort on these blocks
    self.topologicalSort( blocklist , m.structure);
    // And choose the first one to update
    self.currentBlock := blocklist →first();
  }
}
```

Figure 5. `initSchedule` operation in DE

```
// Check all the output pins of the internal model
self.model.structure.pinsOut
  →select(pInt:Pin|pInt.storedTokens→notEmpty())
  →forEach(pInt:Pin) {
    // If FSM events have been produced by the internal model...
    self.pinsOut→forEach(pExt:Pin) {
      pExt.storedTokens→append(
        // they become DE events on the outputs of the block...
        new DEEvent(
          // with time stamps = the last stored time stamp...
          self.tLastDEevt
          // plus the serving delay
            + self.parameters
              →select(param:Parameter|param.name="servingDelay")
        )
      );
    }
    // FSM events are cleared
    pInt.storedTokens→clear();
  }
```

Figure 6. Coffee machine `adaptOut`

sible to handle the interactions between DE and FSM differently with ModHel'X.

The representation of the structure of the DE model in ModHel'X is straightforward. The representation of the FSM model is more involved because a transition may have two associated behaviors: the evaluation of its guard and its action. Since blocks are the basic units of behavior in ModHel'X, a transition is represented using a block for its guard, linked by an *action* relation to a block that performs its action, and by *next* relations to the transitions that become enabled when this transition is taken (these are the transitions that leave the target state of the transition). Therefore, *next* relations between guards represent the states of the automaton.

In DE, when a snapshot is taken, the current time is determined according to the time stamps of the input events and on the time constraints that have already been produced by the blocks. At each computation step, we consider the blocks which have posted a time constraint for the current time and the blocks which are the target of events at this time. The semantics of DE assumes that a given block is observed only once in a snapshot, so any block in DE is guaranteed to have all its input events available when it is updated at a given time stamp. However, a block is allowed to react instantaneously to its inputs, and to produce an event with a time stamp equal to the current time. We must therefore update the blocks of a DE model in such a way that if block B depends on some outputs from A, A must be updated before B in case it produces an event for B at the current time. In our implementation of DE, we always chose to update a block which is minimal according to a topological sort of the blocks of a model. Figure 5 shows the code of the `initSchedule` operation for DE, which is the only scheduling operation for this MoC, the others (`preSchedule`, `interSchedule` and postSchedule) being left empty. initSchedule uses findARoot to find a minimal block according to the partial order induced by the dependency relations between the blocks of the model.

DE and FSM share the notion of event. However, FSM has no notion of time attached to events, and therefore no notion of duration between events. So, when a DE event enters FSM, the interface block has to remove its time stamp to make it look like an FSM event. This is the role of the adaptIn operation of the interface block. When an FSM event leaves the embedded model to enter DE, the interface block has to give it the "right" time stamp during the adaptOut operation. An acceptable way to proceed is to give it the same time stamp as the most recent incoming event (in particular, this is what is done by Ptolemy). We provide an interaction pattern which realizes this adaptation. However, for our coffee machine, this behavior does not model the serving delay, which is an important characteristic of the model which represents the time taken by the internal process of heating water, mixing it with the coffee powder, and pouring it into the cup. This process could be modeled with more details, but here, we just keep an abstract view of it as a serving delay. Therefore, we add a ServingDelay parameter to the coffee machine and we modify the pattern so that the time stamp of the served event is the time stamp of the coffee event plus the ServingDelay. This behavior is implemented in the adaptOut operation, as shown on figure 6. What is important in ModHel'X is that the interaction pattern between two models of computation is part of the model, not of the platform. It is therefore possible to define or reuse the most suitable interaction pattern for each model.

4. Discussion

4.1. Required effort for using ModHel'X

There are two prerequisites to the use of the ModHel'X framework. First, an expert of a modeling language has to describe the structural and semantic elements of this language using our meta-model and our imperative syntax. Since our goal is not to replace existing modeling tools, this expert also defines transformations from the original meta-model of the language to our generic meta-model. This is the difficult part of the work because the semantics of modeling tools is often known intuitively, through the experience we have of the tools. Second, for each pair of MoCs that may interact in heterogeneous models, experts should define interaction patterns, which code standard ways of combining models that obey these MoCs. These steps represent the main effort needed to benefit from the ModHel'X approach. However, they are done once and for all for each modeling language. Then, system designers can assemble heterogeneous models of the parts of the system, and use the interaction patterns to specify how the models are glued together. Parameters of the patterns allow the designers to fine tune the semantic adaptation at the boundary of two models of computation. If no suitable interaction pattern exist yet for a given model, the designer can define his own, or ask an expert to do so. For now, interaction patterns can only be defined as templates for the adaptIn and adaptOut operations of interface blocks. We would like to allow the definition of interaction patterns as ModHel'X models (using blocks and relations), but this requires either the use of blocks with ports that obey different MoCs, or the use of relations between ports that obey different MoCs. Thanks to previous work on flat heterogeneous modeling [22], this seems to be possible, and we plan it as future work when we have used ModHel'X on a larger library of MoCs and interaction patterns.

4.2. Supported models of computation

Considering that a given structure of model can be interpreted as an automaton or as a discrete event model depending on the MoC which is associated to it can seem somewhat extreme. However, this choice has proven to be powerful since a tool like Ptolemy supports, on this basis, paradigms as different as finite state machines, ordinary differential equations or process networks.

In the same way, ModHel'X can support a wide range of models of computation. This includes MoCs for continuous behaviors, which are approximated by the computation of a series of discrete observations since we only address the digital execution of models.

Modal models are also supported. In such models, the behavior of a component may be computed by different models according to the state of the component. The state changes are modeled by a state machine whose transitions are fired by conditions on the inputs or outputs of the model. The problem with modal models is that the triggering conditions are interpreted in the "state machine" model of computation, but the signals they depend on are computed by the underlying model of computation in the currently active model. Using interface blocks, this problem is easily solved in ModHel'X, without coding any explicit support for modal models in its kernel.

ModHel'X also supports models of computation that allow cyclic dependencies in models. Such dependencies are solved by iterating toward a fixed point, as in the Synchronous Reactive domain of Ptolemy. The fixed point is reached only if all blocks are monotonous according to a partial order defined by the model of computation.

4.3. Comparing ModHel'X and Ptolemy

Ptolemy was our main source of inspiration, but we have extended it on several aspects. One of our main contributions is the explicit specification of the interactions between MoCs in the models (see Section 3.4). Moreover, our approach is based on the observation of blocks and not on the triggering of actors. Thanks to this change of paradigm and to the introduction of time constraints, the execution of a ModHel'X model is not necessarily driven by its root level. Indeed, a block at any level of the hierarchy of the model can produce a constraint on the time stamp of its next observation, what will force the execution machine to compute a snapshot at this time, even if no new input is available for the model. Finally, the definition of our abstract syntax as a MOF meta-model allows us to rely on model transformation tools from the MDE community to exchange models with other tools in the design chain.

5. Conclusion

We have presented an approach to multi-formalism modeling which provides support for the specification of the semantics of a modeling formalism through the concept of model of computation, and which allows the definition of the interactions between heterogeneous parts of a model through a special modeling construct and using an imperative syntax. This approach relies on the black-box and the snapshot paradigms to compute the observable behavior of a model by combining the behaviors observed at the interface of its components. A generic MOF meta-model for representing the structure of hierarchical heterogeneous models has been proposed. On this basis, models of computation are described by giving a specific semantics to the primitive operations of a generic algorithm which computes snapshots of models conforming to the proposed meta-model. The result is a simulator which can compute the behavior of heterogeneous models in response to stimulation scripts in a deterministic way.

We are currently developing the MoC library of our prototype in order to further the validation of our approach. The rigid structure of the execution algorithm of ModHel'X is a first step toward the definition of MoCs in a fixed frame with formal semantics. However, for the moment, our imperative syntax is still too close to Java to have a formal semantics. For now, ModHel'X can therefore be used only for testing, not for model-checking or demonstrating properties. Future work will also address the verbosity of the description of the models of computation by defining higher level constructs over our current imperative syntax.

References

[1] 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," *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, vol. 91, pp. 127–144, January 2003.

[2] E. A. Lee and A. L. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, 1998.

[3] P. J. Mosterman and H. Vangheluwe, "Computer automated multi-paradigm modeling: An introduction," *Simulation: Transactions of the Society for Modeling and Simulation International*, vol. 80, no. 9, pp. 433–450, 2004. Special Issue: Grand Challenges for Modeling and Simulation.

[4] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time systems in BIP," in *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM06)*, pp. 3–12, september 2006.

[5] P. Fritzson and V. Engelson, "Modelica — A unified object-oriented language for system modeling and simulation," in *European Conference on Object-Oriented Programming (ECOOP98)*, pp. 67–90, july 1998.

[6] T. A. Henzinger and J. Sifakis, "The embedded systems design challenge," in *Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science*, pp. 1–15, Springer, August 2006.

[7] E. Bruneton, T. Coupaye, and J. Stefani, "The fractal component model specification," February 2004.

[8] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, "Weaving executability into object-oriented meta-languages," in *Proceedings of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS/UML 2005)*, pp. 264–278, 2005.

[9] J. de Lara and H. Vangheluwe, “*$ATOM^3$*: A tool for multi-formalism modelling and meta-modelling,” in *5th Fundamental Approaches to Software Engineering International Conference (FASE 2002)*, pp. 595–603, april 2002.

[10] T. Levendovszky, L. Lengyel, and H. Charaf, “Software Composition with a Multipurpose Modeling and Model Transformation Framework,” in *IASTED on SE*, (Innsbruck, Austria), pp. 590–594, February 2004.

[11] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle, “On the use of graph transformations for the formal specification of model interpreters,” *Journal of Universal Computer Science, Special issue on Formal Specification of CBS*, vol. 9, no. 11, pp. 1296–1321, 2003.

[12] F. Maraninchi and T. Bouhadiba, “42: Programmable models of computation for a component-based approach to heterogeneous embedded systems,” in *6th ACM International Conference on Generative Programming and Component Engineering (GPCE’07)*, pp. 1–3, october 2007.

[13] F. Balarin, L. Lavagno, C. Passerone, A. L. S. Vincentelli, M. Sgroi, and Y. Watanabe, “Modeling and designing heterogeneous systems,” *Advances in Concurrency and System Design*, 2002.

[14] J. R. Burch, R. Passerone, and A. L. Sangiovanni-Vincentelli, “Overcoming heterophobia: Modeling concurrency in heterogeneous systems,” in *Proceedings of the second International Conference on Application of Concurrency to System Design*, p. 13, June 2001.

[15] A. Benveniste, B. Caillaud, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, “Tag machines,” in *Proceedings of the 5th ACM International Conference On Embedded Software (EMSOFT 2005)*, pp. 255–263, ACM, September 2005.

[16] K. M. Chandy and L. Lamport, “Distributed snapshots: Determining global states of distributed systems,” *ACM Transactions on Computer Systems*, vol. 3, pp. 63–75, february 1985.

[17] OMG, “Unified Modeling Language: Infrastructure – version 2.1.1,” january 2007.

[18] B. Meyer, *Introduction to the Theory of Programming Languages*. Prentice Hall, Hemel Hempstead (U.K.), 1990.

[19] Eclipse Foundation, “Eclipse Modeling Framework (EMF).”

[20] OMG, “Meta Object Facility (MOF) 2.0 Query/View/ Transformation specification,” november 2005.

[21] C. Hardebolle, F. Boulanger, D. Marcadet, and G. Vidal-Naquet, “A generic execution framework for models of computation,” in *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007), at the European Joint Conferences on Theory and Practice of Software (ETAPS 2007)*, pp. 45–54, IEEE Computer Society, march 2007.

[22] F. Boulanger, M. Mbobi, and M. Feredj, “Flat heterogeneous modeling,” in *IPSI 2004 conference*, 2004.