# MULTI-FORMALISM MODELLING AND MODEL EXECUTION

C. Hardebolle
cecile.hardebolle@supelec.fr

F. Boulanger
frederic.boulanger@supelec.fr

Supélec - Computer Science Department
3 rue Joliot-Curie, 91192 Gif-Sur-Yvette cedex, France

## Keywords

Multi-formalism modelling, Heterogeneous modelling, Model of computation, Simulation of heterogeneous models, Model driven engineering.

## Abstract

Modelling complex software systems requires multiple modelling formalisms adapted to the nature of each part of the system (control, signal processing, etc.), to the aspect on which the model focuses (functionality, time, fault tolerance, etc.) and to the level of abstraction at which the system, or one of its parts, is studied. The use of different modelling formalisms during the development cycle is therefore both unavoidable and essential. As a consequence, system designers deal with a large variety of models that relate to a given system but do not form a global model of this system. A major difficulty is then to answer questions about properties of the whole system, and in particular about its behaviour. *Multi-Formalism Modelling* allows the joint use of different modelling formalisms in a given model in order to overcome issues related to the integration of heterogeneous models. It applies to different tasks of the development cycle such as simulation, verification or testing. We propose an approach to multi-formalism modelling, called ModHel'X, which is based on the concept of *Model of Computation* and focuses on the simulation of models. Our approach addresses two important issues in this particular field: (a) providing support for the specification of the execution semantics of a modelling formalism, and (b) allowing the specification of the interactions between parts of a model described using different modelling formalisms.

## 1. Introduction

In the context of Model Driven Engineering (MDE), the development of complex software systems calls for different modelling techniques depending both on the different phases of the development cycle (specification, design, test, etc.) and on the different technical domains at stake in the designed system (control, user interface, signal processing, etc.). In particular, the analysis of specific aspects of a design (time, performance, etc.) or the description of the system at different levels of abstraction (system level, algorithmic level, register transfer level) requires the use of adequate modelling languages [19]. The use of different modelling formalisms during the development cycle and for different parts of a system is therefore both unavoidable and essential. As a consequence, system designers have to deal with a large variety of models that relate to a given system but do not form a global model of this system. As a manual task, the integration of heterogeneous models in order to obtain a global view of the system is both tedious and error prone, not mentioning issues related to traceability and maintenability when modifications have to be reported on the integrated models. Answering questions about properties of the whole system, and in particular about its behaviour [4], is therefore a major difficulty. The objective of *Multi-Formalism Modelling* [3] is to ease and automate the integration of heterogeneous models by allowing the joint use of different modelling formalisms in a given model. With respect to the development cycle, multi-formalism modelling is a transverse discipline: it applies to different activities of the development cycle such as specification, verification, code generation or testing. Different aspects of multi-formalism modelling (also called heterogeneous modelling) have been studied: mathematical foundations [10], tools for validation [25] or simulation [27]. A central problem is to establish the meaning of the composition of heterogeneous parts of a model and to ensure their correct interoperation when using the model to answer questions about the designed system [21]. In this paper, we describe an approach called ModHel'X which addresses the issue of the composition of heterogeneous parts of a model in the context of model execution.

The remainder of the paper is organized as follows. We first motivate our approach in Section 2. Section 3 details the main principles of ModHel'X, which we illustrate through an example in section 4. We discuss some specific aspects of our approach in Section 5. Then we review some of the related work in Section 6 before concluding.

This work has been performed in the context of
the Usine Logicielle project
of the System@tic Paris Région Cluster
(www.usine-logicielle.org)

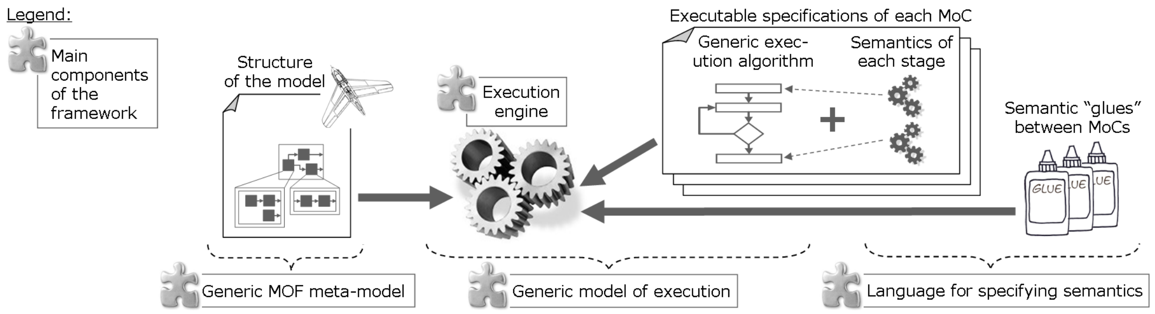
Legend:
Main components of the framework
Structure of the model
Execution engine
Executable specifications of each MoC
Generic execution algorithm
Semantics of each stage
Semantic "glues" between MoCs
GLUE
Generic MOF meta-model
Generic model of execution
Language for specifying semantics

Figure 1: The ModHel'X approach

## 2. Motivations

In the context of model execution, the first difficulty when determining the semantics of a composition of heterogeneous models is to have a precise executable specification of the semantics of the involved modelling formalisms. Indeed, except for a few mathematically founded languages, the semantics of a modelling language is often described using natural language, which may lead to ambiguities and to diverse interpretations by different tools along the design chain. When combining different modelling 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.

A second difficulty for obtaining a meaningful multi-formalism model of a system is to be able to define the semantic adaptation (i.e. the "glue") to use between model parts that use different modelling 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 are provided by different technical teams or by suppliers for instance. Another constraint is that the semantic adjustment between heterogeneous parts of a model depend not only on the formalisms at stake but also on the system which is modeled. Usual adaptation patterns between modelling formalisms often exist, but they are not unique and may need parameter adjustments since they represent default adaptations which do not necessarily fit directly a particular context. For example, integrating a model part which focuses on the notion of time with another which does not, may imply customizing the adaptation which is realized on the notion of time so that it is coherent with the expected behaviour of the system.

An approach for specifying the semantics of a modelling language is to define the constructs of the language in a fixed abstract syntax – or meta-model – which is component oriented (as in [31]), and to consider that the semantics of the language is given by its "Model of Computation" (MoC). Such an approach is implemented in Ptolemy [4]. 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. modelling 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 the modelers have either to rely on the default adaptation performed by the tool, or to modify the design of parts of their model (by adding adaptation components) in order to obtain the behaviour they expect.

In order to illustrate this last issue, let us consider, for example, the model of a system which integrates known traffic conditions when computing driving itineraries. This system is composed of an algorithm which computes a route to the destination from 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 modelling 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 immediate responses to inputs provided in flows while, in DE, events can be produced at any time, not necessarily synchronously (e.g. the traffic information retrieving system produces 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 no longer represents the behaviour of the

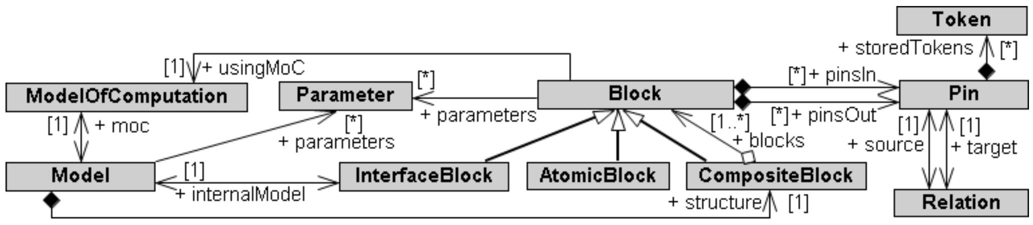

Figure 2: Generic meta-model for representing the structure of models

component which is delivered by the supplier. This may lead to implementation incoherences at the end of the development cycle.

- Since the semantic adaptation is done in the model of the component, it is not protected from changes made by the supplier.
- If the formalism used in one of the models changes – e.g. when refining the model in order to take finer details into account – the adaptation must be expressed again in the new formalism. This is incompatible with modularity and reuse.

It is therefore important to allow the designers to specify the semantic adaptation outside the models of the parts they assemble.

The approach that we propose is based on the concept of *model of computation (MoC)* as defined in [4]. Fig. 1 illustrates the structure of the ModHel'X framework. In order to interpret a model in ModHel'X, it is necessary to describe its structure using our MOF 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 behaviour as the interpretation of the original model according to the semantics of its modelling language. The way MoCs are specified in ModHel'X is detailed in section 3.4. 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. 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 executable specification of the models of computation and of their interactions to determine without ambiguity the behaviour of multi-formalism models.

# 3. The ModHel'X approach

## 3.1 Black boxes and snapshots

In ModHel'X, we adopt a component-oriented approach and 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 behaviour 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 computing something at a given moment.

In addition, instead of "triggering" the behaviour 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 them: we simply observe them at instants suitable for the embedding model. The behaviour of a block or a model is therefore a sequence of 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* [11] 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 [38]). 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, for instance because its state has changed. To this end, each component of an heterogeneous model can give constraints on its own 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

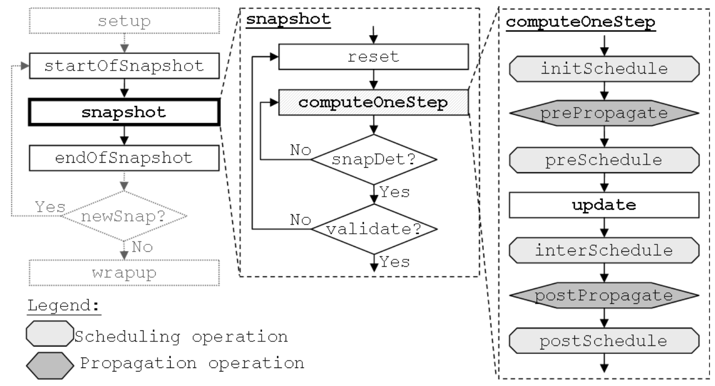

Figure 3: Generic execution algorithm

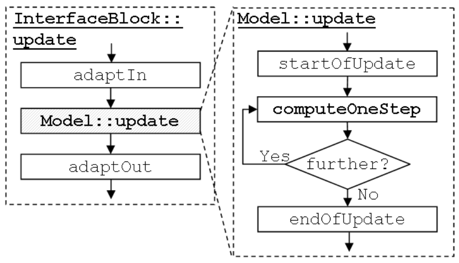

Figure 4: Update on an interface block and its internal model

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 Generic MOF meta-model for representing the structure of models

The generic meta-model that we propose, shown on Fig. 2, 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 modelling language, but their semantics is given by the MoCs which interprets them.

In the *structure* of a *model*, *blocks* are the basic units of behaviour. *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 behaviour: they are interpreted according to the MoC in order to determine how to combine the behaviours of the blocks they connect. For instance, a relation can represent a causal order between two blocks as well as a communication channel. It is important to note that similar concepts can be found in many modelling approaches, in particular in component-oriented ones (such as Ptolemy II for instance).

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 only carry a value. 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 behaviour 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 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 Describing models of computation 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 [11]. 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 behaviour 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.

### 3.4.1 *Overall structure of the algorithm*

Fig. 3 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 its elements 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 in the execution of the model. In the computation of a snapshot, the computation of an observation of one block

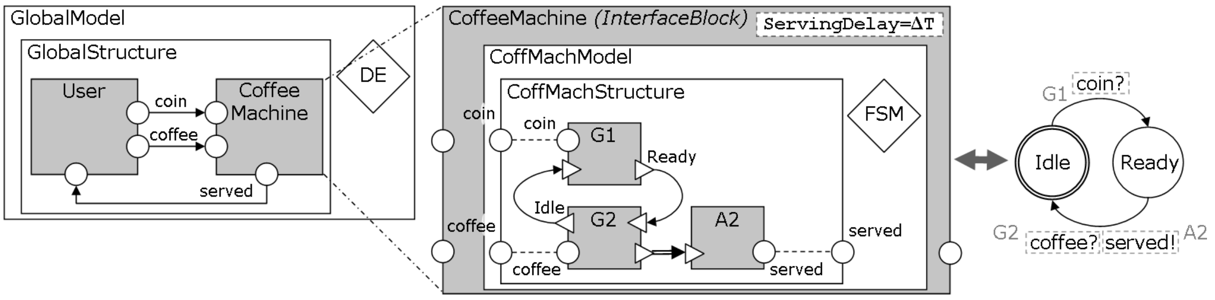

Figure 5: Global model of the coffee machine and coffee machine automaton

brings into play the scheduling and propagation operations mentioned above, and is called a *step* (represented on the right part of Fig. 3 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, which allows the use of non-strict [18] blocks for the computation of fixed point behaviours. Therefore, ModHel'X supports MoCs in which cyclic dependencies are allowed.

### *3.4.2 Computation of one step in a snapshot*

The basic sequence for performing a computation step is to choose a component according to the state of the model and the available inputs (`initSchedule`), 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 in which 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 may not be appropriate for more formal applications such as testing, model checking or validation. An example of a scheduling operation is shown on Listing 1 page 6.

### *3.4.3 Hierarchical execution*

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. A model which is embedded into an interface block is only asked to provide a coherent view of its behaviour when its interface block is updated. The `update` operations of interface blocks and models are shown on Fig. 4. 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. Therefore, these operations represent the meaning the modeler gives to the joint use of two models of computation.

In `adaptIn`, data from the external model is interpreted and translated into the formalism of the internal 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 internal model. On the contrary, if two inputs are exclusive, `adaptIn` may be used to deliver them to the internal model in two separate snapshots if they happen simultaneously in the external model. `adaptIn` can therefore change the data that is passed to the internal model, but it can also change the control, e.g. define when the internal 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

Listing 1: `initSchedule` operation of the DE MoC

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

current snapshot for the internal model of computation from the time stamp of the external 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 `startOfUpdate` operation is used to take new adapted inputs from the interface block into account, and the `endOfUpdate` 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 internal model and translates it so that it is meaningful to the external 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. An example of an `adaptOut` operation is shown on Listing 2 page 7.

### 3.5 Implementation and validation

We have experimented our approach in a prototype of ModHel'X based on the Eclipse EMF framework [39]. We use the ImperativeOCL [37] 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 [20]. 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.

The use of ImperativeOCL for specifying the semantics of the operations of our algorithm is not a definitive choice. We have observed that ImperativeOCL specifications are too verbose and do not allow the designer to work at a suitable level of abstraction. Moreover, ImperativeOCL's semantics is not well defined yet, mainly because it includes complex object-oriented constructs that are irrelevant to our particular application. We are exploring other options including model transformation languages and pre/post conditions such as found in Hoare's logic.

## 4. A multi-formalism example

To illustrate our approach, and in particular the semantic adaptation between a timed and an untimed MoC, we consider as an example a simple hierarchical and heterogeneous model of a coffee machine which works as follows: first the users have to insert a coin, then they have to press the "coffee" button to get their coffee after some preparation time.

### 4.1 Modelling formalisms

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 the coffee. Therefore, we use the Discrete Events (DE) MoC, which is implemented, for example, by SimEvents (The MathWorks) or Verilog. We represent our user by an atomic block, whose behaviour 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 behaviour. We consider here a simple version of this MoC called FSM (Finite State Machines), which is similar to the one presented in [22]. Fig. 5 shows the global model resulting from the combination of the DE and FSM models. We use here a basic graphical syntax in which models and blocks are represented by rectangles, pins by circles and triangles and relations by arrows. A more natural representation of the FSM model (i.e. of the automaton of the coffee machine) is shown on the right part of the figure. The combination of the DE and FSM MoCs is a classical example, which is well addressed by tools like Ptolemy. However, we will see that it is possible 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

Listing 2: `adaptOut` operation of the interface block

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

may have two associated behaviours: the evaluation of its guard and its action. Since blocks are the basic units of behaviour 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, in our implementation of FSM, the current state of the automaton is represented by the set of the currently "fireable" guard blocks. The initial state of the automaton is the set of the initially fireable guard blocks (which contains only the $G1$ block in our example).

In our implementation of DE, which is quite similar to the implementation of DE in Ptolemy, events are stored in a global event queue. When a snapshot is taken, the current time is determined according to the time stamps of the events in the queue and on the time constraints produced by the blocks. At each computation step, we consider the blocks that have posted a time constraint for the current time and the blocks that are the target of events with a time stamp equal to the current time. The semantics of DE assumes that a given block is observed only once in a snapshot, so every block in DE must 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. Events that are delivered to their destination block are removed from the queue. A snapshot is complete only when all the events at the current time have been processed. The events that have a time stamp corresponding to a future time remain in the queue for the next snapshot. Listing 1 shows the code of the `initSchedule` operation for the DE MoC, which is in charge of choosing the block to update in the current step, given the rules that we just mentioned. This is the only scheduling operation for this MoC, the others (`pre`, `inter` and `postSchedule`) being left empty. The `topologicalSort` function is used to find a minimal block according to the partial order induced by the dependency relations between the blocks of the model.

## 4.2 Semantic adaptation of DE and FSM

DE and FSM share the notion of event. However, FSM has no notion of time attached to 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. When an FSM event enters DE, the interface block has to give it the "right" time stamp. 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 behaviour does not model the serving delay, which is an important characteristic of the model. 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 sum of the time stamp of the last `coffee` event and the `ServingDelay` (see the code for the `adaptOut` operation on Listing 2).

## 4.3 Running the example

When executing this example, the execution engine computes three successive snapshots. The trace of the execution sequence is summarized in Table 1. We only show in this table the execution steps of the algorithm that modify the different computation variables. Each line shows the values of the variables after the execution of the corresponding execution step.

During the setup, the user block produces a constraint in order to be observed at time 0 (so that there is a snapshot when it inserts the coin into the coffee machine). This constraint triggers the first snapshot. At the start of the snapshot, the event queue of the DE model is empty, and the set of currently fireable guards in the FSM model contains the $G1$ block. Because of the constraint produced by the user, the current time of the DE model is set to 0 at the first snapshot. No other block having produced a constraint or emitted an event yet, the user is therefore chosen for update by the `initSchedule` operation during the first computation step. When updated, the user emits the `coin` event with time stamp 0. A second computation step happens because the time stamp of the `coin` event is equal to

Table 1
Execution steps when running the coffee machine example

| Execution steps | DE MoC | | InterfaceBlock | FSM MoC |
|---|---|---|---|---|
| | **Event queue** | **Constraints** | **Last stored time stamp** | **Current fireable guard blocks** |
| Setup | [] | $[user : 0]$ | $t_{lastDEevt} = 0$ | $G1$ |
| **First snapshot:** | | | | |
| start of snapshot | [] | $[user : 0]$ | $t_{lastDEevt} = 0$ | $G1$ |
| update of the user | $[coin : 0]$ | [] | $t_{lastDEevt} = 0$ | $G1$ |
| adaptIn in the coffee machine | [] | [] | $t_{lastDEevt} = 0$ | $G1$ |
| update of G1 | [] | [] | $t_{lastDEevt} = 0$ | $G2$ |
| adaptOut in the coffee machine | [] | [] | $t_{lastDEevt} = 0$ | $G2$ |
| end of snapshot | [] | $[user : t_{user}]$ | $t_{lastDEevt} = 0$ | $G2$ |
| **Second snapshot:** | | | | |
| start of snapshot | [] | $[user : t_{user}]$ | $t_{lastDEevt} = 0$ | $G2$ |
| update of the user | $[coffee : t_{user}]$ | [] | $t_{lastDEevt} = 0$ | $G2$ |
| adaptIn in the coffee machine | [] | [] | $t_{lastDEevt} = t_{user}$ | $G2$ |
| update of G2 | [] | [] | $t_{lastDEevt} = t_{user}$ | $G1$ |
| update of A2 | [] | [] | $t_{lastDEevt} = t_{user}$ | $G1$ |
| adaptOut in the coffee machine | $[served : t_{user} + \Delta T]$ | [] | $t_{lastDEevt} = t_{user}$ | $G1$ |
| end of snapshot | $[served : t_{user} + \Delta T]$ | [] | $t_{lastDEevt} = t_{user}$ | $G1$ |
| **Third snapshot:** | | | | |
| start of snapshot | $[served : t_{user} + \Delta T]$ | [] | $t_{lastDEevt} = t_{user}$ | $G1$ |
| update of the user | $[\ldots]$ | [] | $t_{lastDEevt} = t_{user}$ | $G1$ |

the current time of the snapshot. The coffee machine block is chosen for update since it is the target of the `coin` event. Before delegating the update to its internal model, the coffee machine InterfaceBlock executes its `adaptIn` operation to remove the time stamp of the `coin` event. Then the `coin` event is delivered to the `coin` pin of the internal model. The current fireable guard $G1$ is updated and consumes the `coin` event. $G2$ becomes then the current fireable guard. Since no event has been produced by the automaton, the execution of the `adaptOut` operation has no effect. The DE event queue remains empty and the snapshot is therefore over. The user block produces a constraint during the `endOfSnapshot` operation in order to require to be observed at time $t_{user}$ (when it presses the "coffee" button of the coffee machine).

Because of the constraint produced by the user, a second snapshot is triggered and the current time of DE is then $t_{user}$. At the start of the snapshot, the event queue of the DE model is empty, and the set of currently fireable guards in the FSM model contains the $G2$ block. The same computation steps as in the first snapshot happen until the `coffee` event, emitted by the user with time $t_{user}$, is delivered to the coffee machine. During the execution of the `adaptIn` operation, the time stamp of the `coffee` event is removed and its value is stored. Then, the event is delivered to the `coffee` pin of the internal model. The currently fireable guard $G2$ is updated. $G2$ consumes the `coffee` event, $G1$ becomes the current fireable guard and the $A2$ action is scheduled for update. When updated, the $A2$ action produces the `served` event and the FSM model finishes its update. The `adaptOut` operation computes the time stamp of the `served` event by adding the `ServingDelay` value to the stored time stamp of the last `coffee` event. The `served` event is therefore emitted by the coffee machine with a time stamp equal to $t_{user} + \Delta T$ and stored in the DE event queue. Since this event has a time stamp in the future, and no other event for the current time remain in the queue, the snapshot is over.

The last snapshot happens because the `served` event, which remains in the DE event queue, has to be processed. The current time of the DE model is then $t_{user} + \Delta T$ and the `served` event is delivered to the user. What happens then depends on the implementation of the user block. For example, if this block represents a user who drinks coffee all day, it may produce another `coin` event right away with a time stamp equal to $t_{user} + \Delta T$ or to $t_{user} + \Delta T + \delta_{user}$ and the cycle of snapshots may therefore continue.

## 5. Discussion

### 5.1 Required effort for using ModHel'X

There are two prerequisites to the use of the ModHel'X framework. First, an expert of a modelling 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 modelling tools but to allow their joint use in a controlled way, this expert also defines transformations from the original meta-model of the language to our generic meta-model. The specification of the semantics of a modelling language in ModHel'X is the difficult part of the work because the semantics of modelling tools is often known only intuitively, through the experience we have of the tools, and not formally. When no consensus exists on the semantics of a modelling language, this task may even be contentious. However, the advantage of fixing the semantics to use is that no doubt remains on the behaviour of the model, which is absolutely necessary for verification, validation. It is also a major advantage when exchanging models among several actors of the development cycle. This first step is done once and for all for each modelling language.

Second, for each pair of MoCs that may interact in heterogeneous models, experts should define interaction patterns, which model standard ways of combining models that obey these MoCs. The interaction policy actually used in a particular model will be a specialization of one of these patterns, tuned using parameters. The design of interaction patterns depends on the technical domains at stake and on the habits of the designers since it formalizes their know-how and their usual ways of solving heterogeneity issues. Therefore, it is also a difficult task. As a counterpart, it allows the modeler to control what is happening at the border between two heterogeneous models. It is necessary to define at least one semantic adaptation policy for each pair of MoCs that one intends to use together. However, there is no need to define such a policy for any pair of MoCs, first because there may be no sense in making some MoCs interact, second because even when they are used together in a model of a system, some models of computation never interact directly. The number of useful combinations of MoCs is therefore much lower than the number of combinations that are theoretically possible for $N$ MoCs.

### 5.2 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 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 large range of models of computation. This includes MoCs for continuous behaviours, which are approximated by the computation of a series of discrete observations. 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. Last, even if the execution engine of ModHel'X is deterministic (because it is designed to compute one of the possible behaviours of a model during an execution), non deterministic MoCs are supported but require the use of pseudo-random functions in their specification. Such MoCs may be useful when modelling the non deterministic environment of a system, for instance.

### 5.3 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 (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. 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.

Like Ptolemy, ModHel'X is designed to support the widest possible range of modelling formalisms. This implies that its generic execution algorithm is not as efficient as a specific algorithm optimized for a given formalism. In order to add support for the explicit specification of semantic adaptation, we have split our execution algorithm into finer-grained execution steps than those of Ptolemy. The study of the impact this choice may have on performance is planned as future work.

## 6. Related work

Because of its multi-disciplinary nature, multi-formalism modelling has involved research teams with technical backgrounds as different as control science, signal processing, model checking, modelling language engineering or system-on-chip development. In this section, we review different approaches to multi-formalism modelling and propose to sort them according to three main criteria: (1) support for an open set of modelling languages, (2) support for multiple design activities and (3) support for formal verification of properties.

## 6.1 Support for an open set of languages

The ad-hoc combination of a finite set of modelling languages is a well addressed issue. VHDL-AMS, for example, allows the combination of discrete events and continuous time models. However, the development of MDE entails the use of new modelling languages – in particular Domain Specific Languages – thus leading to the need for more flexible tools. In this section, we present approaches that support an open set of modelling languages. They permit the specification of the semantics of modelling languages in two different ways: using meta-models and model transformation, or using models of computation.

### *6.1.1 Meta-modelling and model transformation*

In meta-modelling approaches such as Kermeta [22], the abstract syntax of a modelling language is described as a meta-model. The elements of this meta-model have methods whose semantics is defined in an imperative language. Each modelling language has a different meta-model in Kermeta. In the context of heterogeneous modelling, the definition of the combination of several modelling 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 the modification of the meta-model and of the associated model transformations each time an additional modelling 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 that must be answered about the system. Such an approach is implemented in the $ATOM^3$ tool [23]. However, the way the different heterogeneous parts of the model are "glued" together does not seem to be addressed by this approach.

Other approaches [24, 6] are also based on model transformation. In particular, [6] states that it is possible to formally define the semantics of a modelling language by defining a mapping to an already formally defined modelling language.

### *6.1.2 Models of computation*

Another approach for defining the semantics of a modelling language is to define the constructs of the language in a fixed abstract syntax – or meta-model – which is component oriented (as in [31]), and to consider that the semantics of the language is given by its "Model of Computation" (MoC). Such an approach is implemented in Ptolemy [4] and in ModHel'X.

The "42" approach [26], which is also based on the notion of model of computation, relies 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 resides in the description of the behavioural contract of components. However, such a description may not be available (in the case of an external IP – Intellectual Property – for instance) or may not be easy to establish, in the case of continuous time behaviours for example.

## 6.2 Support for multiple design activities

As they support different design activities, the approaches that we introduce in this section are able to assist the designers along different phases of the development cycle. This is a key benefit when the number of design tools tends to increase dramatically.

In [19], the authors propose an approach to multi-formalism modelling which is oriented toward state-space analysis. The approach relies on an "inframodel" which captures only the aspects of a formalism that are essential for state-space exploration and supports customization for reflecting the particular semantics of each formalism. Hypergraphs are used for representing models. This approach supports a wide range of model analysis techniques, from reachability analysis to model execution. However, it seems that the supported formalisms are limited to state transition formalisms for concurrent systems.

Similarly to the approaches of Section 6.1.2, Metropolis [7] relies on the concept of model of computation, but it focuses on MoCs related to process networks. In Metropolis, the modelling of the function is separated from the modelling of the architecture. A mapping mechanism is provided to produce platform specific models. Metropolis includes tools for verification, simulation and synthesis.

## 6.3 Support for formal properties

In this last section, we present two approaches that allow reasoning about the formal properties of heterogeneous models. Since they target the verification and validation phases of the development cycle, these approaches are of particular interest when designing critical software.

BIP (Behaviour, Interaction, Priority) [25] takes advantage of a hierarchical and component oriented abstract syntax and 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 behaviour. This allows the formal verification of properties on the model.

Rosetta [30] is a system specification language with a formal semantics. It introduces the notion of "facet", which models a particular aspect of a component, e.g. its behaviour or its energy consumption. Rosetta allows the combination of models of computation either in order to assemble components or to model different aspects of one component. Therefore, it provides the designer with the ability to combine multiple views of parts of the system.

## 7. Conclusion

After an overview of the main issues in multi-formalism modelling for model execution, we have presented an approach that (a) provides support for the specification of the semantics of a modelling formalism through the concept of model of computation, and (b) allows the definition of the interactions between heterogeneous parts of a model through a special modelling construct and using an imperative syntax. This approach relies on the black-box and the snapshot paradigms to compute the observable behaviour of a model by combining the behaviours 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 operations of a generic algorithm which computes series of snapshots of models which conform to the proposed meta-model.

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 formal semantics. Therefore, ModHel'X cannot be used for model-checking or demonstrating properties. We are currently studying several possibilities for replacing ImperativeOCL with a more concise and formal language. Moreover, we are considering the use of a formal framework for founding the semantics of our execution algorithm.

## References

### Journal Papers

[1] A. Mbobi, F. Boulanger, and M. Feredj, "An approach of flat heterogeneous modeling based on heterogeneous interface components," *International Review on Computers and Software*, vol. 2, pp. 179–189, March 2007.

[2] M. Feredj, F. Boulanger, and A. Mbobi, "A model of domain-polymorph component for heterogeneous system design," *Journal of Systems and Software*, vol. 82, no. 1, pp. 112–120, 2009.

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

[4] 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.

[5] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivan, c Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky, "Hierarchical modeling and analysis of embedded systems," in *Proceedings of the IEEE*, october 2002.

[6] 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.

[7] 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.

[8] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Science Of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[9] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM.*, vol. 21, no. 8, pp. 666–677, 1978.

[10] 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.

[11] 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.

[12] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, september 1987.

[13] A. Jantsch and I. Sander, "Models of computation and languages for embedded system design," *IEE Proceedings on Computers and Digital Techniques*, vol. 152, pp. 114–129, march 2005. Special issue on Embedded Microelectronic Systems.

[14] D. Lugato, C. Bigot, Y. Valot, J.-P. Gallois, S. Gérard, and F. Terrier, "Validation and automatic test generation on uml models : the agatha approach," *Software Tools for Technology Transfer*, vol. 5, pp. 124–139, march 2004.

### Books

[15] G. T. Heineman and W. T. Councill, eds., *Component-Based Software Engineering*. ACM Press, Addison-Wesley Professional, New-York, NY, USA, 2001.

[16] T. Clark, A. Evans, P. Sammut, and J. Willans, *Applied metamodelling: A foundation for language driven development*. Xactium Ltd., Sheffield, UK, 2004.

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

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

## Proceedings Papers

[19] M. Pezzè and M. Young, "Generation of multi-formalism state-space analysis tools," in *Proceedings of the 1996 International Symposium on Software Testing and Aanalysis*, pp. 172–179, ACM, New-York, 1996.

[20] C. Hardebolle, F. Boulanger, D. Marcadet, and G. Vidal-Naquet, "A generic execution framework for models of computation," in *Proceedings of MOMPES 2007, at ETAPS 2007*, pp. 45–54, IEEE Computer Society, Los Alamitos, CA, USA, March 2007.

[21] 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 Berlin/Heidelberg, August 2006.

[22] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, "Weaving executability into object-oriented meta-languages," in *Proceedings of MODELS/UML 2005*, pp. 264–278, Springer Berlin/Heidelberg, 2005.

[23] J. de Lara and H. Vangheluwe, "$ATOM^3$: A tool for multi-formalism modelling and meta-modelling," in *Proceedings of FASE 2002*, pp. 174–188, Springer-Verlag, London UK, april 2002.

[24] 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.

[25] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," in *Proceedings of SEFM06*, pp. 3–12, IEEE Computer Society Washington, DC, USA, september 2006.

[26] F. Maraninchi and T. Bouhadiba, "42: Programmable models of computation for a component-based approach to heterogeneous embedded systems," in *Proceedings of GPCE'07*, pp. 53–62, ACM New York, NY, USA, october 2007.

[27] P. Fritzson and V. Engelson, "Modelica — A unified object-oriented language for system modeling and simulation," in *Proceedings of ECOOP98*, pp. 67–90, Springer-Verlag, London, UK, july 1998.

[28] A. Benveniste, B. Caillaud, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Tag machines," in *Proceedings of EMSOFT 2005*, pp. 255–263, ACM New-York, NY, USA, September 2005.

[29] 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, IEEE Computer Society, Washington, DC, USA, June 2001.

[30] C. Kong and P. Alexander, "The Rosetta meta-model framework," in *Proceedings of ECBS'03*, pp. 133–140, IEEE Computer Society, Los Alamitos, CA, USA, April 2003.

## Miscellaneous

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

[32] J. Liu, "Continuous time and mixed-signal simulation in Ptolemy II," tech. rep., EECS Department, University of California, Berkeley, 1998.

[33] B. Lee, "Specification and design of reactive systems," Tech. Rep. UCB/ERL M00/29, EECS Department, Univ. of California, Berkeley, 2000.

[34] OMG, "UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) RFP," february 2005.

[35] L. Muliadi, "Discrete event modeling in Ptolemy II," Tech. Rep. UCB/ERL M99/29, EECS Department, Univ. of California, Berkeley, May 1999.

[36] OMG, "Object Constraint Language specification, version 2.0," May 2006.

[37] OMG, "Meta Object Facility 2.0 Query-View-Transform specification," Nov. 2005.

[38] OMG, "Unified Modeling Language: Infrastructure – version 2.1.1," January 2007.

[39] Eclipse Foundation, "Eclipse Modeling Framework (EMF)."

[40] ATLAS group (INRIA – LINA), "ATLAS Transformation Language (ATL)."

[41] Commissariat à l'Énergie Atomique, "Papyrus UML Modeler."

## Biographies

*Frédéric Boulanger* is a professor at the Computer Science Department of Supélec. He got is engineering degree from Supélec in 1989, and a PhD in Computer Science from Paris-Sud XI University in 1993. He is interested in the design and validation of heterogeneous systems.

*Cécile Hardebolle* is a PhD student at the Computer Science Department of Supélec. She got her engineering degree from Supélec in 2004. Started in January 2006, her thesis relates to heterogeneous modelling, the execution of models and meta-modelling.