Bridging the Chasm between Executable Metamodeling and Models of Computation*

Benoit Combemale[1], Cécile Hardebolle[2],
Christophe Jacquet[2], Frédéric Boulanger[2], and Benoit Baudry[3]

[1] University of Rennes 1, IRISA
[2] Supélec E3S – Computer Science Department
[3] Inria Rennes - Bretagne Atlantique

Abstract. The complete and executable definition of a Domain Specific Language (DSL) includes the specification of two essential facets: a model of the domain-specific concepts with actions and their semantics; and a scheduling model that orchestrates the actions of a domain-specific model. Metamodels can capture the former facet, while Models of Computation (MoCs) capture the latter facet. Unfortunately, theories and tools for metamodeling and MoCs have evolved independently, creating a cultural and technical chasm between the two communities. Consequently, there is currently no framework to explicitly model and compose both facets of a DSL. This paper introduces a new framework to combine a metamodel and a MoC in a modular fashion. This allows (i) the complete and executable definition of a DSL, (ii) the reuse of a given MoC for different domain-specific metamodels, and (iii) the use of different MoCs for a given metamodel, to account for variants of a DSL.

1 Introduction

Domain-specific languages (DSLs) offer a limited, dedicated set of concepts to domain experts to let them express their concerns about a system. Previous studies have shown that the limited expressiveness of DSLs, combined with dedicated tools, can increase the productivity in the construction of software-intensive systems, while reducing the number of errors [1]. A recent study by Hutchinson *et al.* has even demonstrated that DSLs are one of the main motors for an industrial adoption of model-driven engineering [2].

Defining a DSL completely and precisely is difficult, in particular when it comes to the formal definition of its semantics. However, Bryant *et al.* [3] point out that the formal definition of DSL semantics is the foundation for the major expected benefits of DSLs: the automatic generation of the DSL tooling (*e.g.,* editor and compiler), the formal analysis of model behavior, or the rigorous composition of multiple concerns modeled with different languages.

* This work has been partially supported by VaryMDE, a collaboration between Inria and Thales Research and Technology.

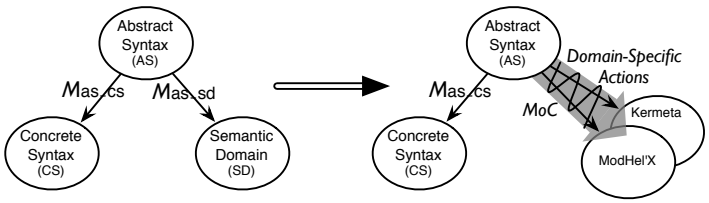

Fig. 1. Our approach to implement the behavioral semantics of a DSL

As described in the left of Figure 1, Harel *et al.* synthesizes the construction of a DSL as the definition of a triple: abstract syntax, concrete syntax and semantic domain [4]. This work focuses on the definition of the abstract syntax (AS), the semantic domain (SD) and the respective mapping between them (M_{as_sd}). Several techniques can be used to define those three elements. This paper focuses on executable metamodeling techniques, which allow one to associate operational semantics to a metamodel. In this context, we argue that the formal definition of the semantic domain must rely on two essential assets: the semantics of domain-specific actions and the scheduling policy that orchestrates these actions. It is currently possible to capture the former in a metamodel and the latter in a *Model of Computation (MoC)*, but the supporting tools and methods are such that it is very difficult to connect both to form a whole semantic domain (see right of Figure 1).

We propose to model domain-specific actions and MoCs in a modular and composable manner, resulting in a complete and executable definition of a DSL. We experiment this proposal by leveraging two state-of-the-art modeling frameworks developed in both communities: the Kermeta workbench [5] that supports the investigation of innovative concepts for metamodeling, and the ModHel'X environment [6] that supports the definition of MoCs. We foresee two major benefits for this composition: the ability to reuse a MoC in different DSLs, and the ability to reuse domain-specific actions with different MoCs to implement semantic variation points of a DSL. Saving the verification effort on MoCs and domain-specific actions also reduces the risk of errors when defining and validating new DSLs and their variants. We illustrate this approach and the reuse capacities through the actual composition of the fUML DSL with a sequential and then a concurrent version of the discrete event MoC.

The rest of the paper proceeds as follows: Section 2 introduces fUML, our case study throughout the paper. Then we describe how to design the domain-specific actions of a DSL and the MoC, respectively using Kermeta (Section 3) and ModHel'X (Section 4). We propose in Section 5 a tool-supported approach to combine them to implement the complete behavioral semantics of a DSL in a modular and reusable fashion. Finally, we present in Section 6 the application of our approach to vary the MoC of fUML. Section 7 presents related work, and Section 8 concludes and proposes directions of future work.

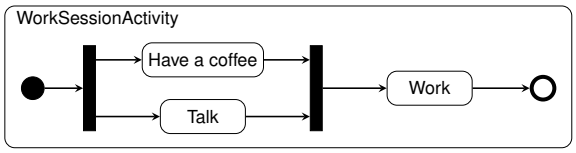

Fig. 2. Activity of the members of our team during our work sessions.

2 Case Study: fUML

The *Semantics of a Foundational Subset for Executable UML Models (fUML)* [7] is an executable subset of UML that can be used to define the structural and behavioral semantics of systems. It is computationally complete by specifying a full behavioral semantics for activity diagrams. This means that this DSL is well-defined and enables implementors to execute well-formed fUML models (here *execute* means to actually *run a program*).

As an example, Figure 2 shows an executable fUML model representing the activity of our team when we meet for work sessions. We are used to first having a coffee while talking together about the latest news. When we finish drinking our coffee and talking, we begin to work.

The fUML specification includes both a subset of the abstract syntax of UML, and an execution model of that subset supported by a behavioral semantics. We introduce these two parts of the specification in the rest of this section.

2.1 The fUML Abstract Syntax

Figure 3 shows an excerpt of the fUML metamodel corresponding to the main concepts of the abstract syntax. The core concept of fUML is *Activity* that defines a particular behavior. An *Activity* is composed of different elements called *Activity Nodes* linked by *Activity Edges*. The main nodes which represent the executable units are the *Executable Nodes*. For instance, *Actions* are associated to a specific executable semantics. Other elements define the activity execution flow, which can be either a control flow (*Control Nodes* linked by *Control Flow*) or a data flow (*Object Nodes* linked by *Object Flow*).

The example in Figure 2 uses an illustrative set of elements of the abstract syntax of fUML. The start of the *Activity* is modeled using an *Initial Node*. A *Fork Node* splits the control flow in two parallel branches: one for the *Action* of having a coffee, the other for the *Action* of talking to each other. Then a *Join Node* connects the two parallel branches to the *Action* of working.

Of course, the abstract syntax also includes additional constraints in the metamodel to precise the well-formedness rules (a.k.a. static semantics). For example, such an additional constraint expresses that control nodes can only be linked by control flows. fUML uses the Object Constraint Language (OCL) [8] in order to define those constraints.

We refer the reader to the specification of fUML for all the details about the comparison with UML2 and the whole description of the fUML metamodel [7].

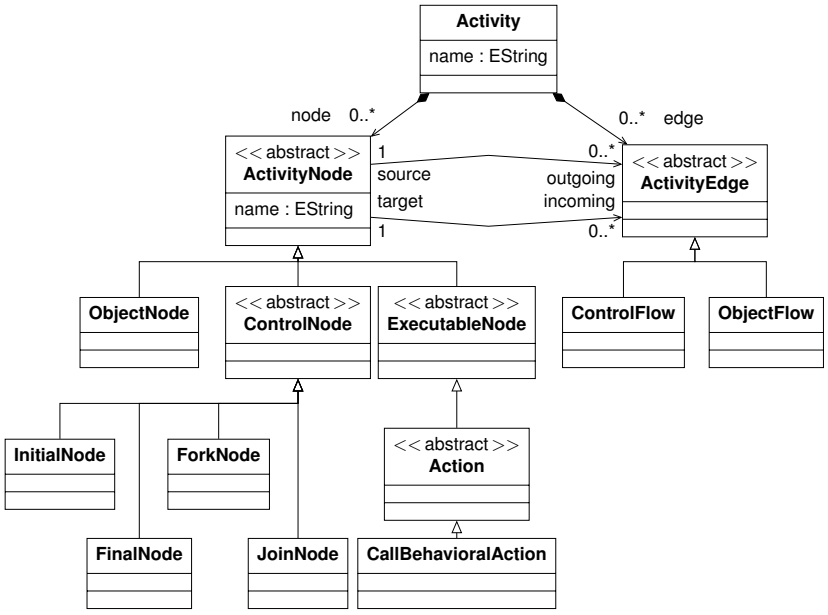

Fig. 3. Excerpt of the fUML Metamodel

2.2 The fUML Behavioral Semantics

To support the execution of models, fUML introduces a dedicated *Execution Model*. The activity execution model has a structure largely parallel to the abstract syntax using the *Visitor* design pattern [9] (called *SemanticVisitor*). Note that although the semantics is explained using visitors, which are rather at the implementation level, it is left open by the fUML specification to implement the language using other means than visitors.

In addition, to capture the behavioral semantics, the interpretation needs to define how the execution of the activity proceeds over time. Thus, concepts are introduced in the execution model for which there is no explicit syntax. Such concepts support the behavior of an activity in terms of *tokens* that may be held by nodes and *offers* made between nodes for the movement of these tokens.

Based on the execution model, the specification denotationally describes the behavioral semantics of fUML using axioms in first order logic. Moreover, a reference implementation of the fUML semantics has been proposed in Java[4]. Both define the *domain-specific actions* (i.e., the behavioral semantics of the domain-specific concepts defined in the abstract syntax) as a concrete implementation of the visitor, including a deeply scattered scheduling of such domain-specific actions. We refer to the latter concern as (part of) the *Model of Computation* (MoC) of the language. Such an implementation prevents the reuse of a MoC for different DSLs (e.g., the fact that all the domain-specific actions should run in sequence is a behavioral specification that can be reused in many domains),

[4] Cf. http://portal.modeldriven.org/

as well as its easy replacement with another one for the same DSL. Indeed, several semantic variation points exist in the MoC. As stated by the specification itself, some semantic areas "are not explicitly constrained by the execution model: The semantics of time, the semantics of concurrency, and the semantics of inter-object communications mechanisms" [7]. We investigate in the rest of this paper an approach to modularly define the domain-specific actions and the MoC of a software language. Such an approach aims at supporting the reuse and the variability in languages, and is illustrated through the fUML case study.

3 Using Kermeta for an executable metamodel of fUML

Kermeta is a workbench that can be used for implementing domain-specific languages (DSLs). It supports different meta-languages depending on the DSL concern (abstract syntax, static semantics, behavioral semantics and connection to concrete syntax), and modularization features. In this section, we provide some background on Kermeta, and we show benefits and drawbacks in implementing both the abstract syntax and the behavioral semantics.

3.1 Abstract Syntax Definition

First of all, to build a DSL in Kermeta, one implements its abstract syntax (i.e., the metamodel), which specifies the domain concepts and their relations. The abstract syntax is expressed in an object-oriented manner using Ecore [10], an implementation aligned with the meta-language MOF (*Meta Object Facility*)[11].

MOF provides language constructs for specifying a DSL metamodel: packages, classes, properties, multiple inheritance and different kinds of associations between classes. The semantics of these core object-oriented constructs is close to the object model common to various languages such as Java and C#.

To implement fUML's abstract syntax, we reuse the metamodel standardized by the OMG (cf. Figure 3 for an excerpt). In practice, the OMG provides the fUML metamodel in terms of MOF, and we automatically translate it into an Ecore-based metamodel (the format supported by the Kermeta workbench).

The static semantics of a DSL (i.e. the context conditions) corresponds to the well-formedness rules on top of the abstract syntax (expressed as invariants of metamodel classes) [4]. The static semantics is used to filter syntactically incorrect DSL models before actually running them. Kermeta supports OCL to express the static semantics[5] and it translates this semantics into equivalent Kermeta constraints, directly woven into the relevant metamodel classes using the Kermeta keyword `aspect`. Listing 1.1 shows the well-formedness rule previously introduced for fUML as expressed in the Kermeta workbench using OCL.

[5] Note that Kermeta fully support OCL, and thus similarly supports the axiomatic semantics (expressed as pre- and post-conditions on operations of metamodel classes). It is used to check the correctness of a DSL model's execution either at design time using model-checking or theorem proving, or at runtime using assertions, depending on the execution domain of the DSL.

Listing 1.1. Weaving the Static Semantics of fUML into the Standard Metamodel

```
1 package fuml;
2 require "fuml.ecore"
3 aspect class ControlFlow {
4   inv : self.source.oclIsKindOf(ControlNode) and
5         self.target.oclIsKindOf(ControlNode)
6 }
```

In Kermeta, the abstract syntax and the static semantics are conceptually and physically (at the file level) defined in two different modules. Consequently, it is possible to define several variants of the static semantics for the same domain, i.e. to share a single MOF metamodel between different static semantics.

3.2 Behavioral Semantics Definition

To define the behavioral semantics of a DSL, one must first define the required data structure (i.e., the execution model) using MOF. The abstract syntax and the execution model are then the basis to implement the behavioral semantics. Nevertheless, MOF does not include concepts for the definition of the behavioral semantics and OCL is a side-effect-free language. To define the behavioral semantics of a DSL, Kermeta provides an action language [5]. It can be used to define either a translational semantics (for building a compiler) or an operational semantics [12] (for building an interpreter).

The Kermeta language is imperative, statically typed, and includes classical control structures such as blocks, conditional statements, loops and exceptions. It also implements traditional object-oriented mechanisms for handling multiple inheritance and generics, and provides an execution semantics to all MOF constructs that must have a semantics at runtime, such as containment and associations. For example, if a reference is part of a bidirectional association, the assignment operator semantics has to handle both ends of the association. Kermeta also borrows the semantics of multiple inheritance from the Eiffel programming language [13].

Using the Kermeta language, the domain-specific actions are expressed as methods of the classes of the abstract syntax [5]. Similarly to the static semantics, the methods are added to the relevant metamodel classes (using the keyword `aspect`). Unlike the specification and the Java implementation that largely duplicate the structure of the abstract syntax to describe the structure of the visitor (see Section 2.2), aspects avoid duplicating the structure while keeping a conceptual and physical separation. For instance, in Listing 1.2, the method *execute* is added to the concept *CallBehavioralAction* of the fUML abstract syntax. This method is the Kermeta-based specification of the corresponding fUML behavioral semantics that consists in calling the behavior associated to the action.

Listing 1.2. Weaving the Behavioral Semantics of fUML into the Standard Metamodel

```
1 aspect class CallBehavioralAction {
2   operation execute() : Integer is do
3     result := self.behavior.call()  // call the associated behavior
4   end
5 }
```

Once the domain-specific actions have been described, it is necessary to describe their scheduling according to a particular model of computation. We describe in the next section the usual way to do this in Kermeta, and we discuss the drawbacks of this approach.

3.3 Mashup of the DSL Concerns

As introduced above, all pieces of static semantics and domain-specific actions are encapsulated in metamodel classes. The `aspect` keyword enables DSL designers to relate the language concerns (abstract syntax, static semantics, and domain-specific actions) together. It allows designers to reopen a previously created class to add some new information such as new methods, new properties or new constraints. It is inspired from open-classes [14].

In addition, Kermeta provides the keyword `require` that one uses to actually *mash up* those concerns. A DSL implementation *requires* an abstract syntax, a static semantics and the domain-specific actions. Listing 1.3 shows how such an implementation looks like in Kermeta. Three `require` keywords are used to import three modules, each of which specifies one of the three concerns. The *require* mechanism also provides some flexibility with respect to the static semantics and the domain-specific actions. For example, several sets of domain-specific actions could be defined in different modules and then chosen depending on particular needs. It is also convenient to support semantic variations of the same concept.

Listing 1.3. Mashup of the fUML Concerns

```
1  package fuml;
2  require "fuml.ecore" // abstract syntax
3  require "fuml.ocl"   // static semantics
4  require "fuml.kmt"   // domain-specific actions
5  class Main {
6    operation Main(): Void is do
7      // Scheduling calls to domain-specific actions
8      // to drive the execution of an fUML model
9    end
10 }
```

The Kermeta-based implementation of fUML follows the approach above. All fUML concerns are separated in different units, and the fUML runtime environment is the result of the mashup.

Finally, to implement the entire behavioral semantics, it is necessary to specify how the domain-specific actions are scheduled. One approach is to scatter the scheduling policy across all the methods defining the domain-specific actions in the visitor, as done in the specification and in the Java-based reference implementation. While this approach is easy to implement, it clearly prevents the modularization of the scheduling policy, which would be required to enable its variability. To avoid scattering the scheduling policy, another approach is to extract it in the main class that starts the execution of the model for a particular purpose, and therefore according to a particular MoC (see Listing 1.3). Although this approach of separating the MoC from the domain-specific actions allows the use of the same MoC for variants of the domain-specific actions, the MoC is

strongly coupled to the DSL and must be redefined from scratch for every new DSL. It is thus impossible to reuse or to adapt a MoC for different DSLs.

In the next section, we present how the MoC-based modeling framework called ModHel'X can be used to improve the aforementioned approach. This new approach paves the way for reusing the implementation of MoCs. Then, we introduce in Section 5 an approach to combine such a MoC with the domain-specific actions, enabling the reuse of a MoC in different domains, and the implementation of different MoCs for a specific domain.

4 Using ModHel'X Models of Computation for fUML

ModHel'X [6,15] is a framework for building and executing multi-paradigm models, that is to say models built from parts described using different modeling paradigms. In ModHel'X, the behavioral semantics of a modeling paradigm is given by the combination of two elements:

1. A *Model of Computation (MoC)*, which is a set of rules defining the semantics of control and concurrency, the semantics of communications and the semantics of time of the modeling paradigm. Synchronous Data-Flows (SDF), Discrete Events (DE), and Kahn Process Networks (KPN) [16] are examples of models of computation.
2. A library of components with predefined behavior. For instance, the component library of the synchronous data-flow MoC of ModHel'X includes components representing mathematic functions like addition, multiplication, etc. The behavior of those components correspond to the *domain-specific actions* introduced in Section 2.2.

Therefore, building a ModHel'X model is a two-step process: (1) choose components to assemble and (2) choose the MoC according to which the components interact. In the following, we present how MoCs and components are represented in ModHel'X to allow model execution. Then we will show how they can be used in the description of any DSL.

4.1 Generic Abstract Syntax

At the core of ModHel'X is a generic metamodel for describing the structure of models, and a generic execution engine for interpreting such structures using the semantics defined by MoCs. This means that all ModHel'X models have the same abstract syntax (given by the generic metamodel of ModHel'X) but each model may have a different execution semantics depending on the MoC which is used by the execution engine to interpret it. The fact that all models have the same abstract syntax is what allows ModHel'X to support the composition of models which have different semantics. The composition mechanism itself is not presented in this paper because it is not used yet in the presented methodology, but a detailed description can be found in [15].

Figure 4 shows a simplified excerpt of the metamodel of ModHel'X. To illustrate the concepts in this metamodel, we show in Figure 5 how the fUML model of Figure 2 would be described in ModHelX. An element of a ModHel'X model which has a behavior is a *block*, represented as a gray rectangle in Figure 5. Indeed, blocks are the mechanism used in ModHel'X to represent *domain-specific actions*. In the fUML example, ActivityNodes are represented by blocks because they all have a behavior (which can relate to control in the case of ControlNodes, or to executable behaviors in the case of ExecutableNodes).

Blocks communicate with their environment through *pins* (black circles in Figure 5), and the structure of a model is defined by establishing *relations* between the pins of blocks (the lines with arrows on the figure).

There are two different kinds of blocks in ModHel'X. *Interface blocks* are blocks whose behavior is described by an internal ModHel'X model. They are the mechanism we use for supporting heterogeneity through hierarchy (see [15] for more details). *Atomic blocks* are the basic building blocks which are the leaves of the hierarchy of models. For instance, the control nodes of fUML (join and fork in particular), would be atomic blocks in ModHel'X. While we provide libraries of atomic blocks for all the MoCs implemented in ModHel'X, we do not provide specific tools to allow users to define their own domain-specific atomic blocks because it is out of the scope of ModHel'X as a MoC-based experimentation platform for heterogeneous modeling. This means that the behavior of atomic blocks has to be described using a formalism which is external to our framework (for instance C or Java). Therefore, it is interesting for ModHel'X to benefit from techniques such as those provided by Kermeta to allow users to design and specify easily their own domain-specific atomic blocs, i.e. their own domain-specific actions. We will show in Section 5 how the approach proposed in this paper allows the use of Kermeta specifications of the behavior of the ActivityNodes of fUML in ModHel'X.

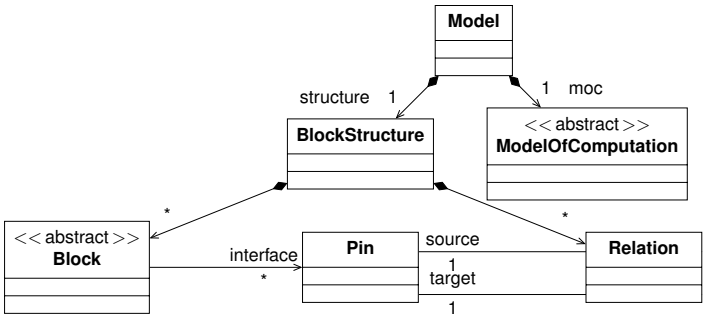

Fig. 4. Simplified excerpt of the generic metamodel of ModHel'X

4.2 Abstract Semantics for Models of Computation

As introduced previously, ModHel'X is dedicated to the execution of models. The execution of a model in ModHel'X is performed by the generic execution

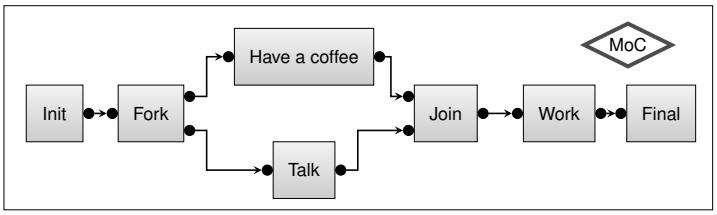

Fig. 5. fUML example model in ModHel'X

engine, which computes a series of observations, or *snapshots*. The MoC of a model is responsible for computing each snapshot of the model according to its specific semantics.

As illustrated by the sequence diagram in Figure 6, to compute a snapshot of a model, the MoC repeatedly chooses a block to observe (*schedule* operation), observes its behavior through its interface (*update*) and propagates observed information between blocks (*propagate*). It repeats this process until the computation of the reaction of the model is complete.

The schedule and propagate operations, as well as the rules for determining the stopping conditions of the algorithm, form the generic interface of MoCs. Together with the generic execution algorithm, they form the *abstract semantics* of ModHel'X. The scheduling and propagation operations must be specified as *MoC-specific actions* for each MoC (see Figure 7) because they are the implementation of the rules that define the control, concurrency, time and communication semantics of the corresponding modeling paradigm. The update operation is delegated to each block to provide the MoC with an observation of its behavior through its interface, while keeping the internal details in a complete black box.

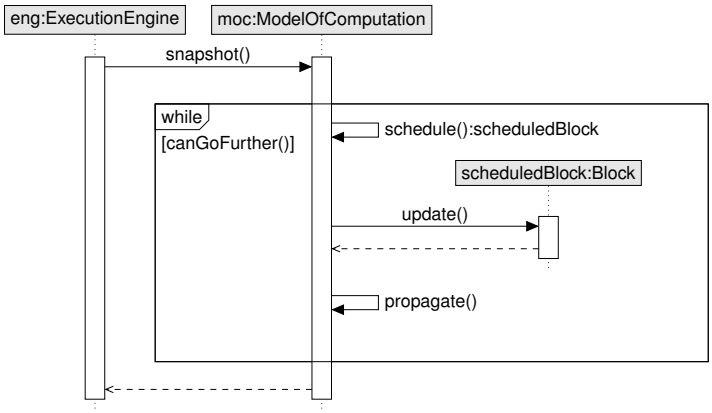

Fig. 6. Abstract semantics of ModHel'X

To implement the semantics of fUML in ModHel'X, we would have to choose or to build a MoC which implements the scheduling and propagation policies defined in the specification. Although the specification does not say much about time, concurrency, and inter-object communications, we know that Activity-

Nodes are linked by control and object flows on which tokens are propagated using the mechanism of offers. This is in favor of an event-based model of computation. Moreover, ExecutableNodes can represent the call of actions, so their behavior can take time. We could therefore choose to use the well-known Discrete Events (DE) model of computation [16], which already exists in ModHel'X. We present its semantics in the following section.

4.3 The Discrete Events (DE) MoC

DE is a MoC for the simulation of communicating processes, for instance hosts exchanging messages on a network. In DE, blocks exchange events at given dates. A block is observed when it receives an event from an upstream block or when it has spontaneous behavior. When observed, a block may produce outgoing events, to be transmitted to downstream blocks. If several events have the same timestamp, they are delivered at the same time, but in a sequence of *microsteps* (determined by a topological ordering of the blocks), so that the overall observation at that time is causal and deterministic.

The classical version of DE allows blocks to run concurrently. The scheduling algorithm for DE in ModHel'X relies on a global event queue. At a given instant, the MoC looks for all the events e_i with the smallest time tag t_{now} and advances the current time to t_{now}. It then looks for the blocks b_j which are the targets of the e_i events and schedules one of the minimal elements among the b_j according to the topological ordering of the blocks. The choice of a minimal element guarantees that events produced at t_{now} during the update of a block can be processed by their target at t_{now} in one iteration. A mechanism which is out of the scope of this paper is used to avoid cycles in the graph of blocks. The snapshot is complete when no event with time-stamp t_{now} remains in the queue.

The fUML specification leaves open the type of scheduling of the activity nodes: their execution may be concurrent or sequential. The classical version of DE, the "concurrent" one described above, may therefore be used as *one* possible scheduling for fUML, but a sequential variant of DE is also possible.

We have designed a "Sequential DE" MoC that has the following differences with the "Concurrent DE" MoC. At a given moment, only one block may be *active*. A block is said to be active if it has been given control (i.e. events have been provided to the block and the block has been observed), but it has not released control (i.e. it has not produced events yet). Sequential DE keeps track of the blocks that are *active-able*. If a block receives an event at t, then it is active-able starting at time t. But contrary to Concurrent DE that systematically and immediately activates all the active-able blocks, Sequential DE waits for the currently active block to release control (which will involve taking a new snapshot if the release is not immediate) before activating another active-able block.

The following section shows how the ModHel'X implementation of these two MoCs, "ConcurrentDE" and "SequentialDE", can be used to drive the execution of an fUML model in which the domain-specific actions are described in Kermeta.

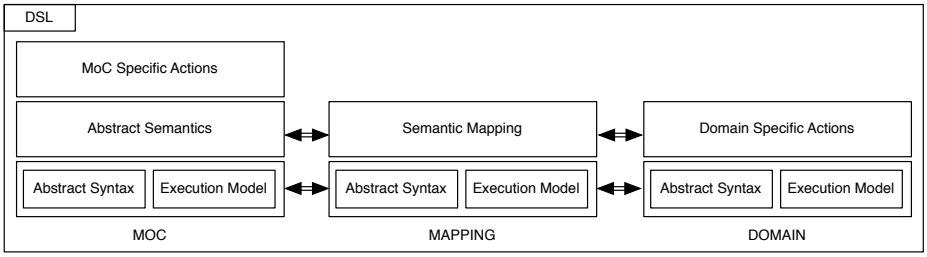

Fig. 7. Elements of the semantics of a DSL in our approach.

5 Bridging the Chasm

The previous sections have shown that:

- It is possible to easily define the abstract syntax and the execution model of a DSL, together with the semantics of its domain-specific actions, using Kermeta. However, a model of computation has to be written from scratch for each DSL in order to define the scheduling policy of these actions.
- ModHel'X offers various models of computation and provides means to define customized models of computation on top of an execution engine which allows the simulation of heterogeneous models. However, no specific tool is provided to help the user specify domain-specific actions which are represented as blocks in a ModHel'X model.

In this section, we now bridge the chasm between these two worlds in order to benefit from the advantages of both. As a result, we present a general methodology that allows us to *execute* models described with DSLs defined in a modular way. We present the application of this methodology to the fUML case study. We have also applied the methodology to the Software and Systems Process Engineering Metamodel specification of the OMG [17]. The corresponding experiments are available online at http://www.gemoc.org/kermeta-modhelx.

Our approach is based on the decomposition of a DSL semantics as shown in Figure 7. The structure of the MoC, on the left, and of the domain, on the right, have been presented in the previous sections. In the following sections, we show how the abstract syntax and execution models on both sides can be mapped, and how the abstract semantics of the MoC modeling framework (ModHel'X in our case), combined with the concrete execution semantics of the MoC, is used to schedule domain-specific actions in order to execute a model.

5.1 Abstract Syntax Mapping

First, the abstract syntax of the DSL is mapped onto the abstract syntax of ModHel'X, to enable model execution through the generic engine. In the case of fUML, the control structure and the activity nodes must be mapped onto ModHel'X elements. Activity nodes have domain-specific actions that must be callable, so they are naturally mapped onto atomic blocks (that can be observed

through the *update* operation). Control edges are mapped onto relations between blocks, which represent the possible flow of control.

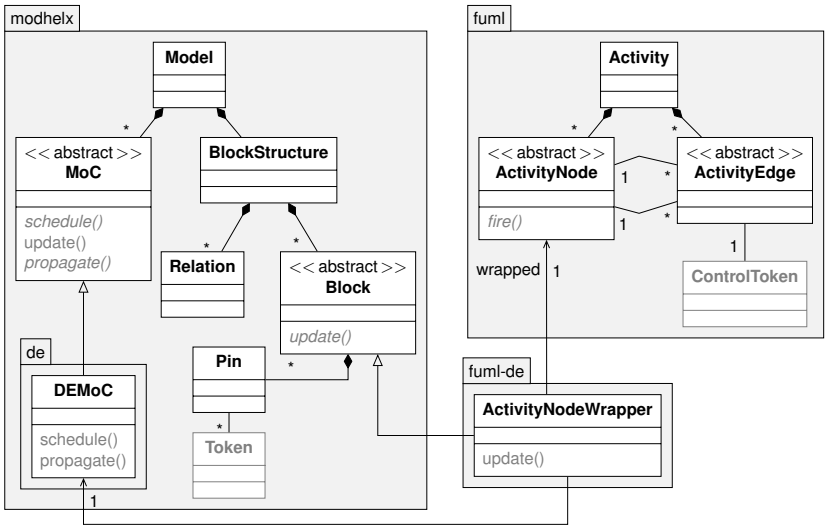

Fig. 8. Mapping (package `fuml-de`) between the kermeta-based implementation of fUML (package `fuml`) and ModHel'X (package `modhelx`) to use any of its MoCs (*e.g.,* here the discrete event MoC, package `de`)

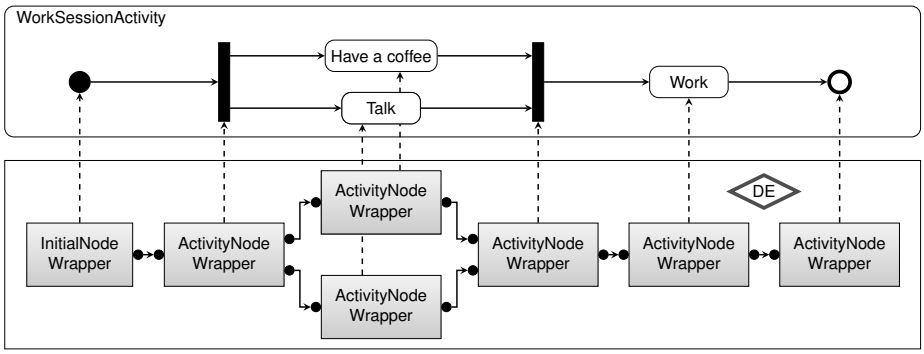

Fig. 9. Example fUML model and its wrapping ModHel'X model using DE.

Figure 8 shows the mapping between the two metamodels, and Figure 9 shows the result of the syntactic transformation of an fUML model into a ModHel'X model using the DE MoC. This model transformation is made before the execution starts, by instantiating a wrapper for each activity node. For fUML, we need two kinds of wrappers. *ActivityNodeWrapper* wraps *reactive* activity nodes, i.e. nodes that start a behavior, which possibly takes some time, when they are

given control. All activity nodes except *InitialNodes* are of this kind. However *InitialNodes* must *create control* at the start of the simulation, without receiving control explicitly. That is why they are wrapped into specific *InitialNodeWrappers*. Then, for each edge in the fUML model, we create a relation between the pins of the relevant blocks in the ModHel'X model. It must be noted that in this case, the mapping between activity nodes and blocks, and between control edges and relations is straightforward. However, in the general case, the structure of the ModHel'X model may be different from the structure of the domain-specific model.

5.2 Abstract Semantics to Domain Specific Actions Mapping

We must now map the abstract semantics of ModHel'X onto the domain-specific actions. The entry point of the abstract semantics for blocks is the *update* operation. Therefore, an activity node is wrapped into a special kind of block, which has an *update* operation that calls the domain-specific actions of the node. The wrapper acts as a block in the ModHel'X model, so its class is a subclass of *Block*. On the other hand it must execute the associated domain-specific actions, so it relies on the DSL's method signatures. Figure 10 shows how the wrapper maps the abstract semantics of ModHel'X onto the domain-specific semantics of fUML. When DE gives control to the wrapper block by calling its *update* method, the wrapper calls the domain-specific action (the *fire* operation). If the wrapped activity node is an action which takes time, the wrapper also requests to be observed in the future, so that it can handle the termination of the action.

The *schedule* and *propagate* operations allow the MoC to choose which block should be updated next, and how information produced by the update should be propagated to the other blocks.

5.3 Execution Model Mapping

The last item of Figure 7 to be mapped is the *execution model*, which represents the state of the execution of the model. The *update* operation of the wrapper synchronizes the execution models on both sides. In the case of the DE MoC and of fUML, DE events represent control on the MoC side, and must be translated into fUML control tokens before the domain-specific actions are called. When the fUML model has updated its execution model, control tokens must be converted into DE events so that the MoC has the necessary information to schedule the rest of the execution. Time must also be synchronized so that the MoC knows when to schedule a block, and activity nodes know when they terminate.

In the general case, the wrapper has to synchronize three aspects of the execution model: control, time and data. In this example, the DE/fUML wrapper adapts control and time only; we did not deal with the adaptation of data in this work.

One of the difficulties of the approach is to decide what to model in the MoC and what to model in the domain-specific actions. In order to favor the modularity and the reuse of the MoC for different DSLs, we decided to handle

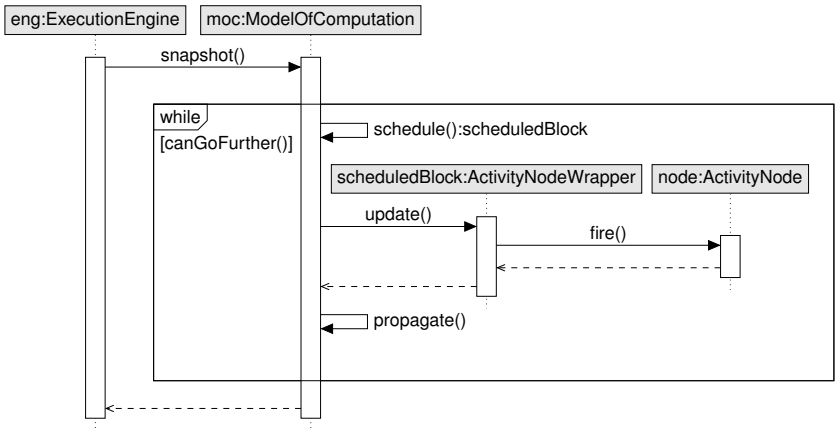

Fig. 10. Mapping fUML domain-specific semantics on ModHel'X abstract semantics

only the control and time aspects in the MoC and the wrapper. An example of such a design decision is the choice of whether to check in the MoC or in a domain-specific action if an activity node can be activated. Both can be done: *the wrapper* can be responsible for calling *fire* only when all the inputs of the block have received an event, or *fire* can be responsible for executing the activity only when all incoming control edges have a control token. We chose to implement the latter behavior in the *fire* domain-specific action even though this is related to control, because it is the *core semantics* of fUML that states that an activity node is executed only when it has received control on all its incoming edges.

6 Implementations and Execution Traces

We have experimented the approach proposed in this paper using Kermeta and ModHel'X to implement fUML[6]. Using our implementation, we have been able to execute the fUML WorkSessionActivity example, wrapped as shown in Figure 9. The following sections present the execution traces obtained using the classical "Concurrent" DE MoC, then its "Sequential DE" variant. To help differentiating the two executions, we have chosen different durations for the *Have a coffee* action (10 minutes), the *Talk* action (15 minutes) and the *Work* action (45 minutes). The execution traces obtained using our implementation are graphically depicted by the timing diagrams shown on Figure 11. Those diagrams illustrate the time at which the different actions respectively start and complete.

6.1 Using the Concurrent DE MoC

The execution obtained using the Concurrent DE MoC is illustrated by the timing diagram shown on the left part of Figure 11. With Concurrent DE, the

[6] The experiments are provided at http://www.gemoc.org/kermeta-modhelx

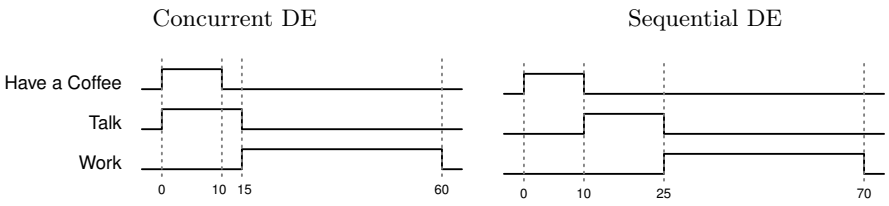

Fig. 11. Timing diagrams of the execution traces when the model is scheduled by different MoCs

two actions after the Fork start concurrently at $t = 0$, the beginning of the execution of the overall activity. So a first snapshot is taken at time $t = 0$, and we see on the timing diagram that both the *Have a coffee* and the *Talk* actions start. After that, two more snapshots are taken when each of the actions completes: at $t = 10$ for *Have a coffee*; at $t = 15$ for *Talk*. Within the latter snapshot, the Join is activated since the two preceding actions are finished and it releases control to the *Work* action, which therefore starts at $t = 15$. A last snapshot is taken when the *Work* action completes at $t = 15 + 45 = 60$.

6.2 Using the Sequential DE Variant

The execution obtained using the Sequential DE MoC is illustrated by the timing diagram shown on the right part of Figure 11. With Sequential DE, the two actions after the Fork become active-able at the initial time ($t = 0$). But since only one of them can be active at the same time, the MoC chooses to start one of them, for instance *Have a coffee*. So a first snapshot is taken at time $t = 0$. A second snapshot is taken when the *Have a coffee* action completes ($t = 10$). At that time, the *Talk* action can start. A third snapshot is taken when the *Talk* action completes ($t = 25 = 10 + 15$). During this snapshot, the Join is activated and it releases control to the *Work* action, which therefore starts at $t = 25$. A last snapshot is taken at $t = 25 + 45 = 70$, when the *Work* action completes.

As illustrated by the timing diagrams of Figure 11, we have managed to obtain two different executions of the same fUML model by changing the model of computation which is used to schedule it. This shows how the modular description of the semantics of DSLs as the association of a model of computation and a set of domain-specific actions facilitates the obtention of variants of a given DSL. In the following, we compare our approach to related work in the domains of modeling language engineering and MoC-based modeling.

7 Related Work

Much work have been done on the design and implementation of both software languages and models of computation. In this paper, we propose a conceptual and technical framework to bridge the chasm between them. This framework leverages experiences of both communities. This section presents related work in

the field of language design and implementation, and then in the field of models of computation.

The problem of the modular design of languages has been explored by several authors (e.g. [18,19]). For example, JastAdd [19] combines traditional use of higher order attribute grammars with object-orientation and simple aspect-orientation (static introductions) to get a better modularity mechanism. With a similar support for object-orientation and static introductions, Kermeta and its aspect paradigm can be seen as an analogue of JastAdd in the DSML world. Rebernak et al. [20] and Krahn et al. [21] contributed to the field in the context of model-driven DSLs. While they also advocate modularity of DSL compilers and interpreters, we go further: we take advantage of modularity mechanisms for integrating the body of knowledge on models of computation, and allow their reuse and variability.

A language workbench is a software package for designing software languages [22,23]. For instance, it may encompass parser generators, specialized editors, DSLs for expressing the semantics and others. Early language workbenches include Centaur [24], ASF+SDF [25], TXL [26] and Generic Model Environment (GME) [27]. Among more recent proposals, we can cite Metacase's MetaEdit+ [28], Microsoft's DSL Tools [29], Clark et al.'s Xactium [30], Krahn et al's Monticore [21], Kats and Visser's Spoofax [31], Jetbrain's MPS [32]. The important difference of our approach is that we explicitly address the MoC concern in the design of a language, providing a dedicated tooling for its implementation and reuse. Our approach is also 100% compatible with all EMF-based tools (at the code level, not only at the abstract syntax level provided by Ecore), hence designing a DSL with our approach easily allows reusing the rich ecosystem of Eclipse/EMF. This issue was previously addressed in the context of the Smalltalk ecosystem [33]. Our contribution brings in a much more lightweight approach using one dedicated meta-language per language design concern, and providing the user with advanced composition mechanisms to combine the concerns in a fully automated way.

In the context of component-based modeling, models of computation are used to define the interactions between the behavior of the components of a model. [34] describes several characteristics of models of computation, as well as a framework for comparing them.

Several approaches to the definition of models of computation have been proposed. Connector-based approaches like BIP [35] describe the interactions between behaviors using connectors, which can be considered as operators in a process algebra. From the properties of the connectors, it is possible to predict global properties of the models. In the case of BIP, the choice of connectors guarantees that the synthesized controller fires only interactions valid in the model. The result is therefore correct by construction.

The Clock Constraints Specification Language (CCSL) [36] can also be used to describe models of computation. It defines the semantics of the MARTE UML profile and it has been used for instance to model communication patterns in

AADL [37]. We also used it in previous work to describe models of computation and the interactions between heterogeneous models of computation [38].

However, these approaches describe how component behaviors are combined in an instance of a model. Other approaches like Ptolemy [16] and ModHel'X [39] allow the definition of models of computation independently of any model instance. Such definitions are therefore reusable for any model which obeys the abstract semantics of the framework. This abstract semantics defines a set of operations which drive the execution of models. Each model of computation provides concrete semantics to these abstract operations. The approach presented in this article relies on such reusable definitions of models of computation.

8 Conclusion and Perspectives

Although previous work has been done on the execution of UML models, as discussed in section 7, to the best of our knowledge, we introduce in this paper the first conceptual and technological bridge between executable metamodeling and models of computation at the level of the metamodels. We leverage on the experience of their respective fields and we provide an approach for a modular design and implementation of executable DSLs.

This approach includes a generic design pattern for metamodels bridging the gap between domain-specific actions woven into the metamodel and a reusable model of computation. We provide an actual implementation of this pattern, using Kermeta to weave executable actions into metamodels, and ModHel'X to schedule their execution according to a reusable MoC. The tools as well as the different fUML bridges presented in the paper can be freely downloaded on line.

Such a modular design and implementation of a behavioral semantics leverages on experience coming from two communities to achieve many expectations. As we illustrate with the fUML example coming from the OMG, many languages have variants of their model of computation, which current implementations do not take into consideration. Moreover, since the correct behavior of models is very dependent on the properties of their MoC, the design and implementation of a MoC can be critical. Being able to reuse validated MoCs, or validating an implementation of a MoC through reuse in various contexts is an advantage. Our approach addresses these two considerations by offering the reuse of MoCs between DSLs. The other way around, being able to reuse the domain-specific actions of a DSL with different MoCs in order to implement semantic variation points is also an advantage.

This first step to combine executable metamodeling and MoCs opens many exciting perspectives that we are proactively exploring. We first plan to examine very carefully the perimeter of the possible wrappers to propose suitable abstractions (*e.g.,* control, time, communication, etc) and patterns for their definition. Then, we would like to fully exploit the benefits coming from the two communities. In particular, we explore the application of this approach for heterogeneous executable modeling, taking advantage of the composition features supported by ModHel'X for multi-paradigm modeling.

References

1. Karna, J., Tolvanen, J.P., Kelly, S.: Evaluating the use of Domain-Specific Modeling in Practice. In: 9th OOPSLA workshop on Domain-Specific Modeling. (2009)
2. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: ICSE), ACM (2011) 471–480
3. Bryant, B.R., Gray, J., Mernik, M., Clarke, P.J., France, R.B., Karsai, G.: Challenges and directions in formalizing the semantics of modeling languages. Comput. Sci. Inf. Syst. **8**(2) (2011) 225–253
4. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? Computer **37**(10) (2004) 64–72
5. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving Executability into Object-Oriented Meta-Languages. In: MoDELS. Volume 3713 of LNCS., Springer (2005) 264–278
6. Boulanger, F., Hardebolle, C.: Simulation of Multi-Formalism Models with ModHel'X. In: Proceedings of ICST'08, IEEE Comp. Soc. (2008) 318–327
7. Object Management Group, Inc.: Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0. (2011)
8. Object Management Group, Inc.: UML Object Constraint Language (OCL) 2.0 Specification. (2003)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)
10. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley (2008)
11. Object Management Group, Inc.: Meta Object Facility (MOF) 2.0 Core Specification. (2006)
12. Combemale, B., Crégut, X., Garoche, P.L., Thirioux, X.: Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. Journal of Software **4**(9) (2009)
13. Meyer, B.: Eiffel: the language. Prentice-Hall, Inc. (1992)
14. Clifton, C., Leavens, G.T.: MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In: OOPSLA. (2000) 130–145
15. Boulanger, F., Hardebolle, C., Jacquet, C., Marcadet, D.: Semantic Adaptation for Models of Computation. In: ACSD. (2011) 153–162
16. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity – the Ptolemy approach. Proc. of the IEEE **91**(1) (2003) 127–144
17. Object Management Group, Inc.: Software and Systems Process Engineering Metamodel specification (SPEM) Version 2.0. (2008)
18. Wyk, E.V., Moor, O.d., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: CC'02, Springer-Verlag (2002) 128–142
19. Ekman, T., Hedin, G.: The JastAdd system – modular extensible compiler construction. Sci. Comput. Program. **69** (2007) 14–26
20. Rebernak, D., Mernik, M., Wu, H., Gray, J.: Domain-specific aspect languages for modularising crosscutting concerns in grammars. IET Software **3**(3) (2009) 184–200
21. Krahn, H., Rumpe, B., Volkel, S.: MontiCore: Modular Development of Textual Domain Specific Languages. In: Objects, Components, Models and Patterns. Volume 11 of LNBIP., Springer (2008) 297–315

22. Fowler, M.: Language workbenches: The killer-app for domain specific languages. Accessed online from: http://www.martinfowler.com/articles/languageWorkbench.html (2005)
23. Volter, M.: From Programming to Modeling-and Back Again. Software, IEEE **28**(6) (2011) 20–25
24. Borras, P., Clement, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., Pascual, V.: Centaur: the system. In: 3rd ACM software engineering symposium on Practical software development environments, ACM (1988) 14–24
25. Klint, P.: A meta-environment for generating programming environments. ACM TOSEM **2**(2) (1993) 176–201
26. Cordy, J.R., Halpern, C.D., Promislow, E.: TXL: a rapid prototyping system for programming language dialects. In: Conf. Int Computer Languages. (1988) 280–285
27. Sztipanovits, J., Karsai, G.: Model-Integrated Computing. IEEE Computer **30**(4) (1997) 110–111
28. Tolvanen, J., Rossi, M.: MetaEdit+: defining and using domain-specific modeling languages and code generators. In: Companion of the 18th annual ACM SIGPLAN conference OOPSLA, ACM (2003) 92–93
29. Cook, S., Jones, G., Kent, S., Wills, A.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional (2007)
30. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling – A Foundation for Language Driven Development. Second Edition (2008)
31. Kats, L.C., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and IDEs. In: OOPSLA '10, ACM (2010) 444–463
32. Voelter, M., Solomatov, K.: Language Modularization and Composition with Projectional Language Workbenches illustrated with MPS. In: SLE'10. LNCS, Springer (2010)
33. Renggli, L., Gîrba, T., Nierstrasz, O.: Embedding Languages without Breaking Tools. In: ECOOP. Volume 6183 of LNCS., Springer (2010) 380–404
34. Lee, E.A., Sangiovanni-Vincentelli, A.L.: A framework for comparing models of computation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **17**(12) (1998) 1217–1229
35. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time systems in BIP. In: 4th IEEE SEFM. (September 2006) 3–12
36. Mallet, F., DeAntoni, J., André, C., de Simone, R.: The clock constraint specification language for building timed causality models. Innovations in Systems and Software Engineering **6** (2010) 99–106
37. André, C., Mallet, F., Simone, R.: Modeling AADL Data Communications with UML MARTE. In: Embedded Systems Specification and Design Languages. Volume 10 of Lecture Notes in Electrical Engineering. Springer (2008) 155–168
38. Boulanger, F., Dogui, A., Hardebolle, C., Jacquet, C., Marcadet, D., Prodan, I.: Semantic Adaptation Using CCSL Clock Constraints. In: Workshops and Symposia at MODELS 2011. Volume 7167 of LNCS., Springer-Verlag (2012) 104–118
39. Hardebolle, C., Boulanger, F.: Multi-Formalism Modelling and Model Execution. International Journal of Computers and their Applications **31**(3) (July 2009) 193–203 Special Issue on the International Summer School on Software Engineering.