A DSL for Explicit Semantic Adaptation *

Bart Meyers[1], Joachim Denil[3], Frédéric Boulanger[2],
Cécile Hardebolle[2], Christophe Jacquet[2], and Hans Vangheluwe[1,3]

[1] MSDL, Department of Mathematics and Computer Science
University of Antwerp, Belgium
[2] Supélec E3S, Department of Computer Science, France
[3] MSDL, School of Computer Science, McGill University, Canada

Abstract. In the domain of heterogeneous model composition, semantic adaptation is the "glue" that is necessary to assemble heterogeneous models so that the resulting composed model has well-defined semantics. In this paper, we present an execution model for a semantic adaptation interface between heterogeneous models. We introduce a Domain-Specific Language (DSL) for specifying such an interface explicitly using rules, and a transformation toward the ModHel'X framework. The DSL enables the modeller to easily customise interfaces that fit the heterogeneous model at hand in a modular way, as involved models are left untouched. We illustrate the use of our DSL on a power window case study and demonstrate the importance of defining semantic adaptation explicitly by comparing with the results obtained with Ptolemy II.

1 Introduction

The growing power of modelling tools allows the design and verification of complex systems. This complexity leads to Multi-Paradigm Modelling [1], because different parts of the system belong to different technical domains, but also because different abstraction levels, different aspects of the system, and different phases in the design require different modelling techniques and tools. It is therefore necessary to be able to cope with multi-paradigm, heterogeneous models, and to give them a semantics which is precise enough for simulating the behaviour and validating properties of the system, and to generate as much as possible of the realisation of the system from the model.

A major challenge in model composition is to obtain a meaningful result from models with heterogeneous semantics. Tackling this issue requires that the semantics of the modelling languages used in the composed models is described in a way such that it can be processed and, more importantly, such that it can be composed. In this article, we focus on the definition of the "composition laws" for heterogeneous parts of a model, which have to be explicitly described. A key point of our approach is modularity: we should be able to compose heterogeneous models without modifying them.

To illustrate what these composition laws, which we call *semantic adaptation*, have to deal with, we first introduce the example of a power window and

* Thanks to Dr. Thomas Feng for giving us insights into Ptolemy II's discrete events semantics

show how it can be modelled and simulated using existing tools for heterogeneous modelling. We introduce a meta-model for modelling semantic adaptation, founded on an execution model. We then present a Domain-Specific Language (DSL) that we designed to specify semantic adaptation in an abstract and compact way. We then discuss the advantages and drawbacks of this approach and conclude, giving some perspective for future work.

2 The Power Window Case Study

The system we model to illustrate our work is a car power window, composed of a switch, a controller board and an electromechanical part. These components communicate through the car's bus. The controller receives commands from the switch and information from the end stops of the electromechanical part, and sends commands to switch the motor off, up or down. If the user presses the switch for a short time (less than 500 ms), the controller fully closes or opens the window until it reaches an end stop.

For simplicity, we don't model the bus as a component of the system, and we consider that the components of the model communicate through discrete events (DE). The controller is modelled as a state machine, with timed transitions to distinguish between short and long presses on the switch. The dynamics of the electromechanical part could be modelled using differential equations, but for making the example easier to understand, we discretise its behaviour and represent it using difference equations, for which a synchronous dataflow model (SDF) is suitable. As a result, the global model of the power window system is a heterogeneous model involving three different modelling paradigms: discrete events, timed finite state machines and synchronous dataflow.

In the following, we show how this system is modelled using different tools for heterogeneous modelling and we illustrate how these different tools deal with the necessary semantic adaptation among the heterogeneous parts of this model.

3 Related Work and Semantic Adaptation

We chose to focus our study of the state of the art on three different tools for heterogeneous modelling and simulation: Ptolemy II [2], Simulink/Stateflow[4] and ModHel'X [3]. All of them support the joint use of different modelling paradigms in a model and they all use hierarchy as a mechanism for composing the heterogeneous parts of a model. Other types of approaches are described in [4].

In Simulink/Stateflow, the power window system has been studied at different levels of detail and the resulting models are available as demos in the tool[5]. We have modelled the power window system using Ptolemy II and ModHel'X, and the resulting models are available online[6].

[4] `http://www.mathworks.fr/products/simulink/`

[5] See the online documentation at `http://www.mathworks.fr/fr/help/simulink/examples/simulink-power-window-controller-specification.html`.

[6] Ptolemy II model: `http://wwwdi.supelec.fr/software/downloads/ModHelX/power_window_fulladapt.moml`
ModHel'X model: `http://wwwdi.supelec.fr/software/ModHelX/PowerWindow`

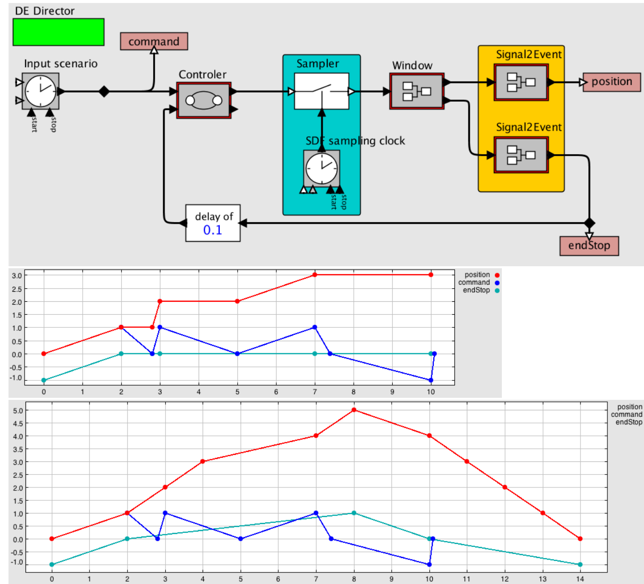

Fig. 1. Ptolemy II model of the power window

Using Ptolemy II for modelling the power window system, we obtain the model shown on Figure 1. In this model, the DE Director box tells that this model is built according to the Discrete Events *model of computation*, or MoC. A MoC defines the execution rules obeyed by the components of a model. The DE MoC considers each component as a process triggered by input events and producing events on the output. The Input scenario block models a simulation scenario for the output of the switch. The controller is modelled as a state machine (FSM MoC), and the window as a data-flow block diagram (Synchronous Data Flow MoC). Due to space constraints, these models are not shown in the paper.

Ptolemy II does not provide a default semantic adaptation between heterogeneous parts of a model. For instance, the Window component, which is modelled using SDF, is considered as a DE component by the DE director. It is therefore activated each time it receives an input event, and each data sample it produces is considered as a DE event on its outputs. This model corresponds to what is shown in Figure 1 without the two large rounded boxes (which deal with semantic adaptation and will be presented below). The result of the simulation without semantic adaptation (i.e., without the two large rounded boxes) is shown just below, in the middle part of the figure. The red line on top shows the position of the window (from 0 to 5), the green line at the bottom shows the end-stop signal (-1 is open, 1 is closed, 0 is in between), and the middle, blue line shows the command played by the scenario (1 is for closing the window, -1 for opening it, and 0 for stopping it). However, the behaviour is not as expected: the window model (using difference equations) is not sampled periodically, meaning that the window is not going up or down with a constant speed, and the model produces consecutive events with the same value.

The problem is that the SDF nature of the model of the window makes it behave inconsistently in a DE environment. Semantic adaptation is needed to sample it periodically, to provide it with correct inputs (as achieved with the left blue box) and to filter its outputs (as achieved with the right orange box). The lowest graph in Figure 1 shows the result with the adaptations. In this case the behaviour is as expected. The window model is sampled at a periodic rate and does not produce duplicate events.

The two highlighted areas were added to the DE model to specify the semantic adaptation with the SDF model of the behaviour of the window. However, their contents depends on the internals of the window model, so the global model is not modular. Moreover, without the highlighting we put on the figure, it is not easy to make a difference between a block which is part of the model and a block which is there for semantic adaptation. Semantic adaptation is therefore diffuse and difficult to specify and understand.

In Simulink/Stateflow, which is a powerful and efficient simulation tool for heterogeneous models, semantic adaptation is even more diffuse. The global semantics of a heterogeneous model (for instance a Simulink model including a Stateflow submodel) is given by one solver which is used to compute its execution. The solver uses parameters of the blocks and their ports, in particular a parameter called "sample time"[7], to determine when to compute the value of the different signals. The resulting execution of the model depends on the value of these parameters and on the type and parameters of the solver itself. Even if default values for these parameters are determined by the simulation engine, to adapt the resulting execution semantics to his needs, the user must have a deep understanding of the solver's mechanisms and fine tune different simulation parameters. A part of the semantic adaptation can also be performed using blocks like in Ptolemy II or truth tables or functions, but with the same drawbacks.

ModHel'X is a framework for modelling and executing heterogeneous models [3] which we developed in previous works. ModHel'X allows for modular semantic adaptation by explicitly defining an *interface block* between the parent model and the embedded model. This block adapts the data, control and time between the different models [5]. In the power window example, we saw examples of the adaptation of data with the filtering of the output signals of the window model, and of control with the periodic sampling. The adaptation of time is necessary when two models use different time scales. Interface blocks are similar to tag conversion actors as presented in [9], but can perform more complex operations. In ModHel'X, control and time adaptation rely on a model of time which is inspired from the MARTE UML profile [6, 7], and can be considered as a restriction of the Clock Constraints Specification Language [7], extended with time tags [8]. An issue with ModHel'X is that the semantic adaptation is specified using calls to a Java API in different methods of an interface block. Therefore, constructing an interface block is a tedious and error prone process. To aid developers in defining interface blocks, we propose a Domain Specific Language for modelling explicitly the adaptation of data, time and control.

[7] http://www.mathworks.fr/fr/help/simulink/ug/types-of-sample-time.html

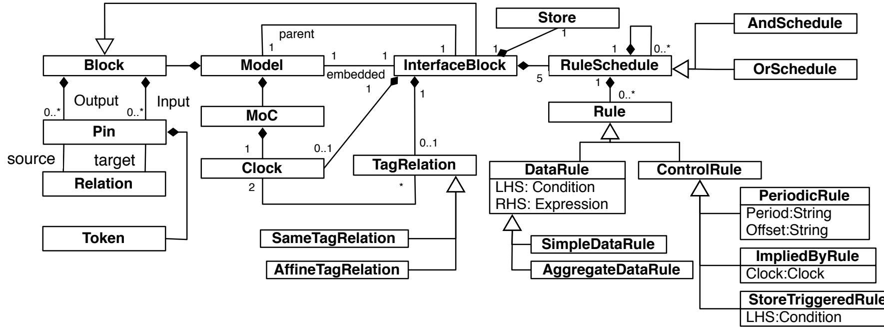

Fig. 2. Partial meta-model of ModHel'X with a focus on the interface block

4 A DSL and Execution Semantics for Semantic Adaptation

The DSL for semantic adaptation provides the modeller with the concepts that are necessary for specifying the glue between heterogeneous models. By insulating the modeller from platform specific issues, it allows him to focus on the problem at stake, and it also reduces the semantic gap between what should be specified and how it is realised. However, the current implementation of the DSL generates code for ModHel'X, so we have to give some background about it.

4.1 The Interface Block Meta-Model

Figure 2 shows a simplified, partial meta-model of ModHel'X with a focus on the interface block. The structure of models in ModHel'X is similar to Ptolemy II's: Models contain Blocks with input and output Pins that can be connected. During execution, values flow through models as Tokens on Pins. A Model has a model of computation (MoC) which defines its execution semantics, similar to a director in Ptolemy II. In ModHel'X a Block can be an InterfaceBlock, containing an embedded Model with any MoC, thus allowing heterogeneous modelling. In an InterfaceBlock, there can be adaptation Relations between input Pins of the InterfaceBlock and input Pins of the embedded Model, and between output Pins of the embedded Model and output Pins of the InterfaceBlock. An InterfaceBlock and a MoC can have a Clock, with its own time scale. An InterfaceBlock has a Store that can hold data, thus adding the concept of state.

Our DSL for describing the behaviour of an interface block is based on rules, which are evaluated and return true when they match. DataRules and ControlRules are used to match conditions for the semantic adaptation of data and control. The order in which rules are evaluated is controlled by a RuleSchedule. Two different scheduling policies (which are hierarchical) are defined: the AndSchedule evaluates the rules in order as long as they return true, and it returns true if all rules returned true; the OrSchedule evaluates the rules in order until one returns true, in which case true is returned.

Data rules are responsible for the adaptation of the data from the parent model to the embedded model and vice versa. Data rules have a left-hand side (LHS), denoting pre-condition for the applicability of the rule and a right-hand side (RHS) providing the needed action when the LHS is matched. Data rules

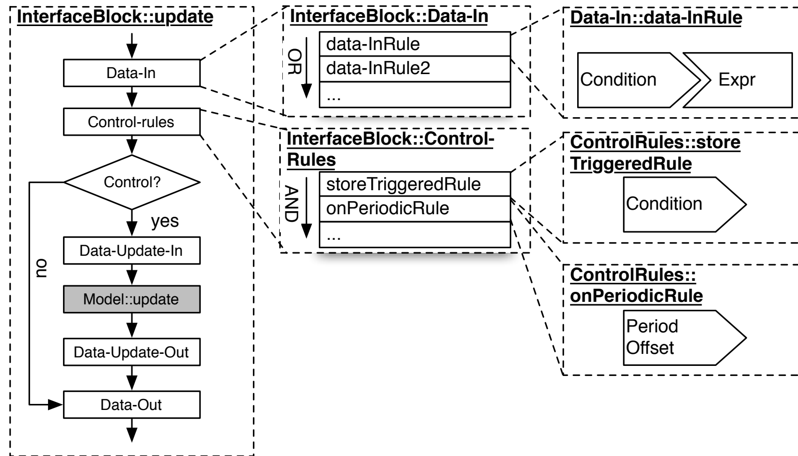

Fig. 3. Execution of an interface block

have access to pins and a store of the interface block which serves as a map of variables (variable name - value pairs). Control rules are responsible for executing the embedded model (also called *giving control*) at the current time instant. Control rules have access to the store to create complex trigger events but they also access the interface block clock to create an observation request in the future. For the semantic adaptation of time, an interface block can contain TagRelations between clocks, which allows to convert dates between their time scales.

4.2 Execution Semantics in Five Phases

As shown in Figure 3, the interface block adapts data and control in five different phases. Each phase has its own schedule (containing a set of rules), and each phase is provided with a set of pins as parameters so that, besides the store, token values on pins become accessible in the rules. In case of the Data-in, Control-rules and Data-update-in phases, parameter pins are the input pins of the interface block. In case of the Data-update-out or Data-out phases the parameter pins are the output pins of the embedded model. The phases are executed in the following order when control has been given to the interface block:

Data-in: (data rules) when the interface block receives control, rules are executed to update the store of the block according to the incoming data tokens. Complex operations can transform the data for further processing;

Control: (control rules) Control rules decide if control should be passed to the embedded model at the current instant according to the store of the block and the data on the input pins. Control rules can also create new observation request on the clock of the interface block so that it receives control later;

Data-update-in: (data rules) if the control rules give control to the embedded model, the Data-update-in schedule is executed before control is passed to the embedded model. These rules provide the internal model with correct inputs according to the values in the store and the data on the input pins of the interface block;

Data-update-out: (data rules) after the execution of the embedded model, the Data-update-out rules are used to update the store with the data available on the output pins of the embedded model;

Data-out: (data rules) finally, the Data-out rules are in charge of producing the outputs of the interface block from the data available in the store. This phase is executed even when control is not given to the embedded model because the interface block may have to produce some outputs any way.

To ease the creation of rules, we identified some common rule patterns, shown as subclasses of DataRule and ControlRule in Figure 2:

- SimpleDataRule: this data rule is evaluated for every parameter pin. If its LHS evaluates to true, its RHS is executed;
- AggregateDataRule: this data rule allows for more complex patterns over multiple parameter pins, and will only be evaluated once. If its LHS evaluates to true, its RHS is executed;
- Periodic: this control rule periodically gives control to the interface block, and variable names are given for period and offset so that they can be set when the interface block is used in a model. The clock calculus in ModHel'X will make sure that the interface block is updated at the specified moments;
- StoreTriggeredRule: this control rule gives control to the embedded model if its LHS (with access to the store) evaluates to true;
- ImpliedByRule: this control rule gives control to the embedded model whenever a given clock is triggered.

A DSL implementing this meta-model, the presented semantics and a concrete textual syntax are defined in metaDepth [10]. The solution includes an ANTLR-based parser, an EOL script for generating code for the rules, and an EGL script [11] for generating Java code for ModHel'X. These steps ensure that the transformation from DSL code to Java code is entirely automated. Due to space constraints, the meta-model, models or scripts in metaDepth are not shown.[8] The concrete syntax of the DSL will be illustrated in the next section.

5 A Semantic Adaptation DSL in Practice

In this section, we reconstruct the two different semantic adaptations between DE and SDF for the power window system presented in section 3, and of which traces are shown in Figure 1. The models presented in this section are transformed using the metaDepth framework into ModHel'X models in Java code.

Listing 1 and 2 show interface blocks for both behaviours as a DSL model. The first 8 lines state the structural properties of the interface block and are the same for both behaviours. Line 1 presents the model name. On line 2-3, the Java class and package are specified for the code generator. Line 4 presents the clock of the interface. In this case the interface has a timed clock, meaning that events occur at specific dates on a time scale (as opposed to an untimed clock, where events have no date). Line 5-6 and 7-8 respectively present the external (parent) model and internal (embedded) model. The parent model is called `de`

[8] The fully operational solution presented in this paper can however be downloaded from `http://msdl.cs.mcgill.ca/people/bart/PowerWindow.zip`

and adheres the ModHel'X `AbstractDEMoC` with a timed clock named `deClock`. Similarly, the embedded model uses SDF, which is by nature untimed.

Listing 1. Model of the default Ptolemy II behaviour

```
1  InterfaceBlock {
2    IMPLEMENTS "DE_SDF_InterfaceBlock"
3            IN "tests.powerwindow"
4    TIMED CLOCK ibClock
5    EXTERNAL MODEL de (
6      "AbstractDEMoC" WITH TIMED CLOCK deClock)
7    INTERNAL MODEL sdf (
8      "SDFMoC" WITH UNTIMED CLOCK sdfClock)
9    RULES:
10     IN
11     CONTROL
12     UPDATEIN
13       FORALLPINS ALWAYS ->
14                    FORWARD VALUE TO SUCCESSORS;
15     UPDATEOUT
16       FORALLPINS ALWAYS ->
17                    FORWARD VALUE TO SUCCESSORS;
18     OUT
19 }
```

Listing 2. Model of the interface block with semantic adaptation

```
1  InterfaceBlock {
2    IMPLEMENTS "DE_SDF_InterfaceBlock"
3            IN "tests.powerwindow"
4    TIMED CLOCK ibClock
5    EXTERNAL MODEL de (
6      "AbstractDEMoC" WITH TIMED CLOCK deClock)
7    INTERNAL MODEL sdf (
8      "SDFMoC" WITH UNTIMED CLOCK sdfClock)
9    RULES:
10     IN
11       FORALLPINS ON DATA: ALWAYS -> TOSTORE(VALUE);
12     CONTROL
13       PERIODIC AT "init_time" EVERY "period"
14     UPDATEIN
15       FORALLPINS ALWAYS ->
16                    FORWARD FROMSTORE TO SUCCESSORS;
17     UPDATEOUT
18       FORALLPINS ON DATA:
19         VALUE != FROMSTORE ->
20                    TOSTORE(VALUE);
21                    FORWARD VALUE TO SUCCESSORS;
22     OUT
23   TAG RELATION deClock = ibClock
24 }
```

From line 9 onward, the rules that model the execution semantics are specified. These differ in both models. The five rule phases, in their execution order, can be recognised from line 10 onward.

The default Ptolemy II behaviour that results in the upper trace of Figure 1 does little semantic adaptation. This behaviour is defined in Listing 1. No particular Data-in rules are given, since the store is not used. There is no Control rule, because whenever the interface block receives control, it will immediately give control to the embedded model. On Data-update-in, the tokens on the input ports are forwarded with no adaptation to the embedded model. This is modelled as the data rule on line 13-14, which should be read as $LHS \rightarrow RHS$. As a SimpleDataRule (denoted by `FORALLPINS`), it is executed for every input pin of the interface block. The LHS is `ALWAYS`, as there are no restrictions on when to forward data. The RHS specifies that the current token value (`VALUE`) should be forwarded to the embedded model's input pins to which the interface input pins are connected to (`SUCCESSORS`). In the Data-update-out phase on line 16-17, the tokens generated by the embedded model are similarly forwarded in another SimpleDataRule. As mentioned before, in this phase `VALUE` and `SUCCESSORS` – which depend on the parameter pins – refer to the token values and the successors of the embedded model's output pins. In summary, the basic Ptolemy model forwards all data instantaneously.

The intended behaviour of the power window's interface that results in the bottom trace of Figure 1 is shown in Listing 2. The interface block periodically samples the SDF model. Data is provided to the embedded model by applying zero-order hold (ZOH) from the input pins of the parent model to the input

pins of the embedded model, meaning that the last known value should be used. The Data-in rule on line 11 saves the data from the input pins in the store, each time a token is received (`ON DATA`). The Periodic Control rule on line 13 specifies the periodic sampling of the embedded model, starting at time "init_time", with a step of "period", which can be set in the instance model. The Data-update-in rule on line 15-16 implements the ZOH functionality by providing the input pins of the embedded model with data from the store. Note that first-order hold (linear extrapolation) could also be modelled by calculating new values based on the previous ones (stored the store) every time the interface block is updated. A Data-update-out rule (line 18-21) checks for every pin whether the embedded model's output value changed, by comparing it to the previous one, which was stored. Initially the value in the store is *null* making sure the LHS evaluates to true. So only when the output of the embedded model changes, tokens are forwarded to the parent model. No event is consequently produced when the window is inactive, as seen in the trace. Finally a SameTagRelation on line 23 specifies that time is measured the same way in the interface block and in the embedding model. Note that in both models we never had to explicitly schedule rules, as every phase contained at most a single rule.

6 Discussion

Our DSL for semantic adaptation allows one to *explicitly* specify the adaptation behaviour intended at the boundary between two heterogeneous parts of a model. This is an improvement in two respects. First, compared to the *implicit* adaptation mechanisms embedded in Ptolemy or Simulink, the semantics is clearly stated. Second, compared to the solution introduced in Section 3 that relies on the modification of the models to introduce new blocks devoted to adaptation, our approach leaves the models unchanged. This improves modularity, as a given model may be embedded as is in different contexts; what needs to be changed each time is just the adaptation layer defined using our DSL.

When embedding a timed finite state machine (TFSM) into a DE model in Ptolemy, one has to weave adaptation mechanisms in the guards and actions of the state machine in order to translate between TFSM events and DE values. Even if this does not introduce new blocks, this is indeed a variant of the second issue outlined above: one has to modify the model to include adaptation, which hinders modularity. We used the proposed DSL to model the adaptation of the DE model to the TFSM model of the power window system. The DSL is able to model all the needed constructs for the interface block and generate Java code. Due to space constraints, the model is not included in the paper. Still, some constructs necessary for the adaptation of a wide variety of MoCs may not be available in the DSL yet. Future enhancements of our DSL will remedy this.

The improvements mentioned here (specifying adaptation between models explicitly while keeping the modularity of models) have been available in ModHel'X for a few years. However in ModHel'X adaptation is coded in Java, so there is a semantic gap between high-level adaptation mechanisms and the way it is actually written as low-level code that makes minute manipulations of data structures. Moreover, an adapter in ModHel'X is scattered in several methods,

which makes its behavior hard to follow. The five phased execution semantics and the DSL are focused and concise: they bridge this semantic gap. More than 300 lines of Java code are reduced to less than 25 lines of DSL code.

7 Conclusion

In the domain of Multi-Paradigm Modelling, semantic adaptation is the "glue" that gives well-defined semantics to the composition of heterogeneous models. In this paper we proposed a DSL with execution semantics to bridge the cognitive gap between the implementation and specification of a semantic interface block. The model of an interface block explicitly specifies the adaptation of data, control and time using a set of rules. Furthermore, a model-to-text transformation is defined to generate the interface block code for the ModHel'X framework. The DSL enables the modeller to easily customise interface blocks that fit the heterogeneous models in a modular way, as involved models are left untouched.

The approach was illustrated on the power window case study by reconstructing in ModHel'X both the implicit Ptolemy II adaptation and the specific one needed between DE and SDF to obtain the expected behaviour. We believe that the DSL and the execution semantics are, as a principle, expressive enough to model different semantic adaptations of heterogeneous models, though additional constructs might be needed for additional behaviours.

References

1. P. J. Mosterman and H. Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. *SIMULATION*, 80(9):433–450, 2004.
2. J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming Heterogeneity - The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.
3. Frédéric Boulanger and Cécile Hardebolle. Simulation of Multi-Formalism Models with ModHel'X. In *ICST 2008*, pages 318–327, 2008.
4. Cécile Hardebolle and Frédéric Boulanger. Exploring multi-paradigm modeling techniques. *SIMULATION* , 85(11/12):688–708, 2009.
5. F. Boulanger, C. Hardebolle, C. Jacquet, and D. Marcadet. Semantic Adaptation for Models of Computation. In *ACSD'11*, pages 153–162, 2011.
6. Charles André, Frédéric Mallet, and Robert De Simone. Modeling Time(s). In *MoDELS'07*, volume LNCS 4735, pages pp. 559–573. Springer, 2007.
7. R. Gascon, F. Mallet, and J. Deantoni. Logical time and temporal logics: comparing UML MARTE/CCSL and PSL. In *TIME'11*, pages 141–148, Lubeck, Germany, 2011.
8. F. Boulanger, C. Hardebolle, C. Jacquet, and I. Prodan. Modeling time for the execution of heterogeneous models. Technical report 2013-09-03-DI-FBO, Supélec E3S, 2012.
9. P. Caspi, A. Benveniste, R. Lublinerman, and S. Tripakis. Actors without directors: A kahnian view of heterogeneous systems. In R. Majumdar and P. Tabuada, editors, *Hybrid Systems: Computation and Control*, volume 5469 of *LNCS*, pages 46–60. Springer Berlin Heidelberg, 2009.
10. Juan de Lara and Esther Guerra. Deep meta-modelling with metadepth. In *TOOLS (48)*, volume 6141 of *LNCS*. Springer Berlin Heidelberg, 2010.
11. Juan de Lara and Esther Guerra. Domain-specific textual meta-modelling languages for model driven engineering. In *ECMFA'12*, pages 259–274, 2012.