Semantic Adaptation using CCSL Clock Constraints

Frédéric Boulanger, Ayman Dogui, Cécile Hardebolle
Christophe Jacquet, Dominique Marcadet, Iuliana Prodan

Supelec Systems Sciences (E3S)
Computer Science Department
Gif-sur-Yvette, France
<firstname>.<lastname>@supelec.fr

Abstract: When different parts of a system depend on different technical domains, the best suitable paradigm for modeling each part may differ. In this paper, we focus on the semantic adaptation between parts of a model which use different modeling paradigms in the context of model composition. We show how CCSL, a language for defining constraints and relations on clocks, can be used to define this semantic adaptation in a formal and modular way.

Keywords: Heterogeneous Modeling, Semantic Adaptation, Clock Calculus

1 Introduction

Models are the primary way of handling complexity by providing abstract representations of a system, in which only the details that are useful for a given task are kept. When different parts of a system depend on different technical domains (e.g. signal processing, automatic control, power management, etc.), the best suitable modeling paradigm may differ for each part. A global model of such a system is a heterogeneous model. Heterogeneous modeling, or multi-paradigm modeling, is the research domain which aims at handling heterogeneous models.

This paper focuses on *model composition*, one of the existing multi-paradigm techniques [HB09]. The main principle of model composition is the “gluing” of model parts which are described using different modeling languages. In model composition, the main difficulty is to define accurately the *semantic adaptation*, i.e. the mechanism to "glue" together model parts that may have very different semantics, in order to obtain a global heterogeneous model which is meaningful and can therefore be used for early verification and validation in the design process.

We have developed a framework called ModHel’X for heterogeneous model composition. ModHel’X is mainly aimed at model execution, i.e. techniques such as simulation or code generation. We compose models in a hierarchical way.In [BHJM11], we have presented in detail how hierarchical semantic adaptation between two models is handled in ModHel’X. One drawback of our current approach is the lack of conciseness and the rather low level of abstraction of the semantic adaptation descriptions. Inspired by the work by André et al. [MDAS10] on the description of the semantics of dataflow models using MARTE’s Clock Constraint Specification Language (CCSL), we propose an approach in which CCSL is used to model semantic adaptation.

The paper is organized as follows. After presenting ModHel’X in Section 2, we focus on semantic adaptation in Section 3. An example illustrates the underlying concepts and underlines relevant practical concerns. Then, Section 4 briefly introduces the basic concepts of CCSL needed

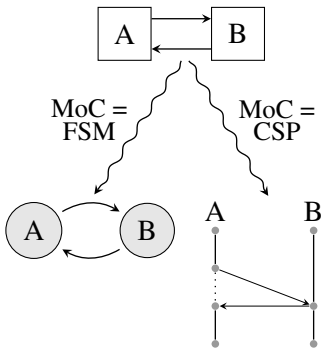

Figure 1: Models of computation.

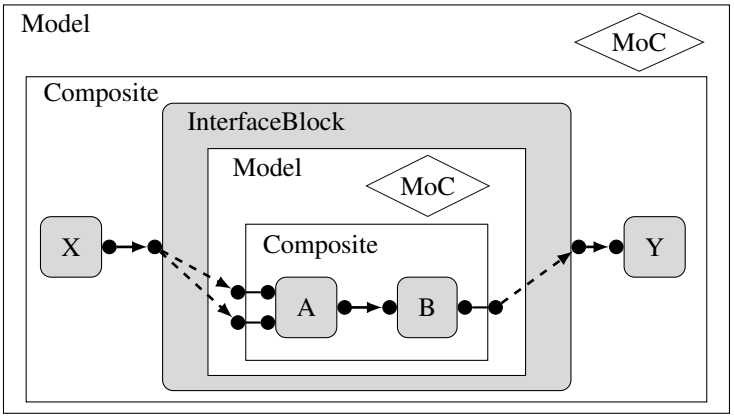

Figure 2: Model conforming to the meta-model of ModHel'X.

in Section 5 to describe the semantics of different models and the semantic adaptation between them. We discuss the results in Section 6 and, after a comparison of our approach with related work in Section 7, we conclude in Section 8.

2 ModHel'X, a framework for heterogeneous modeling

There are two main tasks to achieve in order to obtain a meaningful heterogeneous model using model composition: (1) the precise definition of the semantics of each modeling paradigm; (2) the precise definition of the semantic adaptation between parts of a model which use different modeling paradigms. One method for defining the semantics of different modeling paradigms is to use a common syntax or meta-model to describe the structure of models, and to attach semantics to this structure using so-called *models of computation (MoC)*. A model of computation is a set of rules which define the nature of the components of a model and how their behaviors are combined to produce the behavior of the model. For instance, Figure 1 shows that two models can share the same structure (two components linked by two arrows) with different semantics: a finite state machine or two communicating sequential processes, depending on the model of computation.

ModHel'X [BHJM11] is an experimental framework developed at Supélec in order to test new ideas about the executable semantics of heterogeneous models. ModHel'X allows one to describe the structure of heterogeneous models, to define models of computation for interpreting such structures, and to define the semantic adaptation between heterogeneous parts of a model. For this, ModHel'X relies on a meta-model which is the common syntax for all models, whatever their semantics. Figure 2 presents an example model conforming to that meta-model. ModHel'X uses *Blocks* as the basic unit of behavior. On the figure, X, Y, A and B are blocks. Blocks are considered as black boxes, meaning that their behavior can only be observed at their interface which is composed of *Pins* (black circles on the figure). The structure of a model is defined by setting relations between pins, shown as solid arrows on the example.

Such a structure has a meaning only when it is associated to a model of computation. Therefore, a ModHel'X model is composed not only of a structure (the *Composite*), but also of a *Model of*

Computation (MoC) that allows the interpretation of the structure. In ModHel'X, interpreting a model means executing the behavior described by that model according to the semantics of the MoC. An execution is a series of observations of the model, each observation being computed through the sequential observation of the blocks of the model using a fixed-point algorithm. The observation of one block is called an *update*. Each MoC dictates the rules for scheduling the update of the blocks of a model, for propagating values between blocks, and for determining when the computation of the observation of the model is complete.

In ModHel'X, heterogeneity is handled through hierarchy: the behavior of some blocks can be defined by another ModHel'X model. Such blocks are *InterfaceBlocks*. The model of computation used in the model of the block (the inner MoC) can differ from the model of computation of the model in which the interface block is used (the outer MoC). The dashed arrows between the pins of the interface block and the pins of the internal model represent the *semantic adaptation* between the two MoCs, which is realized by the interface block. As we have shown in [BHJM11], three aspect must be considered in the semantic adaptation: the adaptation of data (data may not have the same form in the inner and outer models), the adaptation of time (the notions of time and the time scales may differ in the inner and outer models) and the adaptation of control (control meaning the instants at which it is possible or necessary to communicate with a block through its interface). In the next section, we illustrate these three aspects on an example.

3 Semantic Adaptation

In order to illustrate what semantic adaptation is, we present the example of a power window system. The system, shown on Figure 3, is composed of a control switch, a controller board and an electro-mechanical subsystem. These components communicate through a bus.

Since the communications on the bus can be modeled by events which carry some data and occur at a given time, a "Discrete Events" (DE) [BLL+08] model of computation is suitable for the top level of the hierarchical model of this system. The control switch is considered as an atomic component which produces an event each time its position (neutral, up or down) changes.

The controller board is in charge of interpreting both the actions of the user on the switch and the information from the electro-mechanical subsystem in order to drive the motor which makes the window move. Advanced features of the window such as the *"one touch"* mode (i.e. the automatic raising or lowering of the window after a *brief* pull or push of the control switch), are realized by the controller. The behavior of this controller can naturally be described using a finite state machine. However, the one touch mode feature implies *timed behavior*: it is activated only when the control switch has been pulled or pushed during less than a given delay, 50ms for instance. Therefore the state machine describing the behavior of the controller board includes *timed transitions*, so the "Timed Finite State Machine" (TFSM) model of computation is used.

Finally, the electro-mechanical part is described as a periodically sampled system, and we use a "Synchronous Data Flow" (SDF) [BLL+08] model of computation for it. In this model of computation, blocks are data flow operators which consume and produce a fixed number of data samples on their pins each time they are activated.

Once these choices are made, it is necessary to define how these three models, involving three different models of computation, interact. In the following, we present each of the three aspects

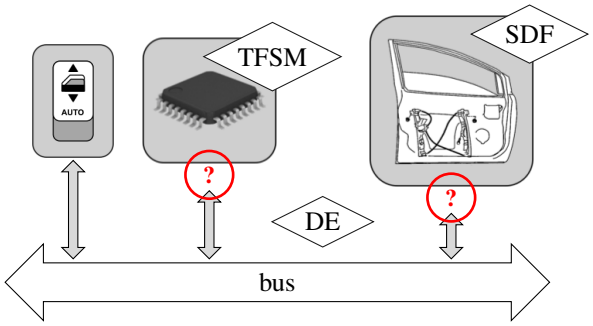

Figure 3: Structure of the Power Window Model.

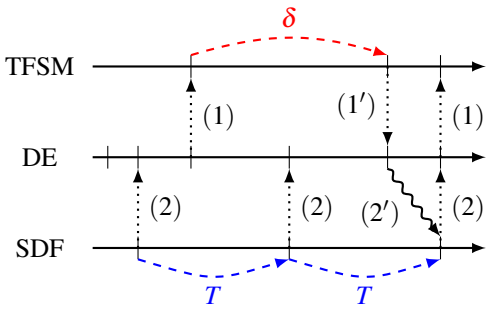

Figure 4: Adaptation of control.

to consider in semantic adaptation: the adaptation of data, of time and of control.

Semantic adaptation of data: The most obvious form of adaptation between models of computation is adaptation of data. For instance, in the DE model of computation, blocks communicate by posting events which are composed of a value and a timestamp. In the finite state machine, data appears as symbols which can trigger transitions. In the data flow model of the electro-mechanical part, data appears as periodic samples. The adaptation of data between DE and TFSM can be performed by mapping symbols (processed by the TFSM) to event values. The adaptation of data between DE and SDF is more complicated because SDF expects periodic samples while DE has only sporadic events. A usual way to handle this it to interpret a DE event as a new value for the next samples of a SDF signal, until a further event is received. Similarly, a *change* in a sequence of SDF samples is converted into a DE event. The value carried by this event is easy to determine: it is the new value of the SDF signal. However, it is also necessary to choose a timestamp for this event because there is no explicit notion of time in SDF: *time* needs to be adapted too.

Semantic adaptation of time: When we generate a DE event to reflect a change of a SDF signal, a possible timestamp for this event is the value of the current time in the DE model when the SDF signal changes (the DE MoC maintains a *current time* at each instant of the model execution [BLL+08]). Adaptation of time is more complex between the DE model and the timed finite state machine. We can assume that the state machine reacts instantaneously to input symbols, and also uses the current time in DE as timestamp for the events it produces. However, time can also trigger transitions in the TFSM model. We recall here that the duration of a timed transition is expressed on the time that is local to the TFSM model. Assuming that such a transition may produce an event which will have to be adapted for the DE model, this duration must have a correspondence in the DE time. If, for the sake of simplicity, we assume that time runs at the same rate in the DE model and in the timed finite state machine, the adaptation of time between DE and TFSM consists in reseting a timer each time a new state is entered, and therefore measuring the time elapsed in DE since entering the state in the automaton.

Semantic adaptation of control: The semantic adaptation of control is the most complex type of adaptation. Control is the set of instants at which a block should be able to take inputs into

account and to produce outputs. The adaptation of control is related to the adaptation of data and of time. Figure 4 illustrates the adaptation of control between DE, TFSM and SDF. On this figure, the "ticks" on each timeline represent the instants at which each model is given control. The arrows represent the adaptation of control performed between the models by the interface blocks.

For instance, when the DE model produces an input for the state machine, control should be given to the TFSM model so that it can process the symbol and take a transition. This is illustrated by arrows labelled (1) on the figure. Conversely, control is created in DE when the state machine produces an output (arrow $(1')$). If the state machine enters a state with an outgoing timed transition, the state machine should receive control when the delay δ expires so that the transition fires (red arrow labelled δ on the figure).

Regarding the DE/SDF adaptation, the sampled nature of SDF signals induces periodic control for the model of the electro-mechanical part of the system (arrows labelled T at the bottom of the figure). Since this model is embedded in the DE model, control in DE has a periodic part induced by SDF. This is shown by arrows labelled (2) on Figure 4. But when some data is made available by DE to the SDF model, this data must not create control directly in SDF but must be processed at the next periodic control point, as shown by the wavy arrow labelled $(2')$ on Figure 4.

Adaption of control therefore depends on data and time, but it must also obey rules that depend on the models of computation. It is therefore cumbersome to define this adaptation in an operational way as we did until now in ModHel'X. In this paper, we present an approach in which we *declare* all the constraints that apply on the control points of the different parts of a model. Solving these constraints allows us to find the control points of the blocks in the model that represent an execution according to the semantics of the MoC. This work has been inspired by the work by André et al. [MDAS10] in which the Clock Constraints Specification Language (CCSL) is used to define the SDF model of computation. Here, we use CCSL not only to define models of computation, but also to model the semantic adaptation between them. In the following section, we briefly introduce the basic elements of the CCSL language that are necessary to understand how we model MoCs and semantic adaptation between them (see section 5).

4 The Clock Constraint Specification Language (CCSL)

CCSL (Clock Constraint Specification Language) is a declarative language annexed to the specification of the MARTE UML Profile (Modeling and Analysis of Real Time and Embedded systems). CCSL is based on the notion of *clock* which represents a set of discrete event occurrences, called *instants*. A clock can be either *chronometric* or *logical*. Chronometric clocks are a mean to model "physical time" and to measure durations between two instants. Logical clocks represent discrete time in which instants are occurrences of any kind of logical event. Each occurrence of the event is represented by a *tick* on the corresponding clock. The distance between two ticks is measured in terms of ticks and has no meaning by itself (no link with "physical durations").

CCSL offers means to specify constraints between clocks. Clock constraints can be classified into four categories: synchronous, asynchronous, mixed and non-functional contraints. **Synchronous** clock constraints are based on the concept of coincidence. Examples are *subclocking* and *discretization*. A subclocking constraint such as `A isSubClockOf B;` is an order-preserving mapping of each instant of the subclock A with an instant from the superclock B. The `discretizedBy`

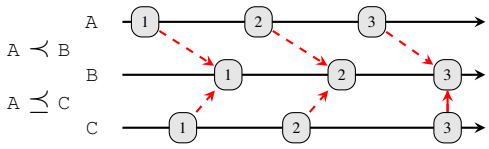

Figure 5: Example of strict and non-strict precedence.

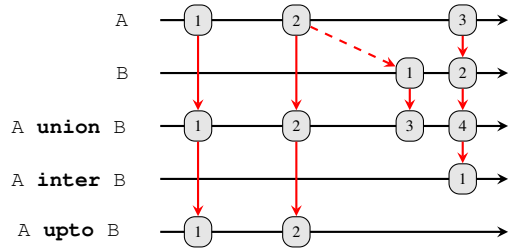

Figure 6: Example of union, intersection and upto clock expressions.

constraint uses *IdealClk* to define a chronometric clock. *IdealClk* is a dense clock defined by MARTE. For instance, `Clock C = IdealClk discretizedBy 0.001;` specifies a discrete chronometric clock with a period of 0.001 second = 1 ms. **Asynchronous** clock constraints specify precedence relationships ($\prec$ or $\preceq$) between all the respective instants of two clocks (see Figure 5). In a **mixed** clock constraint, coincidence and precedence are combined. For example the constraint `Clock C = A delayedFor n on B;` specifies a clock C that has all its instants coincident with the n^{th} instant of B that follows an instant of A.

There is also a series of operators to define new clocks. The boolean operators `union` and `inter` act onto pairs of respective samples (see Figure 6). `A upto B` defines a clock that ticks each time A ticks up to the first tick of B; from this tick on, it does not tick anymore.

The TimeSquare environment, an Eclipse plug-in, may be used to calculate solutions to a set of CCSL constraints. A graphical interface displays waveforms for the solution clocks, as well as the constraints between the instants of different clocks.

5 Semantic Adaptation using CCSL

This section describes our general methodology for describing MoCs and semantic adaptation between them using CCSL. The methodology is illustrated on the power window example introduced in Section 3. Figure 7 shows how this system is modeled using ModHel'X. For the sake of simplicity, only part of the behavior of the window controller is taken into account, the half that controls upward movements, including the one touch mode. The other half is symmetric. For the same reasons, the SDF model of the electromechanical part is considered as a black box.

The control switch is an elementary DE block that models the user's actions. The outputs *outUUp* and *outUNeutral* correspond respectively to the user pulling and releasing the button.

The controller is described using a timed finite state machine, with initial state *Stop*. When the automaton receives the *evtUUp* event indicating that the user wants to raise the window, it produces the *evtWUp* event to start the window motor and goes to the *Up* state. If the automaton receives the *evtUNeutral* event before 50 units of time, thereby indicating that the user has released the button to activate the one touch mode, it goes to the *UpAuto* state. Else, and after 50 units of time, it goes to the *UpManu* state. The controller produces the *evtWStop* when the user releases the button in manual mode, when the user pulls the button in one touch mode, or when the window is fully closed (*evtWEnd* events produced by the electro-mechanical subsystem represent end-stops).

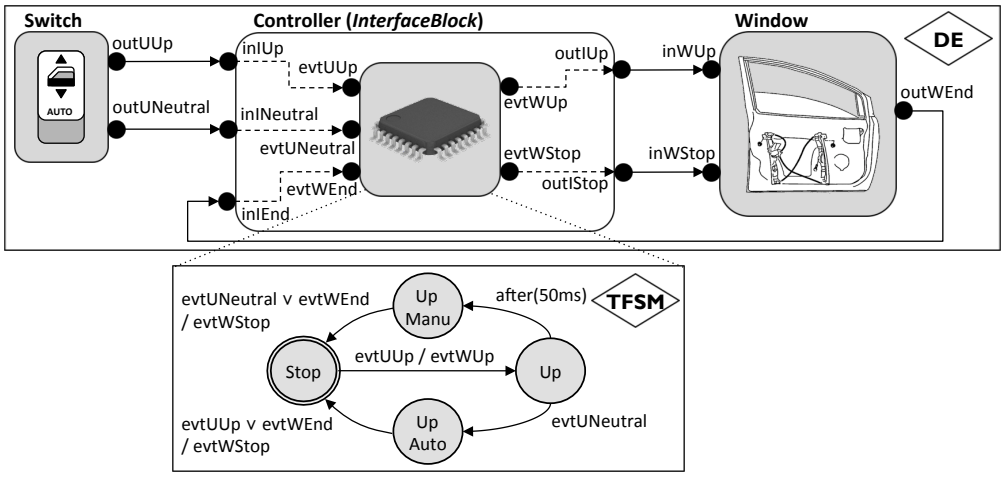

Figure 7: Simplified ModHel'X model of the power window system.

In the following sections, we describe how models governed by the TFSM and DE MoCs may be described using CCSL. Then we present the semantic adaptation between them.

5.1 Describing TFSM using CCSL

This section describes our methodology for translating a TFSM model into a CCSL specification.

In CCSL, all clocks must be subclocks of a root clock. We choose to explicitly define a chronometric clock called *chronoTFSM*. This clock serves several purposes: it measures the durations of the timed transitions, the input events occur at instants of this clock (they are subclocks of it), and therefore the state machine reacts at instants of this clock.

Simulating the behavior of a state machine implies the memorization of its current state. For this purpose we associate two clocks to each state S: *enterS* which ticks when a transition leading to S fires, and *inS* which ticks at each instant when the state machine is in state S. *enterS* is the *condition* for entering state S; *inS* is a *memory* of the current state.Therefore *inS* must tick at each instant after *enterS*, until a transition leading to another state fires, i.e. when one of the $enterS_i$' ticks, where $\{S_i\text{'}\}$ is the set of states accessible from S by one outgoing transition. This gives:

```
inS = sustain enterS upto ((enterS1' union enterS2') ... union enterSn')
```

To define the *enterS* family of clocks, let us describe first *when* transitions are followed. A *non-timed* transition T that goes from S upon receipt of E is fired when the current state was S at the last tick, and E occurs. This provides a clock associated with T:

```
T = E inter (inS delayedFor 1 on chronoTFSM);
```

For a *timed* transition T that fires from state S after d units of time, we use the same method, except that the firing event is derived from *enterS*: `T = (enterS delayedFor d on chronoTFSM) inter (inS delayedFor 1 on chronoTFSM);`

We are now able to define the *enterS* clock of a state S, with incoming transitions $T_1, \ldots, T_n$. If S is not the initial state, S is entered when one of the transition fires, thus:

```
enterS = ((T1 union T2) union ...) union Tn;
```

If S is the initial state, then there is a slight difference: S is also entered at the start. For this, we define a clock that ticks only once, on the first tick of *chronoTFSM*:

```
Clock initial is chronoTFSM filteredBy 0B1(0); //"filteredBy" applies a binary pattern)
```

Then all we have to do is add *initial* to the conditions for entering S:

```
enterS = (((T1 union T2) union ...) union Tn) union initial;
```

The events produced as outputs by the state machine are modeled by clocks too. A given output event E is emitted when one of the transitions that may produce it is fired. Let us call $T_1, \ldots, T_n$ the family of such transitions. Then we can define a clock E as:

```
E = ((T1 union T2) union ...) union Tn;
```

Based on those generic constraints, we have created a script that generates automatically the constraints needed for any instance of a TFSM model. For the TFSM model of the power window system example described previously, we obtain the following constraints:

```
Clock chronoTFSM is IdealClk discretizedBy 0.001;
Clock initial is chronoTFSM filteredBy 0B1(0);
// state S [Stop] and incoming transitions
transition1 = (evtUNeutral union evtWEnd) inter (inM delayedFor 1 on chronoTFSM);
transition2 = (evtUUp union evtWEnd) inter (inA delayedFor 1 on chronoTFSM);
enterS = ((transition1 union transition2) union initial);
enterS isSubClockOf chronoTFSM;
inS = sustain enterS upto enterU;
inS isSubClockOf chronoTFSM;
// state M [Up Manu] and incoming transitions
transition4 = (enterU delayedFor 50 on chronoTFSM) inter
                                        (inU delayedFor 1 on chronoTFSM);
enterM = transition4;
enterM isSubClockOf chronoTFSM;
inM = sustain enterM upto enterS;
inM isSubClockOf chronoTFSM;
... // same thing for states U [Up] and A [Up Auto]
// output events
evtWStop = (transition1 union transition2);  // ...
```

Figure 8 shows the simulation of these constraints in TimeSquare with a simulation scenario specified using the *evtUUp*, *evtUNeutral* and *evtWEnd* clocks. The arrows show the causal relationships between the ticks leading to state changes for the automaton.

5.2 Describing DE using CCSL

We now describe our methodology for translating a DE model into a set of CCSL constraints.

As for TFSM, we define a chronometric clock *chronoDE* to measure time. All the other clocks are subclocks of *chronoDE*. The main challenge is to schedule the updates of the blocks globally. Therefore each block B is associated with a clock *updateB* that ticks at each update of B. As the updates themselves depend on events sent and received by the blocks, we need to associate a clock to each pin that ticks each time an event is sent or received. For a pin X, this clock is called *inputX* or *outputX*, depending on the nature of the pin.

DE semantics imply the following constraints on these clocks: (a) the clock of an output pin must coincide with all the clocks of the connected input pins and (b) the *update* clock of a block is the union of all the clocks of its input and output pins.

Again, based on these rules, a script can generate automatically the constraints needed for any specific DE model. On the power window system example, we obtain the following constraints :

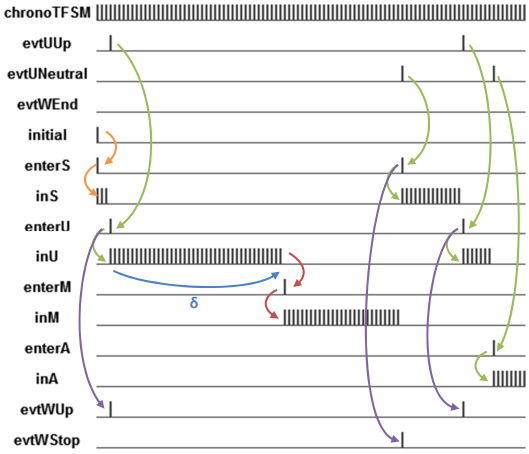

Figure 8: Simulation of the TFSM model.

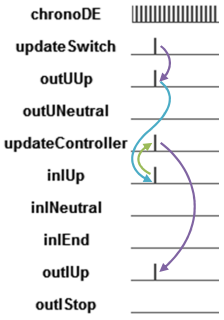

Figure 9: Simulation of the DE model.

```
Clock chronoDE is IdealClk discretizedBy 0.001;
// "Switch" block
updateSwitch = outUUp union outUNeutral;
updateSwitch isSubClockOf chronoDE;
// "Controller" block
updateController = inIUp union (inINeutral union ...;
updateController isSubClockOf chronoDE;
// Relations between blocks
outUUp = inIUp;
outUNeutral = inINeutral;
...
```

Figure 9 shows traces obtained when an *outUUp* event is sent by the switch (when it is updated). It is received by the controller (as an *inIUp* event), which sends out an *outIUp* event to the window.

5.3 Semantic Adaptation between DE and TFSM

Once the TFSM and DE models have been generated using the scripts implementing the methodology, semantic adaptation is very simple to model. The main principle is to focus on the interface block and to write relations between clocks of the outer and inner models. More specifically: (a) an equality is written for each pair of related input and output pins of the inner/outer models, and (b) there must be a relation between the two chronometric clocks *chronoTFSM* and *chronoDE*.

```
// Adaptation of inputs
inIUp = evtUUp;
inINeutral = evtUNeutral;
inIEnd = evtWEnd;
// Adaptation of outputs
evtWUp = outIUp;
evtWStop = outIStop;
// Relation between clocks
chronoDE isPeriodicOn chronoTFSM period 3;
```

Figure 10 shows the result obtained in TimeSquare for the global heterogeneous model made from the DE top level model of the system and the TFSM model of the window controller.

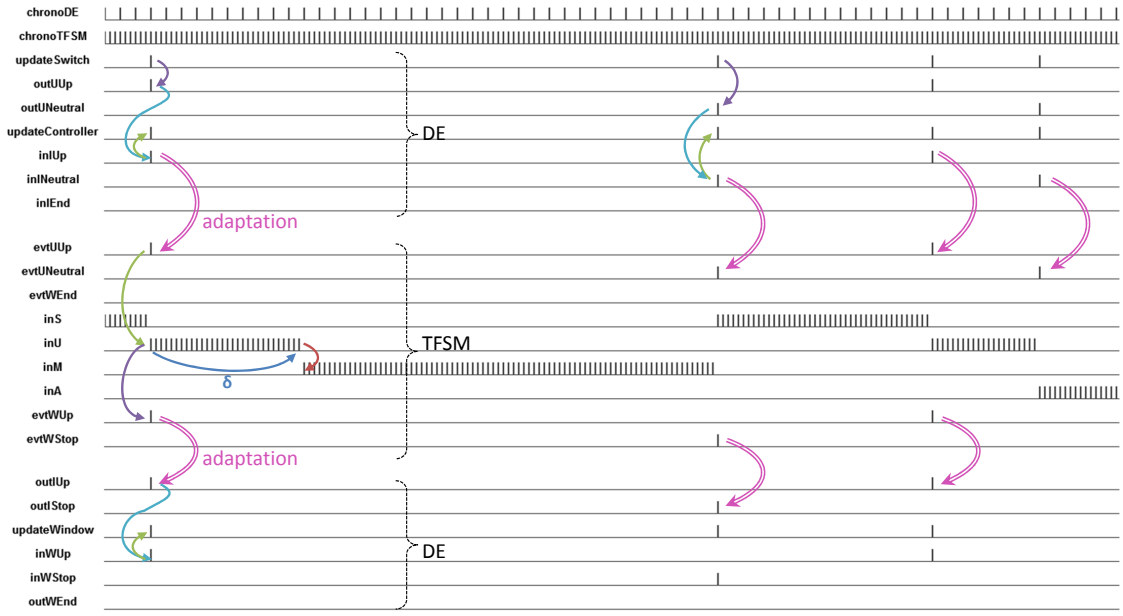

Figure 10: Waveforms for the complete power window system example

6 Discussion

The above results show some benefits and drawbacks of our approach. We managed to obtain very concise CCSL specifications for MoCs, what is very positive compared to the lengthy descriptions in ModHel'X. However, we consider as a drawback the fact that the CCSL specifications are model instances instead of independent descriptions of MoCs. To enforce genericity, we had to write scripts that generate model instances according to rules defining the semantics of the MoC.

Another positive point is that semantic adaptation of control and time is quite easy to define using CCSL. In addition, we were able to check the consistency of the CCSL specifications of the whole heterogeneous model of the power window (although interpreting the errors reported by TimeSquare was difficult). For instance, if the adaptation constraints specify the DE clock to be of higher frequency than the TFSM clock, the global specification is inconsistent: the delay of timed transitions in TFSM cannot be mapped on DE time. The solver actually detects a deadlock. Analysis features are of utmost interest for an approach dedicated to the specification of MoCs (and of semantic adaptation), which by nature are very difficult to verify and validate.

Regarding the power window case study, we discussed a simplified version of the system to demonstrate the feasibility of the approach. Therefore, we did not address the DE/SDF semantic adaptation, even if the SDF MoC could easily be adapted from [MDAS10]. We are currently working on a complete version of the example including this interesting adaptation.

One limitation of this clock-based approach is that semantic adaptation of data cannot be addressed in the same way since primitives for manipulating data structure and values do not exist in CCSL. Therefore, another methodology has to be defined for it. Another issue is the integration of this approach in ModHel'X. TimeSquare's solver is a static solver, i.e. the computed solution

is a set of clocks with all the ticks for the whole timespan. It is not possible to compute the ticks *at runtime*. In consequence, the use of TimeSquare in the execution algorithm of ModHel'X is far from being straightforward. For the time being we cannot use the existing mechanisms to handle the adaptation of data together with CCSL specifications for the adaptation of control and time.

In the following section, we compare this paper's contributions with existing approaches.

7 Related Work

As stated earlier, this paper is inspired by the work by André et al. [MDAS10]. First we have adapted their approach to the ModHel'X framework. We use CCSL clocks to model the control points of the execution algorithm of ModHel'X on the different elements of a model (conforming to the meta-model of ModHel'X). Then we have applied this approach to two MoCs. But our main contribution is the use of CCSL specifications not only to model MoCs but also to model the semantic adaptation between two models involving different MoCs. We have shown on an example that this approach is particularly suitable for describing the semantic adaptation of control and of time, and that using CCSL specifications is significantly simpler than using an imperative method. Although not well integrated in ModHel'X yet (as exposed in section 6), this preliminary work seems promising since it allowed us to detect inconsistencies in the specifications.

To our knowledge, no other approach uses clocks and clock constraints to model semantic adaptation in the context of model composition. However, the issue of handling different notions of time and multiple control clocks has been extensively studied, in particular in the domain of hardware synthesis. Synchronous languages (see [BG92, BCE+03]) like Lustre, Esterel and Signal use abstract logical time and introduce the notion of multiform time. Other approaches, like Lucid Synchrone [BCHP08], have explicit support for specifying multi-clock systems.

Regarding model composition itself, ModHel'X can be compared to other approaches such as Ptolemy II or the MATLAB/Simulink toolchain. Ptolemy II [EJL+03] is one of the first approaches to model composition. It supports a wide range of MoCs that may be combined with each other to form heterogeneous models. In ModHel'X, we propose an extension and a generalization of the solutions introduced by Ptolemy. Adaptation rules at the boundary between two heterogeneous models is one of our main contributions. In Ptolemy, these rules are hardcoded in the kernel. The modeler has either to rely on default adaption and design its system accordingly, or to explicitly add adaptation blocks into the models themselves, what makes models less reusable and more difficult to understand. In ModHel'X, adaptation is explicit, insulated from the models and encapsulated into interface blocks. This work on the modeling of semantic adaptation using CCSL is another step towards an easier way to "glue" together heterogeneous parts of a model.

A power window case study, available on The MathWorks' website[1], illustrates heterogeneous model composition for Simulink (SDF-like) and Stateflow (TFSM-like). Semantic adaptation between Simulink and Stateflow is specified explicitly using functions and truth tables. However, all MoCs cannot be composed like this. For instance, using a Simulink (SDF-like) model into a SimEvents (DE-like) model requires different adaptation artifacts such as event translation blocks [CCM06]. Not only are the interactions of SimEvents with Simulink hardcoded: SimEvents is actually executed *on top of* Simulink, thus constraining their interactions. The abstract syntax

[1] See http://www.mathworks.com/products/demos/simulink/PowerWindow/html/PowerWindow1.html.

and semantics at the core of ModHel'X allow MoCs to be described independently from each other, and interface blocks allow the description of adaptation patterns for any pair of MoCs.

8 Conclusion

In this paper, we propose to use CCSL, a language for defining clocks and clock constraints, to specify the semantic adaptation at the border between two heterogeneous models composed in a hierarchical way. We have adapted an approach proposed in [MDAS10] to our framework for model composition called ModHel'X. This paper contains two examples of models of computation described using CCSL and we show how semantic adaptation of control and of time can be specified between two models using these MoCs. Although preliminary, this work shows interesting results regarding the conciseness and the readability of the descriptions of both MoCs and semantic adaptation. Moreover, the TimeSquare solver allowed us to check the consistency of the semantic adaptation between the two models. This work will be integrated into ModHel'X so that CCSL-like specifications for the semantic adaptation of control and of time can be used. In parallel, we are working on a methodology for modeling the semantic adaptation of data.

References

[BCE+03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, R. de Simone. The Synchronous Languages 12 Years Later. *Proc. of the IEEE* 91(1):64–83, 2003.

[BCHP08] D. Biernacki, J.-L. Colaco, G. Hamon, M. Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. 2008.

[BG92] G. Berry, G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming* 19(2):87–152, 1992.

[BHJM11] F. Boulanger, C. Hardebolle, C. Jacquet, D. Marcadet. Semantic Adaptation for Models of Computation. In *Proceedings of ACSD 2011*. Pp. 153–162. 2011.

[BLL+08] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng. Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains). Technical report UCB/EECS-2008-30, University of California, Berkeley, 2008.

[CCM06] C. G. Cassandras, M. I. Clune, P. J. Mosterman. Hybrid System Simulation with SimEvents. In *Proceedings of ADHS*. Pp. 267–269. 2006.

[EJL+03] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong. Taming heterogeneity – The Ptolemy approach. *Proc. of the IEEE* 91(1):127–144, 2003.

[HB09] C. Hardebolle, F. Boulanger. Exploring Multi-Paradigm Modeling Techniques. *SIMULATION* 85:688–708, 2009.

[MDAS10] F. Mallet, J. DeAntoni, C. André, R. de Simone. The clock constraint specification language for building timed causality models. *Innovations in Systems and Software Engineering* 6:99–106, 2010.