Semantic Adaptation using CCSL Clock Constraints

Frédéric Boulanger, Ayman Dogui, Cécile Hardebolle,
Christophe Jacquet, Dominique Marcadet, and 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: Multi-Paradigm Modeling, 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 [8]. 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 [4], 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. [9] 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. We first present the concept of model of computation in Sect. 2, together with an example that we use throughout the paper to illustrate the underlying concepts of our approach. After presenting ModHel'X in Sect. 3, we focus on semantic adaptation in Sect. 4. Then, Section 5 briefly introduces the basic concepts of CCSL needed in Sect. 6 to describe the semantic adaptation between heterogeneous models. We discuss the results in Sect. 7 and, after a comparison of our approach with related work in Sect. 8, we conclude in Sect. 9.

2 Models of Computation

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. It can be seen as a way to interpret the structure of a model. For instance, Figure 1 shows that two models can share the same structure (two components linked by two arrows) with different semantics, depending on the model of computation: here a finite state machine or two communicating sequential processes.

We use this concept of model of computation to achieve hierarchical model composition. To illustrate our approach, let us introduce the example of a power window system that we will use throughout the paper. The system, shown on Fig. 2, 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) [5] 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. It also implements advanced features 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. The behavior of this controller can be described naturally using a finite state machine. However, the one touch mode feature implies *timed behavior*: it is activated only when the

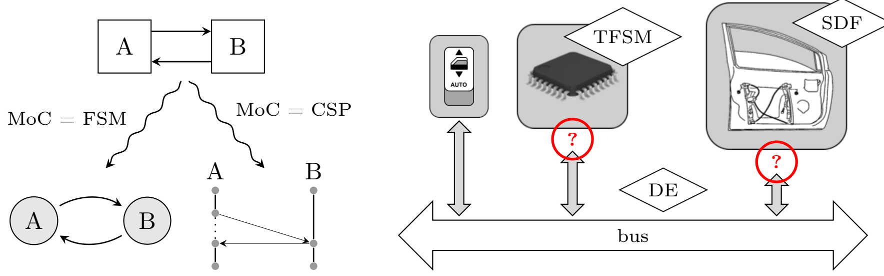

Fig. 1. Models of computation.

Fig. 2. Structure of the power window model.

control switch has been pulled or pushed during less than a given delay, 10 ms 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, represented by a “Synchronous Data Flow” (SDF) [5] MoC. 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, can be composed. In the following, we present how the power window system is modeled in ModHel’X.

3 ModHel’X, a Framework for Heterogeneous Modeling

ModHel’X [4] is an experimental framework developed at Supélec. It 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 3 shows how the power window system is modeled using ModHel’X. ModHel’X uses *Blocks* as the basic unit of behavior. For instance, the Switch, Position and EndStop elements on the figure 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.

A structure (set of blocks and relations) has a meaning only when it is associated with a model of computation that allows its *interpretation*. Therefore, a ModHel’X model is a ⟨structure, MoC⟩ *pair*. MoCs are depicted by diamonds on Fig. 3. 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

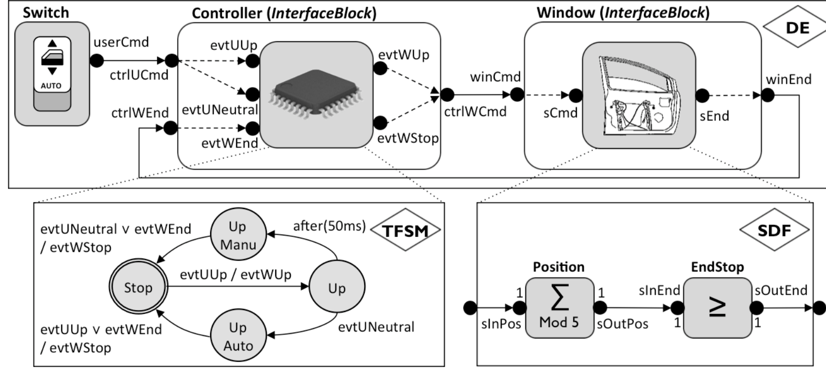

Fig. 3. Simplified ModHel'X model of the power window system.

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.

Note that like all other models, the TFSM model is a set of interconnected blocks. However, a more traditional depiction is used on Fig. 3. Also for simplicity's sake, only the upward movement of the window is taken into account, including the one touch mode. The other part of its behavior is symmetric.

In ModHel'X, heterogeneity is handled through hierarchy: the behavior of some blocks can be defined by another model. Such blocks are called *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 Controller and Window elements on Fig. 3 are examples of InterfaceBlocks. The dashed arrows between the pins of an interface block and the pins of its internal model represent the *semantic adaptation* between the two MoCs, which is performed by the interface block. As we have shown in [4], semantic adaptation must consider three aspects: 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 the power window example.

4 Semantic Adaptation

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

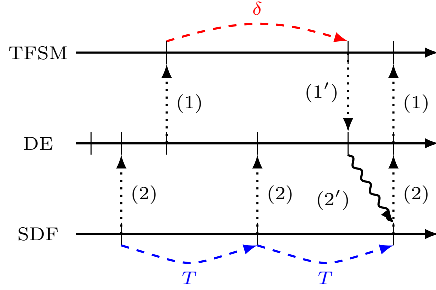

Fig. 4. Adaptation of control.

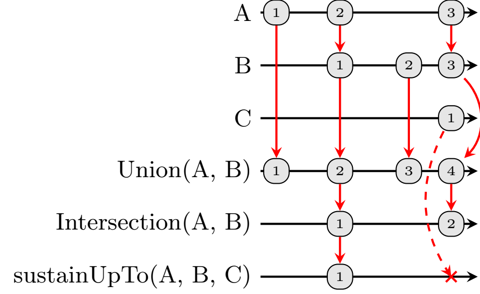

Fig. 5. Example of clock expressions.

(processed by the TFSM) to event values. The adaptation of data between DE and SDF is more complex 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.

When we generate a DE event to reflect a change of an 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 [5]). **Adaptation of time** must also be performed 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 a timestamp for the events it produces. However, time can also trigger transitions in the TFSM model. Such a transition is based on a duration expressed on a time scale that is local to the TFSM model. The transition may produce an event that will have to be adapted to the DE model, so the duration must have a correspondence in DE time. 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.

Control is the set of instants at which a block should be able to take inputs into account and to produce outputs. 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. Let us have a look at a few examples.

Between DE and TFSM. 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 (arrow labelled δ on the figure).

Between DE and SDF. 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 Fig. 4. When 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 Fig. 4.

As we see in this example, adaption of control not only 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. This work has been inspired by the work by André et al. [9] in which the Clock Constraints Specification Language (CCSL) is used to define the SDF model of computation. Our goal is to use CCSL to model the semantic adaptation between models involving different models of computation. Section 5 introduces the basics of the CCSL language, as a prerequisite to Sect. 6 that describes our methodology for semantic adaptation using CCSL.

5 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 means to model "physical time" and to measure durations between two instants. Logical clocks represent discrete time composed of abstract instants called *ticks*. The number of ticks between two instants may have no relation to any "physical duration".

The concrete syntax of CCSL is quite verbose and requires to prefix actual parameters with the name of the formal parameter in operator calls. For the sake of simplicity and conciseness, we will omit such prefixes and use generalized n-ary versions of the binary operators. For instance, we will write `Expression E = Union(C1, C2, ..., Cn)` instead of:

```
Expression U1 = Union(Clock1->C1, Clock2->C2)
Expression U2 = Union(Clock1->U1, Clock2->C3)
   ...
Expression E = Union(Clock1->Un-1, Clock2->Cn)
```

CCSL has a series of operators to define new clocks. The operators `Union` and `Intersection` build clocks which consist of respectively the union and the intersection of the ticks of two clocks (see Fig. 5). We also use `sustainUpTo(A, B, C)`, which defines a clock that starts ticking each time A ticks at the first tick of

B, and stops ticking at the first tick of C. **`Discretize`** defines a chronometric clock from *physicalTime*, a dense clock defined by MARTE. For instance, **`Discretize`**`(physicalTime, 0.001)` specifies a discrete chronometric clock with a period of 0.001 second = 1 ms. **`DelayFor`**`(A, B, n)` specifies a clock which ticks at the nth instant of B that follows an instant of A.

CCSL also offers means to specify constraints between clocks, namely subclocking and coincidence. **`Relation`**`[`**`SubClock`**`](A, B)` means that the set of ticks of A is a subset of the set of ticks of B. **`Relation`**`[`**`Coincides`**`](A, B)` means that A and B share the same set of ticks.

The TimeSquare environment, an Eclipse plug-in, may be used to solve a set of CCSL constraints. A graphical interface displays waveforms for the solution clocks and shows the constraints between their instants.

6 Semantic Adaptation using CCSL

This section presents our general methodology for describing semantic adaptation between models involving different MoCs using CCSL. The methodology is illustrated on the power window example introduced in Sect. 2.

The switch (see Fig. 3) is an elementary DE block that models the user's actions. The user pulling and releasing the button is represented by DE events on the switch's *userCmd* pin with values "1" and "0" respectively. These events are provided to the controller, which, in turn, sends *ctrlWCmd* DE events to the window to turn the motor on (value "1") or off (value "0"). The controller knows when the window is fully closed when it receives an event with value "1" from the window on its *ctrlWEnd* pin.

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 10 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 10 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 represent end-stops). As introduced in Sect. 4, the notions of data, control and time have to be adapted between the DE top level model and the internal TFSM model of the controller. In particular, a correspondence has to be defined between DE events carrying given values and TFSM events.

The window is described using a synchronous data-flow model. In this simplified example, we suppose that the window has 5 vertical positions, where 0 is the lowest and 4 is the highest. The *Position* block is a modulo-5 accumulator which computes the position of the window from the input command signal. The role of the *EndStop* block is to detect when the window reaches its highest possible position. It produces a signal with value "1" when the window is at its highest position and "0" otherwise. Again, the notions of data, control and time have to

be adapted between the DE top-level model and the internal SDF model of the window. In particular, a correspondence has to be defined between DE events (carrying values) and SDF samples (carrying values).

In order to be able to describe the semantic adaptation between models using CCSL and to simulate the global heterogeneous model using ModHel'X, we have to integrate TimeSquare, the CCSL solver, with ModHel'X. However, given the fundamental differences between the tools, we have preferred to first experiment our approach using CCSL only, as a proof of concept. For this reason, the following sections explain how models governed by the TFSM, SDF and DE MoCs may be described using CCSL, before focusing on semantic adaptation between them. Then we present the simulations obtained with TimeSquare.

6.1 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.

For simulating the behavior of a state machine, we need to memorize its current state. For each state S, we use an *enterS* clock which ticks when a transition leading to S fires, and an *inS* clock which ticks at each instant when S is the current state. *enterS* is the *condition* for entering S, and *inS* is a *memory* of the current state.

To define the *enterS* family of clocks, let us describe first *when* transitions are followed. A *non-timed* transition T that leaves S upon receipt of E is fired when the current state is S, and E occurs. Therefore, the clock which ticks each time T fires can be defined as:

```
Expression T = Intersection(E, inS)
```

For a *timed* transition T that leaves state S after d units of time, the firing event is derived from *enterS*:

```
Expression T = Intersection(DelayFor(enterS,chronoTFSM,d), inS)
```

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:

```
Expression enterS = Union(T1, T2, ... , Tn)
```

For any state S, the state machine is in state S the instant just after the firing of a transition leading to S, so:

```
Expression enteredS = DelayFor(enterS, chronoTFSM, one)
Expression inS = sustainUpTo(chronoTFSM, enteredS,
                             Union(inS', inS'', ...))
```

where *S', S", ...* is the list of successor states of S.
If S is the initial state, then the state machine S is also in S at the first instant. For this, we define a clock that ticks only once, on the first tick of *chronoTFSM*:

```
Expression initial = FilterBy(chronoTFSM, 1(0))
```

and we add *initial* to the conditions for being in S:

```
Expression enteredS = Union(initial, DelayFor(enterS, ...))
```

The events produced 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 which ticks each time E is present as:

```
Expression E = Union(T1, T2, ..., Tn)
```

Based on those generic patterns, 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 (using the simplified CCSL syntax):

```
Clock physicalTime:Dense
Expression chronoTFSM = Discretize(physicalTime, 0.001)
Expression initial = FilterBy(chronoTFSM, 1(0))
// state S [Stop] and incoming transitions
Expression transition1 = Intersection(Union(evtUNeutral, evtWEnd), inM)
Expression transition2 = Intersection(Union(evtUUp, evtWEnd), inA)
Expression enterS = Union(transition1, transition2)
Expression enteredS = Union(initial, DelayFor(enterS, chronoTFSM, one))
Relation[SubClock](enterS, chronoTFSM)
Expression inS = sustainUpTo(chronoTFSM, enterS, inU)
Relation[SubClock](inS , chronoTFSM)
// state M [Up Manu] and incoming transitions
Expression transition3 = Intersection(DelayFor(enterU, chronoTFSM, 10), inU)
Expression enterM = transition3
Expression enteredM = DelayFor(enterM, chronoTFSM, one)
Expression inM = sustainUpTo(chronoTFSM, enteredM, inS)
Relation[SubClock](inM, chronoTFSM)
[...] // same thing for states U [Up] and A [Up Auto]
// output events
Expression evtWStop=Union(transition1, transition2)
[...]
```

Figure 6 shows the causal relationships between the ticks upon which the automaton changes its state.

6.2 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*. 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 (with the same name), which ticks each time an event is sent or received. 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:

```
Clock physicalTime:Dense
Expression chronoDE=Discretize(physicalTime, 0.001)
// "Switch" block
Clock updateSwitch
Clock userCmd
Relation[Coincides](updateSwitch, userCmd)
Relation[SubClock](updateSwitch, chronoDE)
// "Controller" block
Clock updateController
Clock ctrlUCmd
Clock ctrlWEnd
Clock ctrlWCmd
Relation[Coincides](updateController, Union(ctrlUCmd, ctrlWEnd, ctrlWCmd))
Relation[SubClock](updateController, chronoDE)
// Relations between blocks
Relation[Coincides](userCmd, ctrlUCmd)
[...]
```

To fully simulate our model in TimeSquare, we have to take into account the values of the DE events. However, CCSL has no mechanism for representing data. Therefore, we represent data values by clocks. For instance, since the *Switch* block produces DE events with values “0” or “1” on its *userCmd* pin, we use a *userCmd0* (resp. *userCmd1*) clock which ticks each time the value of the produced event is 0 (resp. 1). This mechanism is applied to all the clocks representing the emission or reception of events in the DE model. Figure 6 shows traces obtained when a *userCmd* event is sent by the switch (upon update) with value “1”. It is received by the controller as a *ctrlUCmd* event with value “1”, which sends out a *ctrlWCmd* event to the window with value “1” to start the window motor.

6.3 Describing SDF using CCSL

This section explains how an SDF model can be translated into a CCSL specification. First, we define a *superSDF* clock to represent the instants at which blocks are updated in the SDF model. For each block B we associate a clock *updateB* that ticks at each update of B. The *updateB* clock is necessarily a subclock of *superSDF*. For each input/output pin of a block B, we define a clock *sInBi*/*sOutBj* that ticks each time it receives/produces a token. Block B is updated when each of its input pins has received at least one token. Therefore, the *updateB* clock must coincide with the slowest clock among all the *sInBi* clocks in order to tick upon the receipt of the last required token. This gives:

```
Relation[Coincides](updateB, Sup(sInB1, ..., sInBn))
```

When B is updated, each of its output pins produces a token. Therefore, the *updateB* clock coincides with all the *sOutBj* clocks. This gives, for each j:

```
Relation[Coincides](updateB, sOutBj)
```

The semantics of SDF implies that, for two blocks A and B connected through relations, each token produced by each output pin $sOutA_j$ of block A is received instantaneously by the input pin $sInB_i$ of block B connected at the other end of a relation. Therefore the clocks *sOutAj* and *sInBi* must coincide:

```
Relation[Coincides](sOutAj, sInBi)
```

Based on these rules, a script generates automatically the constraints needed for any SDF model. On the window system, we obtain the following constraints:

```
Clock physicalTime:Dense
Clock superSDF
// "Position" block
Clock updatePosition
Relation[SubClock](updatePosition, superSDF)
Clock sInPos
Clock sOutPos
Relation[Coincides](updatePosition, Sup(sInPos))
Relation[Coincides](updatePosition, sOutPos)
[...] // same thing for the "EndStop" block
// Relation: sOutPos ==> sInEnd
Relation[Coincides](sOutPos, sInEnd)
```

As for DE, we also need to represent data in SDF to be able to simulate the behavior of the model. In the window system example, two clocks are associated with the two possible values of tokens produced by the *sInPos* input pin: *sInPos0* for value "0" and *sInPos1* for value "1". We do the same for the possible values at the *sOutPos*, *sInEnd* and the *sOutEnd* pins.

Figure 6 shows the evolution of *sOutPos* and *sOutEnd* with respect to *sInPos*.

6.4 Semantic Adaptation between DE and TFSM

Semantic adaptation between the DE and TFSM models boils down to a set of 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
Relation[Coincides](ctrlUCmd1, evtUUp)
Relation[Coincides](ctrlUCmd0, evtUNeutral)
Relation[Coincides](ctrlWEnd, evtWEnd)
// Adaptation of outputs
Relation[Coincides](evtWUp, ctrlWCmd1)
Relation[Coincides](evtWStop, ctrlWCmd0)
// chronoDE is periodic on chronoTFSM with period 2 and offset 0
Expression chronoDE = Periodic(chronoTFSM, 2, 0)
```

Figure 6 shows an example of adaptation between the DE signal *ctrlUCmd* with value "1" and the TFSM event *evtUUp* (dashed arrow).

6.5 Semantic Adaptation between DE and SDF

Since the SDF model of the window is periodic, the semantic adaptation between DE and SDF must enforce the fact that the SDF model is updated every T ticks of *chronoDE*, and not at other instants. This also implies that if an event is present on the *winCmd* pin at an instant when the model should not be updated, the event must be memorized until the next update of the SDF model.

The first relation in the listing below states that *superSDF* is periodic on *chronoDE* with period 2. The next two lines specify that the *sCmd* pin receives a new value (*sCmd0* or *sCmd1*) only on ticks of *superSDF*, and keeps the last value until a different one is produced. The initial value of *sCmd* is set to 0.

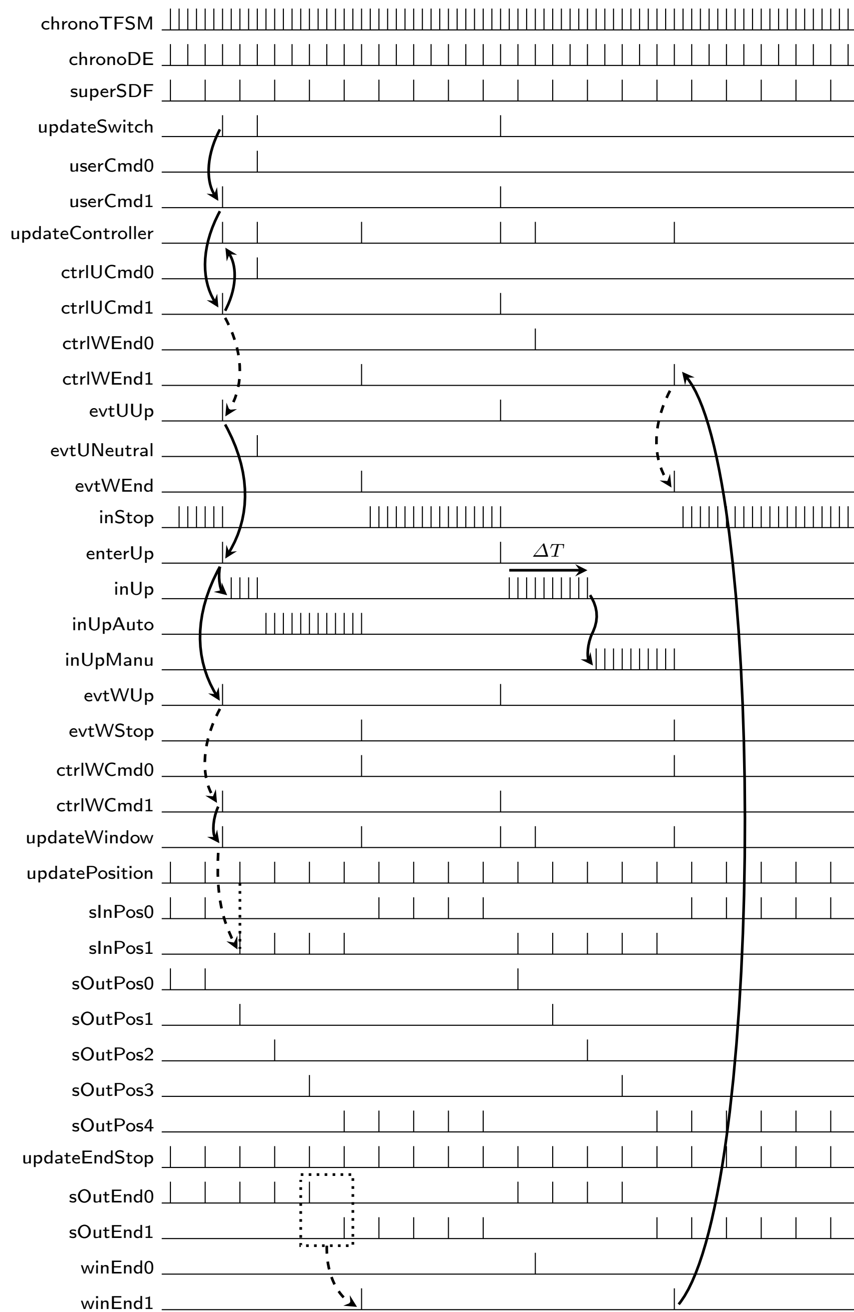

Fig. 6. Simulation of the overall model. Solid lines represent causal relationships within a given model; dashed lines represent *adaptation* between MoCs. Some clocks are omitted to make the figure clearer.

The adaptation of the output is described by the last two lines. In order to detect when *sOutEnd* goes from 0 to 1, and to generate an "on" *winEnd* DE event, we compute the intersection of *sOutEnd1* with *sOutEnd0* delayed by one sample. A similar calculus on *sOutEnd1* allows us to generate "off" *winEnd* events each time the window leaves the end stop position. Notice the delay on the output events to avoid an instantaneous dependency loop in the DE model.

```
// The activation of the SDF model is periodic
Relation[Coincides](superSDF, Periodic(chronoDE, 2, 0))
// The value of the input signal is the value of the last DE event
Relation[Coincides](sCmd0, sustainUpTo(superSDF, Union(initial, winCmd0), winCmd1))
Relation[Coincides(sCmd1, sustainUpTo(superSDF, winCmd1, winCmd0))
// A DE event is generated only when the output of the model changes
Relation[Coincides](winEnd0, DelayFor(Intersection(sOutEnd0,
                                DelayFor(sOutEnd1, superSDF, 1)), chronoDE, 1))
Relation[Coincides](winEnd1, DelayFor(Intersection(sOutEnd1,
                                DelayFor(sOutEnd0, superSDF, 1)), chronoDE, 1))
```

Figure 6 shows how the *ctrlWCmd* event induces a change of the value of *sInPos*, only on the next tick of *superSDF*. The dotted square shows how a change in the value of *sOutEnd* translates into a *winEnd* event. Globally, Figure 6 shows that the overall model is correctly simulated using CCSL: we see the controller change state, and command the window to go up, as well as the window model calculate the successive positions of the window until the end stop is reached, causing the controller to finally rest in the *Stop* state. Some clocks have been omitted for clarity.

7 Discussion

The above results show some benefits and drawbacks of our approach. We were able to obtain concise CCSL specifications for MoCs, which is an improvement over the lengthy descriptions of MoCs 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 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. For instance, if the adaptation constraints specify that the DE clock is 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.

One limitation of this clock-based approach is that CCSL lacks primitives for manipulating data. Therefore, we had to define an ad-hoc methodology for it, which is not satisfactory. Another issue is the integration of this approach in ModHel'X. TimeSquare's solver is a static solver, which computes solutions over the whole timespan. It is not possible to compute the ticks *at runtime*. Therefore,

we cannot use TimeSquare directly in ModHel'X. For the time being we cannot use the mechanisms that exist in ModHel'X to handle the adaptation of data together with CCSL specifications for the adaptation of control and time.

The following section compares this paper's proposal with existing approaches.

8 Related Work

As stated earlier, this paper is inspired by the work by André et al. [9]. 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 additional 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 integrated in ModHel'X yet (as exposed in Sect. 7), 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 [2,1]) like Lustre, Esterel and Signal use abstract logical time and introduce the notion of multiform time. Other approaches, like Lucid Synchrone [3], 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 Simulink/Stateflow. Ptolemy II [7] 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, those rules are hardcoded in the kernel. The modeler has either to rely on default adaption and design his system accordingly, or to add adaptation blocks explicitly into the models themselves, which 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 case study about a similar power window, 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

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

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 [6]. 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 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.

9 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 to ModHel'X, our framework for model composition, an approach proposed in [9]. This paper contains three 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 adapters. Moreover, the TimeSquare solver allowed us to check the consistency of the semantic adaptation between pairs of 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

1. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proc. of the IEEE 91(1), 64–83 (2003)
2. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science Of Computer Programming 19(2), 87–152 (1992)
3. Biernacki, D., Colaco, J.L., Hamon, G., Pouzet, M.: Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In: Proceedings of LCTES (2008)
4. Boulanger, F., Hardebolle, C., Jacquet, C., Marcadet, D.: Semantic Adaptation for Models of Computation. In: Proceedings of ACSD 2011. pp. 153–162 (2011)
5. Brooks, C., Lee, E.A., Liu, X., Neuendorffer, S., Zhao, Y., Zheng, H.: Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains). Tech. Rep. UCB/EECS-2008-30, University of California, Berkeley (2008)
6. Cassandras, C.G., Clune, M.I., Mosterman, P.J.: Hybrid system simulation with SimEvents. In: Proceedings of ADHS. pp. 267–269 (2006)
7. 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), 127–144 (2003)
8. Hardebolle, C., Boulanger, F.: Exploring multi-paradigm modeling techniques. SIMULATION 85, 688–708 (2009)
9. 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, 99–106 (2010)