Semantic Adaptation for Models of Computation

Frédéric Boulanger, Cécile Hardebolle, Christophe Jacquet, Dominique Marcadet
Supelec Systems Sciences (E3S)
Computer Science Department
Gif-sur-Yvette, France
<firstname>.<lastname>@supelec.fr

***Abstract*—In the context of Model Driven Engineering, models are the primary artifacts of the system development cycle. In order to manage the complexity of systems, models are decomposed into models of simpler subsystems. A major difficulty is to handle the heterogeneity of the different models of computation used for modeling the subsystems. Through the example of a power window system, this article presents an approach to the specification of the semantic adaptation of data, time and control between models of computation. The approach is supported by ModHel'X, a heterogeneous modeling and simulation environment. The example is simple enough to be completely described in the article, but rich enough to illustrate the matters of (a) defining models of computation, and (b) specifying the semantic adaptation between models of computation.**

***Keywords*-model driven engineering; model of computation; heterogeneous modeling; model semantics;**

I. Introduction

Modeling is the most common way of handling complexity through abstraction. A model of a system is a simplified representation of the system which keeps only the features which are important for a given goal. When building a model of a system, one uses a paradigm or a modeling technique which suits the modeling need. Therefore, different parts of a system may be modeled using different modeling languages, leading to a heterogeneous model of the system [1].

In this article, we are interested in the computation of the behavior of the model of a system, which may be called a "simulation" of this behavior. When modeling an existing system, such a simulation is useful for computing properties of the system without making a real experiment. When modeling a system under design, the simulation allows an early validation of the design of the system, by running tests on the model of the system for example.

However, as we explained in [2], it is difficult to define the behavior of heterogeneous models, first because it requires to precisely define the semantics of each one of the modeling languages used in the model, and second because it requires to define how information is interpreted at the boundaries between the models of the heterogeneous subsystems. Indeed, different modeling paradigms may use different structures for data (arrays, samples, functions), different notions of time (continuous, discrete, periodic, triggering), and different ways of combining the behavior of the elements of a model (sequential, concurrent, synchronous, with blocking communications). These semantic components of a modeling paradigm define its underlying Model of Computation (MoC). In this work, we consider the heterogeneity of the modeling languages to be equivalent to the heterogeneity of the models of computation.

This work has been performed in the context of the
EDONA project (http://www.edona.fr/)
and partially financed by the
System@tic Paris-Région Competitiveness Cluster
(http://www.systematic-paris-region.org)

In this article, we briefly introduce ModHel'X [3], a platform for heterogeneous modeling and simulation, and we show on a concrete case study (a) what heterogeneous model composition is, (b) how heterogeneous model composition is performed in ModHel'X and (c) how ModHel'X compares to other tools. The example of a power car window is used throughout the paper; it is inspired by a case study [4] which was conducted using tools by The MathWorks. This example helps us illustrate how the semantic adaptation between heterogeneous models can be broken down into three adaptation primitives – the adaptation of data, the adaptation of control and the adaptation of time – which is our main contribution in this paper.

Section II introduces the power window system used as an example. Section III presents ModHel'X, and then Section IV shows in details how the power window system is modeled using ModHel'X, with a special focus on the mechanisms which allow the composition of heterogeneous models. We discuss the results obtained on this case study and we compare with different modeling approaches in Section V.

II. The Power Window System

In the example power window system, the user has a three-position button (up/neutral/down) to control the window. The stable position of the button is *neutral*. When the user moves the button in the *up* or *down* position, the window goes up or down. If he/she briefly switches the button to the *up* or *down* position, and then releases it, the window goes up or down automatically until fully closed or opened. When the window is going up, if it encounters an obstacle, it goes down for two seconds so as to get away from the obstacle, and then stops.

The model of the global system is shown on Figure 1. The components of this system, in dark gray on the figure, are designed by different people from different technical domains (electronics, control science, mechanics, etc.) who use different modeling languages and tools. The role of the components in lighter gray is to feed the model with simulation scenarios and to observe its behavior, display it and check it against what is expected.

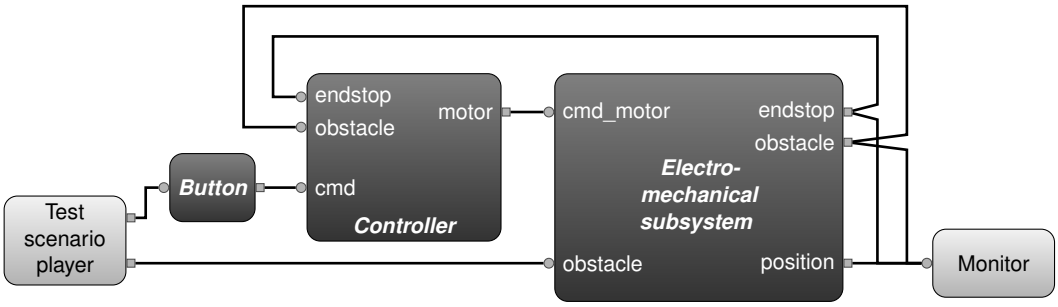

Fig. 1. Overview of the system. Lines are connections via the bus.

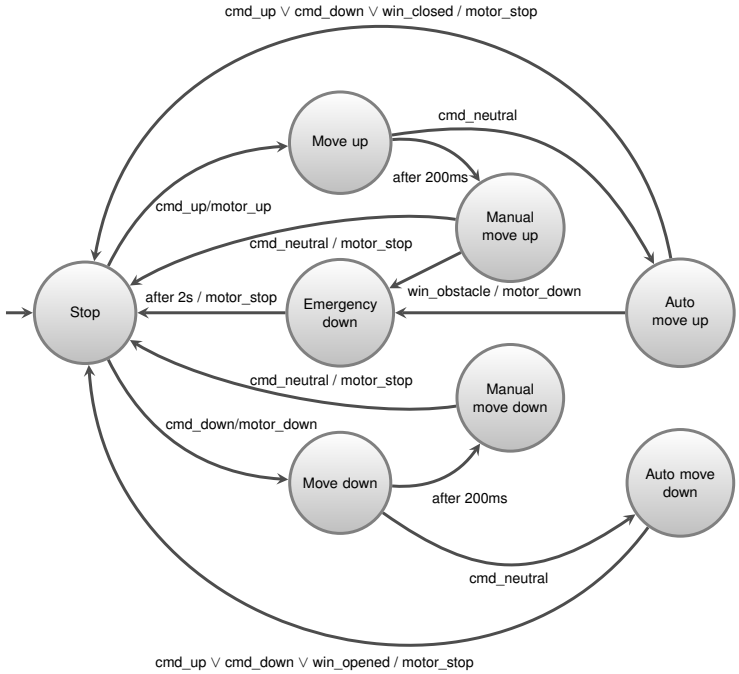

Fig. 2. Finite state machine that specifies the behavior of the window controller.

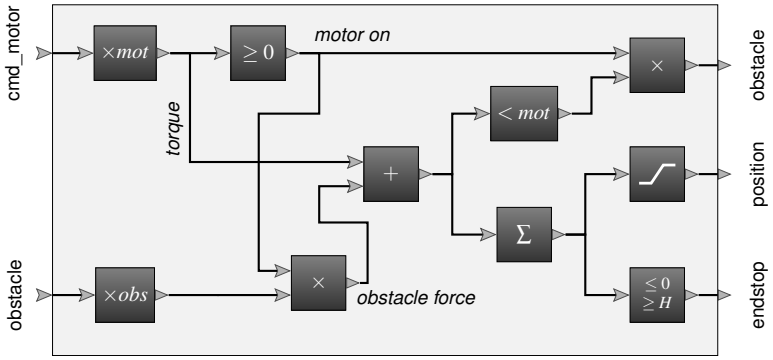

Fig. 3. Dataflow model for the electro-mechanical behavior of the window.

This global model represents the subsystems and their interconnection. In a car, communication between components is multiplexed onto a bus which is represented by the set of lines which connects the controller, the button, and the electro-mechanical subsystem. In the following paragraphs, we detail the models of the controller and of the electro-mechanical system.

The simplified behavior of the controller is given on Figure 2 as a state machine. Since some actions have to be taken after a given delay (if the user releases the button *quickly* then the raising or lowering of the window is automatic), the state machine representing the controller has *timed transitions* that fire automatically after a given delay.

A realistic model of the electro-mechanical subsystem of the power window would probably involve modeling languages specific to electrical and mechanical modeling. For the sake of simplicity, we use a single dataflow model which simulates the behavior of the whole electro-mechanical subsystem. This model receives on its *cmd_motor* input the information about the command of the motor (stopped, forward or reverse). When the motor is on, this subsystem calculates the resulting force applied to the window, taking into account the presence of a potential obstacle (*obstacle* input). If the resulting force is less than expected (i.e. less than the force delivered by the motor), an *obstacle* output message is sent onto the bus. The subsystem also calculates the current position of the window, and simulates the behavior of *end stops* on the path of the window. The dataflow model for this subsystem is presented on Figure 3. On this figure, *mot* is the value of the vertical force produced by the motor, *obs* is the value of the force produced by the obstacle, and H is the height of the window.

III. ModHel'X

To complete this case study, we have used ModHel'X for two different tasks: (1) modeling the behavior of the different components of the power window system using different modeling paradigms and (2) assembling the models of the components to obtain a model of the whole system and simulate its behavior. A global description of ModHel'X can be found in [5]. The present paper focuses on the mechanisms for assembling the models of the components, and contains only a summary of the notions required to understand the example.

At the core of ModHel'X is a generic meta-model for describing the structure of models, and a generic execution engine for interpreting such structures. In ModHel'X, the interpretation of a model by the generic execution engine is directed by a *model of computation (MoC)*. A model of computation is a set of rules for combining the behaviors of a set of components into the behavior of a model. Synchronous data-flows, state machines and continuous time [6] are examples of models of computation. In ModHel'X, a model of computation dictates the rules for scheduling the observation of the components of a model, for propagating values between components, and for determining when the computation of the reaction of a model to an input is complete. The concept of model of computation is essential because it allows ModHel'X to support different modeling languages, and to execute heterogeneous models i.e. models composed of sub-models described using different modeling languages. In order for ModHel'X to support a given modeling language, an expert of this language must describe the corresponding model of computation. Once described in ModHel'X, this model of computation is used by the generic execution engine in order to interpret any model described using the chosen modeling language.

A. Struture elements

The ModHel'X meta-model may be seen as a very *abstract* modeling language that allows one to describe *concrete* modeling languages (models of computations), and to create models.

The elementary unit of behavior in ModHel'X is the *block*. The elements X, Y, A and B on Figure 4 are blocks. A block is defined (a) by its interface which is composed of *pins*, and (b) by *an update operation* which allows the observation of the behavior of the block through its interface. To observe the behavior of a block, input information is put on its pins, and its update operation is executed. In response, the block reads the data available on its pins and updates its outputs accordingly. A *token* is an elementary unit of information which can be observed on a pin.

Blocks can be assembled by defining *relations* between their pins. A *composite block* is a block which is composed of a set of blocks with relations between their pins (see figure 4). Its interface is composed of some of the pins of its blocks. A composite block defines a *structure* as a graph of interconnected blocks. In order to define the semantics of this structure, one must specify the *model of computation* used to drive the graph. A composite block associated with a model of computation forms a *model*. The blocks appearing in the structure of a model may be *atomic blocks*, whose behavior is defined outside ModHel'X (their update operation is opaque); composite blocks made of other blocks; or *interface blocks*, which allow the behavior of a block to be defined by a ModHel'X model.

Interface blocks are a key element of the meta-model of ModHel'X because they provide support for hierarchical composition of models and heterogeneity. An interface block embeds a model, and allows its use as a block in another model. Like any other block, an interface block has a set of pins and an update operation, but it also possesses: (i) an *internal model*, which defines the behavior of the block; (ii) a set of *internal relations*, which connect the pins of its interface to the pins of its internal model; (iii) *two operations*: adaptIn and adaptOut. The internal relations of the interface block along with its adaptIn and adaptOut operations define the semantic adaptation performed by the interface block.

The internal model may use a model of computation that is different from the (external) one used by the model containing the interface block. As such, the interface block acts as an adapter for its internal model with respect to the (external) model in which it is used. The interface block IB on Figure 4 is used to assemble two models called "model" and "internalModel", which have different models of computation,

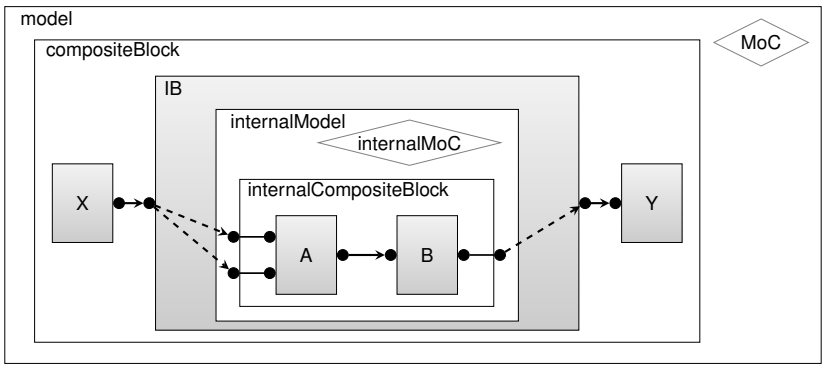

Fig. 4. Hierarchical composition using an interface block.

respectively "MoC" and "internalMoC". The model resulting from the composition is therefore a heterogeneous model. *Hierarchy through interface blocks is the structural mechanism used to compose heterogeneous models in ModHel'X.*

The following section explains how heterogeneous models obtained by hierarchical composition using interface blocks are executed by ModHel'X according to the interpretation rules defined by the MoCs, and to the semantic adaptation specified in the interface blocks.

B. Execution algorithm

To compute the behavior of a model made of elements which are themselves modeled using different modeling paradigms, ModHel'X relies on a black box approach. The behavior of a block may be described by an internal model, but it can only be observed at its interface, i.e. on its pins, by using the update operation (see section III-A). In this approach, computing the behavior of a model consists in (a) observing the behavior of the blocks which are part of its structure and (b) composing these observations according to the set of rules of the MoC used for the model.

The general algorithm of ModHel'X consists in computing a series of snapshots. The *snapshot* operation computes an instantaneous observation of a model. The algorithm first determines the time of the current snapshot, then schedules a block for update, propagates data to its pins, updates the block and then propagates new data from the block to their destination. The schedule-update-propagate loop is repeated until the status of all pins is known and the snapshot is completely determined. Computing a snapshot can be considered as computing the fixed point of the combination of the behavior of the blocks (*update* operation) according to the rules of the MoC (*schedule* and *propagate* operations).

This algorithm is generic with respect to the different modeling paradigms because *the scheduling and the propagation operations are defined specifically by each model of computation*. Therefore, the choice of the order in which blocks are observed and the meaning of relations between pins are given by the models of computation. Examples of models of computation described in ModHel'X are given in section IV-A.

Let us review how ModHel'X handles time, and how an interface block handles heterogeneity.

1) Time handling: To enable the use of various clocks in different parts of a system and of various notions of time in different models of computation, ModHel'X relies on a model of time inspired by that of the MARTE UML profile [7], [8]. It is therefore possible in ModHel'X to have MoCs where the notion of time is reduced to "now", as well as MoCs with durations between clock instants and MoCs where elapsed time can trigger behaviors.

Since a snapshot is an instantaneous observation of a model, time is constant during a snapshot, and the date of a given snapshot must be computed according to each clock of the model. Time computation relies on relations between clocks and on *time constraints*, which may be posted by blocks to constrain the instant of their next update with respect to a given clock. For instance, constraints can be used to trigger periodic behavior by posting a request for an update at the current time plus the period of the behavior. It can also be used for more complex features such as timers or watch dogs. Examples of use of time constraints are given in section IV-B about interface blocks.

2) Heterogeneity and interface block adaptation: An interface block is scheduled by its external MoC and receives and produces data in it. However, it has to schedule, feed data to, and get data from its internal model according to the internal MoC. Therefore, *an interface block performs an adaptation between the semantics of its internal and external models of computation*. This adaptation has three aspects:

- adaptation of data, since different models of computation may have different data structures for the tokens;
- adaptation of time, since there are numerous notions of time, from the sequential numbering of discrete instants to continuous time with a notion of duration between instants;
- adaptation of control, which allows a subsystem to be observed at a different rate than the system in which it is embedded.

Adaptation of data is performed by the adaptIn and adaptOut operations specific to interface blocks. AdaptIn uses the valuation of the pins of the interface block to compute status information and a valuation of the pins of the internal model. AdaptOut uses status information and the valuation of the pins

of the internal model to give a valuation of the pins of the interface block.

Adaptation of time relies on the relations between the clocks of the different models for computing the date of the current snapshot on each of these clocks.

Adaptation of control is the most complex form of adaptation. It must trigger observations of the internal model exactly at instants requested by the internal MoC. For instance, if a model is designed to work with a period of 20 ms, it must be updated exactly and only at instants with timestamps separated by 20 ms. If such a model is embedded in a model which needs to be updated only in response to sporadic events, it is mandatory to update the subsystem every 20 ms even when there is no event for the embedding model, and it is mandatory not to update the subsystem when the embedding model is updated to process an event which occurs at an instant which does not belong to the 20 ms clock.

ModHel'X supports adaptation of control by allowing blocks to post time constraints on their next update, and by letting interface blocks choose to update or not their internal model according to an adaptation policy. This choice is based on the interface block's available inputs, state and current time.

In summary, an interface block performs data, time and control adaptation between two models of computation. Next we show how these operations combine with the general execution algorithm of ModHel'X.

3) Hierarchical execution: Let us now explain how a heterogeneous model like the one on Figure 4 is executed using the general algorithm of ModHel'X. The focus is set on the operations of the interface block so as to show how the execution flow runs hierarchically through all the layers of a heterogeneous model.

When ModHel'X computes a snapshot of the top-level model, the associated MoC begins by computing its own current date and then it collects the possible time constraints of all the blocks of the model. When the MoC collects the time constraints of an interface block such as IB, this interface block may first emit its own time constraints, but it also triggers the computation of the current time of its internal MoC. The internal MoC, in turn, computes its own current date and collects the time constraints successively on each of the blocks of the internal model. In this way, the computation of the current time runs through the whole hierarchy of a heterogeneous model.

After computing the date, the MoC of the top level model enters the loop which computes the global observation of the model, i.e. the snapshot. In this loop, the MoC schedules a block to update, updates the chosen block and then propagates the observed information to other blocks of the model structure. This loop ends when the snapshot is complete. As specified earlier, the scheduling and propagation rules depend on the semantics of the MoC. In the same way, the criteria for determining if the snapshot is complete also depend on the semantics of the MoC. Concrete examples of execution rules are presented in section IV. The sequence diagram of Figure 5 shows the update of the interface block IB. When IB is updated by its external MoC, it uses its adaptIn operation to adapt the

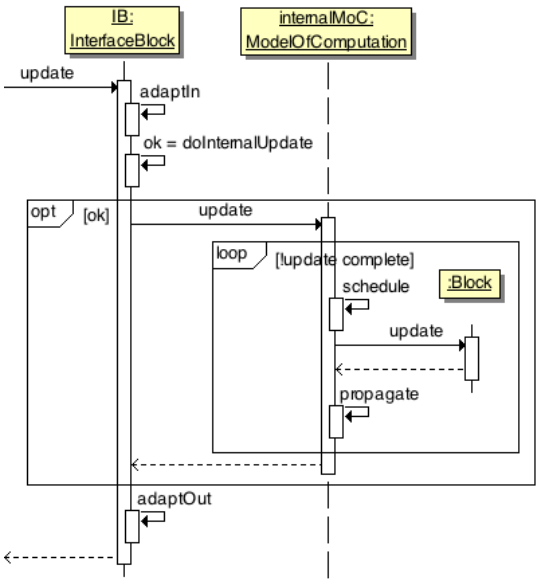

Fig. 5. Update of "IB"

data present on its pins and to provide the adapted data to its internal model. Then, IB has to determine if, according to its adaptation policy (*doInternalUpdate* operation), an update of its internal model is necessary. If so, it delegates the update of its internal model to the internal MoC. To compute an observation of the internal model, the internal MoC uses the same schedule-update-propagate loop as the MoC of the top-level model, but it applies its own scheduling and propagation rules. Last, the interface block has to adapt the data produced by its internal model using its adaptOut operation so that it can be used back into the top-level model.

This illustrates how interface blocks allow the hierarchical execution of the generic algorithm of ModHel'X by delegating operations to their internal model, and how they handle heterogeneity through their specific adaptation rules. Any number of models can be composed hierarchically by pairs using interface blocks. The execution flow runs through all the layers, and semantic adaptation is performed at each border between two models by the corresponding interface block. In the next section, we present how this applies to the concrete case of the power window example.

IV. The Power Window Model

Since ModHel'X allows the combination of models which obey different models of computation, it is possible to choose the most appropriate MoC for modeling each subsystem of our example, and to combine them to obtain a global model. We describe which MoC we use for each model in section IV-A and we determine how to compose the different heterogeneous models in section IV-B.

A. Models of computation at play

The controller is modeled using a timed finite state machines (TFSM) MoC, as discussed in Section II. The electro-mechanical window subsystem is modeled using the *Syn-*

chronous Data Flow (SDF) MoC. The bus connecting the different parts of the system can be considered as a medium for sending time-stamped messages, whose semantics is captured by the *Discrete Event* (DE) MoC. In the remainder of this section, we introduce these models of computation and show how they are implemented in ModHel'X.

1) Timed Finite State Machine Model of Computation (TFSM): The TFSM model of computation gives the semantics of finite state machines with timed transitions, as described in Section II.

In ModHel'X, states and transitions are represented by blocks, which are the elementary units of behavior. Transitions are composite blocks containing a block for the guard and a block for the action performed by the transition. The TFSM MoC maintains a reference to the *current state* of the automaton. When performing a snapshot, the MoC interprets the structure of the model by scheduling the guards of the transitions which leave the current state. If a guard is satisfied (i.e. produces *true* when updated), the MoC schedules the associated action which produces the tokens to be provided as outputs of the TFSM model during its update. The MoC then sets the new current state to the target state of the transition. When entering a state with timed outgoing transitions, the TFSM MoC issues a time constraint for its next update at the earliest dead-line of the timed transitions. Therefore, the timed transition will be triggered, even if no input event is available for the state machine. The exact details depend on the notion of time in the embedding model and will therefore be given when dealing with interface blocks in section IV-B.

The representation of the model of the controller state machine using ModHel'X blocks is not given in this paper because it is meant *for machine use only*. Designers use the usual graphical syntax for automata shown on figure 2.

2) Synchronous Data Flow Model of Computation (SDF): SDF [9], [10] gives the semantics of graphs composed of blocks that exchange flows of data tokens through pins. There may be several data tokens on a pin at the same instant, and the blocks always produce or consume the same number of tokens on a given pin each time they are updated (hence the *synchronous* nature of SDF). Each pin has a *data rate*, and a block may be updated only when all its input pins have at least as many data tokens as their data rate. During an update, a block consumes the specified number of tokens on its input pins and produces the specified number of tokens on its output pins.

Although static scheduling of SDF models is possible, ModHel'X uses a dynamic scheduler. First, the MoC determines which blocks have all their inputs available in sufficient quantity. These blocks are updated, and the tokens produced are propagated to the input pins which their output pins are connected to. This schedule-update-propagate behavior is repeated until the number of data tokens on each pin comes back to its initial value. In order to minimize the number of updates necessary to come back to the initial number of tokens on each pin, when several blocks are ready to be updated, the MoC schedules the one which has been updated the least number of times since the start of the snapshot.

The SDF model of the electro-mechanical subsystem of the power window is shown on Figure 3. The components of this model are atomic blocks, which are blocks whose behavior is defined using tools external to ModHel'X.

3) Discrete Events Model of Computation (DE): DE [6] is used in the example to model the exchange of messages on a bus. Each message has a value and a time-stamp, and if several messages have the same time-stamp, they are delivered in a sequence of *microsteps* (determined by a topological ordering of the blocks), so that the overall observation at that time is causal and deterministic. The scheduling algorithm for DE in ModHel'X relies on a global event queue. At a given instant, the MoC looks for all the events e_i with the smallest time tag t_{now} and advances the current time to t_{now}. It then looks for the blocks b_j which are the targets of the e_i events and schedules one of the minimal elements among the b_j according to the topological ordering of the blocks. The choice of a minimal element guarantees that events produced at t_{now} during the update of a block can be processed by their target at t_{now} in only one update operation. If there are loops in the connection graph of the model, there may not be any minimal element according to the topological order. Our implementation of DE in ModHel'X allows loops if they are not instantaneous, i.e. if at least one block in the loop produces an event with a timestamp in the future. The snapshot is complete when no event with time-stamp t_{now} remains in the queue.

Building the DE model shown on Figure 1 using ModHel'X is straightforward. The controller and the electro-mechanical subsystems are modeled by interface blocks which embed the corresponding TFSM and SDF models. The light gray components are atomic blocks used for feeding the model with a simulation scenario and observing the outputs of the simulation. The next section describes the semantic adaptation performed by the two interface blocks.

B. Interface blocks at play

We saw in section III-B that interface blocks perform semantic adaptation according to three aspects: data, time and control. For instance, between DE and SDF, one has to adapt data from events (in DE) to samples (in SDF). Regarding time adaptation, SDF has no notion of time beyond the succession of data samples, while DE puts a time-stamp on each event. A simple and useful adaptation of time between SDF and DE is to use a sampling period T to compute the time in DE from the succession of instants in SDF. The SDF model is considered to produce data instantaneously from its inputs, but it is updated every T units of time in DE. This adaptation of time must be consistent with the adaptation of control, so when an SDF model is embedded in DE, control must be adapted so that the SDF model is updated periodically, even when the DE model has no event to process.

The adaptation is to some extent *specific* to the models involved. In the DE/SDF example above, the sampling period is specific to a given application. Therefore one cannot build a universal DE/SDF interface block, completely independent of

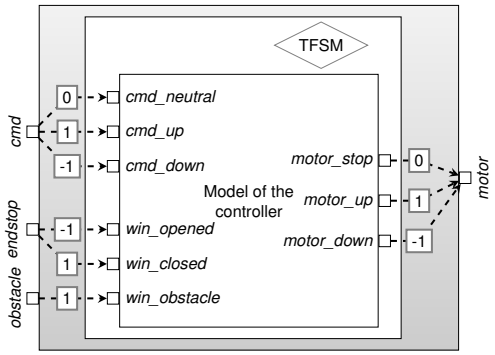

Fig. 6. DE/TFSM interface block used in the power window model.

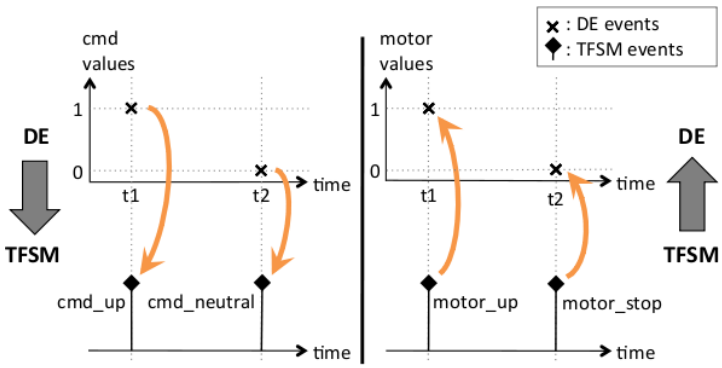

Fig. 7. Example of DE/TFSM adaptation.

the models involved. However, sampling is a general adaptation policy which can be implemented in a *parameterized interface block*. Such adapters are generic, and are parameterized by the designer when used in a model. When a more specific adaptation is necessary, it is still possible to build an adapter specifically tailored to a given pair of models. In the following, we present the two parameterized interface blocks involved in the power window model.

1) DE/TFSM interface block: The DE/TFSM interface block allows the use of finite state machines as DE blocks. In input, DE events are translated into TFSM events. TFSM actions are translated into DE events in output. ModHel'X relations between pins of the outer DE block and pins of the inner TFSM model are used to define correspondence relationships between DE events and TFSM events and actions. So, this adapter is generic and can easily be adapted to various situations by creating the required relations.

However, a DE event carries a value whereas a TFSM event or action has no associated value. Therefore, a given DE event may trigger various TFSM events, depending on the carried value. To account for this, a given input pin of the external DE block may be connected (via relations) to several input pins of the internal TFSM model, as in a demultiplexer. To discriminate between several possible TFSM events for a DE event, each relation has a parameter called "AssociatedValue". When a DE event is received on an input pin of the interface block, the *adaptIn* operation tries to match the carried value with the *AssociatedValue* of the corresponding relations. A TFSM event is created when one of them matches. To be able to sort out between errors and voluntary ignorance of values when none of the "AssociatedValue" matches, we use a parameter called "IgnoredValues". The solution is similar for the outputs.

Figure 6 shows the instance of the DE/TFSM interface block which embeds the model of the controller in the model of the power window system. The figure shows the "AssociatedValue" parameters used for multiplexing and demultiplexing DE values. Figure 7 illustrates the adaptation performed by the DE/TFSM interface block on example data. The left part of the figure shows how an event on the "cmd" DE input pin is interpreted and translated into a "cmd_up" or a "cmd_neutral" TFSM event according to its value (respectively 1 and 0 in this example). The right-hand side of figure shows how the "motor_up" and "motor_stop" TFSM events are translated into a "motor" DE event with value 1 or 0 respectively.

When dealing with non-timed transitions only, the behavior of the TFSM model is totally controlled by the surrounding DE model. When a DE event occurs, the interface block creates a TFSM event that may fire a transition. Consequently, the TFSM changes state and may trigger an action. In turn, the action may generate a DE event, at the same time as the original (incoming) DE event. The reaction of the state machine to an event is considered as instantaneous, and the interface block just has to memorize the time of the current DE event, so as to use it as a timestamp if the finite state machine triggers an action.

However, when dealing with timed transitions, the TFSM model may trigger actions at instants when there is no incoming DE event. Therefore, when the state machine enters a state which has outgoing timed transitions, the interface block must request (via time contraints) a snapshot at the earliest time at which one of these transitions may occur. In this way, the interface block controls the occurrence of snapshots that account for timed transitions. If a timed transition fires, an action may be produced by the TFSM model. In this case, the timestamp of the associated DE event has to be calculated by the interface block. To do so, the interface block keeps track of the DE time at which the finite state machine has entered the current state. When a timed transition occurs, the interface block adds the DE time at which the current state was entered to the delay associated with the transition in order to get the DE time at which the transition occurs.

This pattern of semantic adaptation is interesting because timed transitions have no meaning *per se* in TFSM since time in this MoC is just the discrete series of instants at which the state machine receives events. Timed transitions in TFSM refer to durations between instants of the clock of the *embedding* model. The semantic adaptation of time therefore contributes to the meaning of the TFSM model by synchronizing the clocks in DE and in TFSM, and cannot be inferred automatically unless an implicit global time reference is assumed, which is not the case in ModHel'X. By requiring an explicit adaptation, ModHel'X

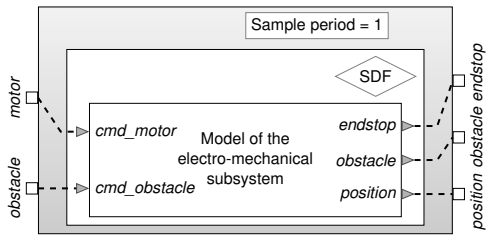

Fig. 8. DE/SDF interface block used in the power window model.

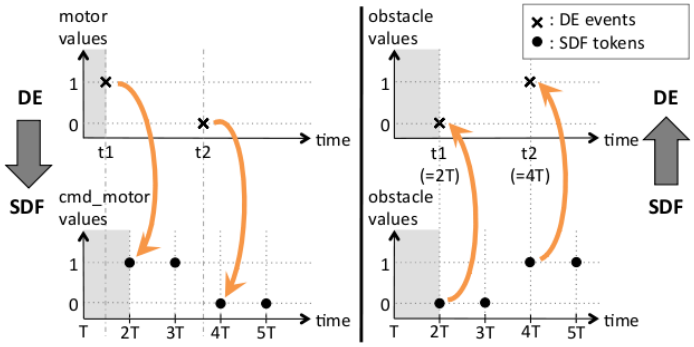

Fig. 9. Example of DE/SDF adaptation

allows for greater flexibility: it is for instance possible to have two sub-models that use the DE MoC in the same ModHel'X model, and for which time does not run at the same speed. This is related to the multi-form nature of time in MARTE: one DE sub-model could count time as the angular position of some rotating shaft, while the other would count time in milliseconds of physical time.

2) DE/SDF interface block: The DE/SDF interface block allows the use of SDF models as DE blocks. It adapts between a world of events occurring at instants defined on a continuous clock and a world of sampled signals. In SDF, the notion of time is reduced to that of *the index of a sample*. The mapping between signal samples and dates in DE can be built according to a *sampling period*. If T is the sampling period used by the interface block, time T elapses in DE between the consumption or production of two successive tokens in SDF. The sampling period is a parameter of the general-purpose DE/SDF sampling adapter.

Even when no event occurs in the DE model, the inner SDF model must be updated at its sample rate. Therefore, a snapshot of the whole model must be computed at least every T in DE time. The DE/SDF interface block uses time constraints in order to request periodic snapshots. Conversely, when an event occurs in DE at a time which does not match a sampling instant of SDF, the interface block doesn't update the internal model. To account for this in the data adaptation performed by this DE/SDF adapter, DE events are considered as notifications of value changes. Therefore, each time a DE event occurs, the interface block memorizes its value as the new value that will be used every T until an event changes the value of the signal again. In output, each data sample is compared to the previous one, and when they differ, a DE event is produced with the new value at the current DE time.

This semantic adaptation pattern illustrates the cross-influence of the data, time and control adaptation aspects. The implicit periodic nature of samples in the SDF model makes elapsed time create control. Changes in the value of the signals computed by the SDF model create events, and therefore control in DE. Although SDF is described as a data-triggered MoC (components are activated when data is available), the interface block updates the SDF model on a time-triggered basis because of the sampling rate.

Figure 8 shows the instance of the DE/SDF interface block which is used in the power window system example. This interface block wraps the SDF model of the electro-mechanical subsystem for use in the DE global model of the power window system. The *sample period* T of the DE/SDF interface block is set to 1 so that the SDF model is updated every unit of the DE time. Figure 9 illustrates the adaptation performed by the DE/SDF interface block on example data. The left part shows how "motor" events are taken into account as changes of the value of the "cmd_motor" input pin of the inner SDF model. These changes are seen by the SDF model only at the next sampling time. The right part of the figure shows how the changes of value of the "obstacle" SDF output are translated into DE events.

V. Discussion and Related Work

Several categories of approaches, like model transformation, language composition or model composition, address the problem of modeling a system composed of heterogeneous components. A classification and a comparison of these approaches is proposed in [2]. In the following, we illustrate the differences between ModHel'X, Ptolemy II [6], Metropolis [11] and the MATLAB/Simulink toolchain by The MathWorks with respect to the way adaptation between MoCs is performed.

Ptolemy II is one of the first approaches for 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 which exist in Ptolemy. One of our contributions regards the adaptation rules at the boundary between two heterogeneous models. In the power window example, the SDF model embedded in the global DE model must have data on all its inputs to be able to compute its behavior (this is the semantics of SDF). Since the semantics of DE doesn't guarantee synchronous data feeding, a semantic adaptation has to be performed. In Ptolemy, this adaptation is coded in the kernel and forbids the firing of the SDF model unless the DE model provides data on all its inputs [9]. The modeler has either to rely on that default adaption and design its system accordingly, or to explicitly add adaptation blocks into the models themselves. For instance, samplers could be added to the DE model in order to feed the SDF model with the value

of the last event. Such artifacts render models less reusable and more difficult to understand. In ModHel'X, the explicit adaptation is insulated from the models and encapsulated into interface blocks.

Another innovation in ModHel'X lies in the handling of different notions of time and multiple control clocks. This issue has been extensively studied in the domain of hardware synthesis. Synchronous languages (see [12], [13]) like Lustre, Esterel and Signal use an abstract logical time, which is only a series of events on a clock, and introduce the notion of multiform time. Other approaches, like Lucid Synchrone [14], have explicit support for specifying multi-clock systems. Ptolemy supports heterogenous notions of time, however, it has a hierarchical structure for time scales in which the top-level model drives the time in the models of its components. In contrast, ModHel'X relies on independent clocks with dynamic constraints between their instants, which allows deeply nested blocks to request an update. It also relies on the filtering of updates by interface blocks to avoid the observation of blocks at instants that do not exist on their activation clock. However, the stricter separation of time, control and data in ModHel'X makes it more difficult to handle modal models as efficiently as in Ptolemy [15] for the moment.

Metropolis [11] is a heterogeneous system design environment which relies on the separation of communication and computation concerns. In a way similar to Ptolemy and ModHel'X, it defines an abstract semantic framework for defining models of computation and communication. Thanks to a formal semantics based on trace algebras, Metropolis can be used both for formal verification and for simulation and code generation. It is difficult to compare how the adaptation between heterogeneous models is performed in Metropolis and in ModHel'X because of two main differences:

1) Metropolis models are made of communicating processes, while ModHel'X has been specifically designed to support the broadest possible set of MoCs, including MoCs for communicating processes but not only, has shown by the example in this paper. As a counterpart, ModHel'X is not able to provide as many analysis methods for models as Metropolis.
2) In ModHel'X, the semantic adaptation between MoCs is made at the border between hierarchical models while Metropolis allows the use of adapters to connect heterogeneous processes at the same level of a model. This means that, while in ModHel'X communication is defined uniformly by the MoC for all ports of all blocks of a model, in Metropolis custom adaptation can be made on each port of a process. As such, Metropolis shares similarities with BIP [16], in which different communication and synchronization operators can be mixed to connect components described as finite state machines.

As a conclusion, ModHel'X and Metropolis have been designed for very different purposes. However, even if they support adaptation between heterogeneous elements of a model in different ways, both Metropolis and ModHel'X support the adaptation of data and control. The support of the adaptation of time is not very detailed in Metropolis but the trace-based semantics on which it relies theoretically allows the use of multiple clocks in a model.

Regarding the MATLAB/Simulink toolchain, a power window case study, available on The MathWorks' website [17], illustrates heterogeneous model composition using Simulink (SDF-like) and Stateflow (TFSM-like) among other tools. The semantic adaptation between a Simulink and a Stateflow models can be specified explicitly using functions and truth tables. However, all MoCs cannot be composed in the same way. For instance, using a Simulink (SDF-like) model into a SimEvents (DE-like) model requires different adaptation artifacts such as event translation blocks [18]. This is not only because the way SimEvents interacts with Simulink is hardcoded in the tool, but also because SimEvents is 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.

Tools which work with a fixed set of MoCs, like those by The Mathworks, are of course more efficient and can rely on predefined semantics to validate models or generate code. However, when the predefined semantics does not match the designer's needs, this can lead to modeling errors or to the introduction of modeling artifacts to work around the built-in semantics of the tools. We do not present ModHel'X as the ultimate modeling tool. On the contrary, we want to show that it is possible to federate heterogeneous models by considering them as black boxes and wrapping them in explicit semantic adaptation boxes. This way, dedicated tools can still be used for each subsystem, and the global behavior of the whole system in response to a simulation scenario can be computed and checked against the expected one.

VI. Conclusion

The example presented in this article illustrates that several models of computation are necessary for describing real-world systems, even of moderate complexity. The ModHel'X framework has been designed specifically to address the composition of models directed by different models of computation. To achieve this goal, ModHel'X supports the precise description of both the models of computation and the semantic adaptation between them.

The adaptation between models of computation is taken into account in an abstract and reusable way thanks to interface blocks. A key point of our approach is the possibility to explicitly model the three aspects of the semantic adaptation between models of computation: adaptation of data, adaptation of time and adaptation of control.

Once a heterogeneous model of a system has been built, it can be executed and test scenarios may be run, allowing an early validation of the design of the system. Beyond simulation, well-defined interactions between models of computation may be used for generating glue code for the semantic

adaptation between models of subsystems, and for the global validation of heterogeneous models through testing and model-checking [19]. In particular, we have developed a formal theory of conformance testing for heterogeneous models in [20] and an algorithm to generate interactive test cases that could be run by ModHel'X.

In this paper, we have dealt only with heterogeneity between the models of different parts of a system. However our framework may be used to handle the heterogeneity of different views of the same part of a system as well, as we have shown in [21]. Such views are used for modeling non-functional aspects of the system, and the approach presented here may be extended for semantic adaptation between them.

REFERENCES

[1] T. A. Henzinger and J. Sifakis, "The embedded systems design challenge," in *Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science*. Springer, August 2006, pp. 1–15.

[2] C. Hardebolle and F. Boulanger, "Exploring multi-paradigm modeling techniques," *SIMULATION: Transactions of The Society for Modeling and Simulation International*, vol. 85, pp. 688–708, November 2009.

[3] F. Boulanger and C. Hardebolle, "Simulation of Multi-Formalism Models with ModHel'X," in *Proceedings of ICST'08*. IEEE Comp. Soc., 2008, pp. 318–327.

[4] P. J. Mosterman and H. Vangheluwe, "Computer automated multi-paradigm modeling: An introduction," *Simulation: Transactions of the Society for Modeling and Simulation International*, vol. 80, no. 9, pp. 433–450, 2004.

[5] C. Hardebolle, F. Boulanger, D. Marcadet, and G. Vidal-Naquet, "A generic execution framework for models of computation," in *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007), at the European Joint Conferences on Theory and Practice of Software (ETAPS 2007)*. IEEE Computer Society, march 2007, pp. 45–54.

[6] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity – the Ptolemy approach," *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, vol. 91, no. 1, pp. 127–144, January 2003.

[7] C. André, F. Mallet, and R. De Simone, "Time Modeling in MARTE," in *ECSI Forum on specification & Design Languages (FDL)*. ECSI, 2007, pp. 268–273. [Online]. Available: http://hal.archives-ouvertes.fr/inria-00204481/en/

[8] OMG, "UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) RFP," february 2005. [Online]. Available: http://www.omg.org/cgi-bin/doc?realtime/2005-2-6

[9] W. Chang, S. Ha, and E. Lee, "Heterogeneous simulation – Mixing discrete-event models with dataflow," *The Journal of VLSI Signal Processing*, vol. 15, no. 1, pp. 127–144, 1997.

[10] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," in *Proceedings of the IEEE*, vol. 75, no. 9, september 1987.

[11] F. Balarin, L. Lavagno, C. Passerone, A. L. S. Vincentelli, M. Sgroi, and Y. Watanabe, "Modeling and designing heterogeneous systems," *Advances in Concurrency and System Design*, 2002.

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

[13] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages twelve years later," *Proc. of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan. 2003, special issue on Embedded Systems.

[14] D. Biernacki, J.-L. Colaco, G. Hamon, and 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)*, Tucson, Arizona, June 2008.

[15] E. A. Lee and S. Tripakis, "Modal models in ptolemy," in *Proceedings of 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT 2010)*, October 2010, pp. 1–11. [Online]. Available: http://chess.eecs.berkeley.edu/pubs/700.html

[16] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time systems in BIP," in *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM06)*, september 2006, pp. 3–12.

[17] The MathWorks, "Automotive power window system," http://www.mathworks.com/products/simulink/demos.html?file=/products/demos/simulink/PowerWindow/html/PowerWindow1.html.

[18] C. G. Cassandras, M. I. Clune, and P. J. Mosterman, "Hybrid system simulation with SimEvents," in *Proceedings of the 2nd IFAC Conf. on Analysis and Design of Hybrid Systems*, 2006, pp. 267–269.

[19] C. Jacquet, F. Boulanger, and D. Marcadet, "From data to events: Checking properties on the control of a system," in *Proceedings of the Sixth ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2008)*, IEEE. Anaheim, California: IEEE Computer Society, June 2008, pp. 17–26.

[20] B. Kanso, M. Aiguier, F. Boulanger, and A. Touil, "Testing of abstract components," in *Proceedings of the 7th International Colloquium on Theoretical Aspects of Computing (ICTAC 2010)*, ser. Lecture Notes in Computer Science 6255. Springer-Verlag, 2010, pp. 184–198.

[21] F. Boulanger, C. Jacquet, C. Hardebolle, and E. Rouis, "Modeling heterogeneous points of view with modhel'x," in *MODELS 2009 Workshops*, ser. Lecture Notes in Computer Science, S. Ghosh, Ed., vol. 6002. Berlin Heidelberg, Germany: Springer-Verlag, 2010, pp. 310–324.