Simulating Rhapsody SysML Blocks in Hybrid Models with FMI

Yishai A. Feldman[1] Lev Greenberg[1] Eldad Palachi[2]
[1]IBM Research – Haifa, Israel [2]Rational, IBM Israel, Rehovot, Israel
Haifa University Campus, Mount Carmel, 3498825 Haifa, Israel
{yishai,levg,eldadpal}@il.ibm.com

Abstract

The Functional Mockup Interface (FMI) standard enables hybrid simulation of models from different tools. Such tools can have different underlying behavioral semantics, creating challenges when models are combined. A case in point is the combination of the Rhapsody tool, widely used to describe and implement discrete control behavior, and Modelica, widely used to describe continuous plant behavior.

This paper describes a plugin we developed for exporting Functional Mockup Units (FMUs) from Rhapsody, and the results of combining generated FMUs with continuous models. When a Rhapsody FMU is used in a different environment, some basic assumptions on its behavior are challenged. We describe the semantic mismatches between the tools, to what extent they can be overcome, and what modelers need to do in order to preserve the intended semantics of an exported FMU.

Keywords: FMI, Rhapsody, SysML, Hybrid simulation

1 Motivation and Overview

Complex cyber-physical systems are composed out of components and subcomponents, often designed and manufactured by different organizations. Each component can come from a wide range of different engineering domains, including mechanical, electrical, control, and software. Each engineering domain uses its own languages and tools; these are often not integrated with each other. This makes it difficult to perform analysis, verification, and design-space exploration at the model level, resulting in errors that are discovered late in the process (typically during integration), and are expensive to fix.

In some cases there is ad-hoc integration between a pair of tools; for example, organizations commonly create connections using ad-hoc scripts. However, this is expensive, brittle, and sometimes wrong. Functional Mockup Interface (FMI)[1] is an open international standard for the integration of models between different tools that may use different underlying semantics (e.g., discrete state machines and differential algebraic equations). It specifies an interface that encapsulates a model as a *Functional Mockup Unit (FMU)*, for communicating with hosting tools. The standard defines two types of export: *model exchange* and *co-simulation*. The main difference between them is that the former uses a solver of the hosting tool, while the latter includes its own solver and the hosting tool only orchestrates the simulation (transfers variables values, orders FMUs invocation, and selects the next communication step size).

SysML[2] and UML[3] are modeling languages standardized by the OMG to describe structure and behavior of systems at various levels of abstraction. Statecharts [6] are a popular formalism for specifying behavior in SysML and UML; they are an extension of finite-state machines, and enable very concise decriptions of finite-state models. They are used to describe reactive systems at a wide range of abstraction, from the details of embedded controllers to high-level manufacturing processes. Some tools, such as IBM Rational Rhapsody®,[4] can execute statechart models, and, in the case of software applications, to synthesize production code from these models. Model execution (sometimes called "simulation") can be used to analyze and verify the design.

However, many modern systems are cyber-physical and include physical models whose behavior is modeled in other languages and tools. In the example of Section 2, a statechart is used to model the controller of a heating system; other aspects of the system, such as the thermal plant, sensors, and actua-

[1]`https://www.fmi-standard.org`.
[2]`http://www.omgsysml.org`.
[3]`http://www.uml.org`.
[4]`http://www-03.ibm.com/software/products/en/ratirhapfami`.

tors, are modeled in Modelica®.[5] In order to simulate the whole system, it is therefore necessary to combine the discrete-event simulation of the statechart model in Rhapsody with the continuous dynamics described in Modelica. This paper describes the way in which an FMU that encapsulates the statechart can be exported from Rhapsody and used inside FMI-compliant tools.[6]

In Section 3 we analyze the semantic differences between the behavior of a SysML block in Rhapsody and that of the exported FMU. We highlight subtle, but in some cases significant differences. The naive expectation is for two communicating statecharts from the same Rhapsody model to retain their behavior when each is exported as an FMU and composed in the same way in a hosting tool. Because of the different conceptual semantics of Rhapsody statecharts and the FMI standard, this is very challenging. It follows that designers of SysML models that participate in hybrid simulations as FMUs need to be aware of this and design appropriately. We present a set of guidelines in Section 3.6.

2 Example

This section presents an example of a hybrid model, in which a controller specified as a statechart is exported as an FMU and used in a Modelica model. Figure 1 shows a Modelica model of a heater[7] in the SimulationX tool.[8] The model consists of the heater plant (bottom), a temperature sensor (displayed as a thermometer), a controller (top, in black) that receives the sensor input and a parameter specifying the desired temperature, and two actuators that can turn on or off two heating elements (lower left, in blue). A switch (labeled "`booleanStep1`" in the middle of the diagram) turns the heating system on or off.

The plant is specified in Modelica using a set of differential algebraic equations, and the sensor and actuators are also simple Modelica models. The controller, however, is specified as a SysML statechart in Rhapsody; it is shown in Figure 2. The controller starts in state `OFF`; it moves to `ON` when the switch signal changes to `true`, and moves back when it changes to `false`.

In state `ON`, the controller keeps track of the number of active heaters (between 0 and 2); in the initial sub-state, `NoHeating`, this is set to zero, and both heaters are turned off. Each of the three states has a timeout transition (labeled `tm(switchingTimeout)`), it is used to sample the temperature sensor periodically. The operation `tempDiff()` returns the difference between the reference temperature (provided as the top input in Figure 1) and the measured temperature (bottom input). The model has two parameters, `tempDiff1` and `tempDiff2`. Whenever the temperature difference is more than `tempDiff1`, the first heater is turned on; if the difference exceeds `tempDiff2`, both heaters are turned on. This is achieved by the transitions between the states, which check the temperature difference. These transitions have just a condition (specified in brackets) but no trigger. They are checked each time the state is entered; that is, periodically whenever the timeout expires.

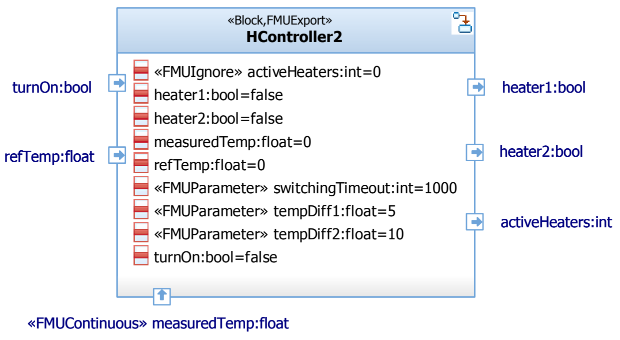

Figure 3: The interface of the controller block.

The interface of the controller is a SysML *block*, shown in Figure 3. It has three inputs, which are SysML *flow ports*: the boolean-valued `turnOn`, for the on/off switch, and two float-valued input ports, one for the reference temperature, and one for the measured temperature. The last is annotated with the stereotype «FMUContinuous» to indicate that it is a continuous input; by default, inputs are assumed to be discrete (see Section 3.5). Each flow port is associated with a block attribute of the same name (shown inside the block); each of these has an associated initial value. The block has three output flow ports: the first two are boolean-valued signals that turn the heaters on and off, and the third is an integer-valued signal that carries the number of currently active heaters. In addition, the block has three parameters, for the switching timeout and the temperature thresholds.

Figure 4 shows the essential parts of the model description in the FMU generated by the plugin from this block. It starts with a type definition describing the `float` type associated with the block's inputs. Since

[5]`https://www.modelica.org`.

[6]See `https://www.fmi-standard.org/tools` for a list of tools that support FMI.

[7]Based on the ControlledTemperature example from the Modelica Standard Library.

[8]`http://www.simulationx.com`.

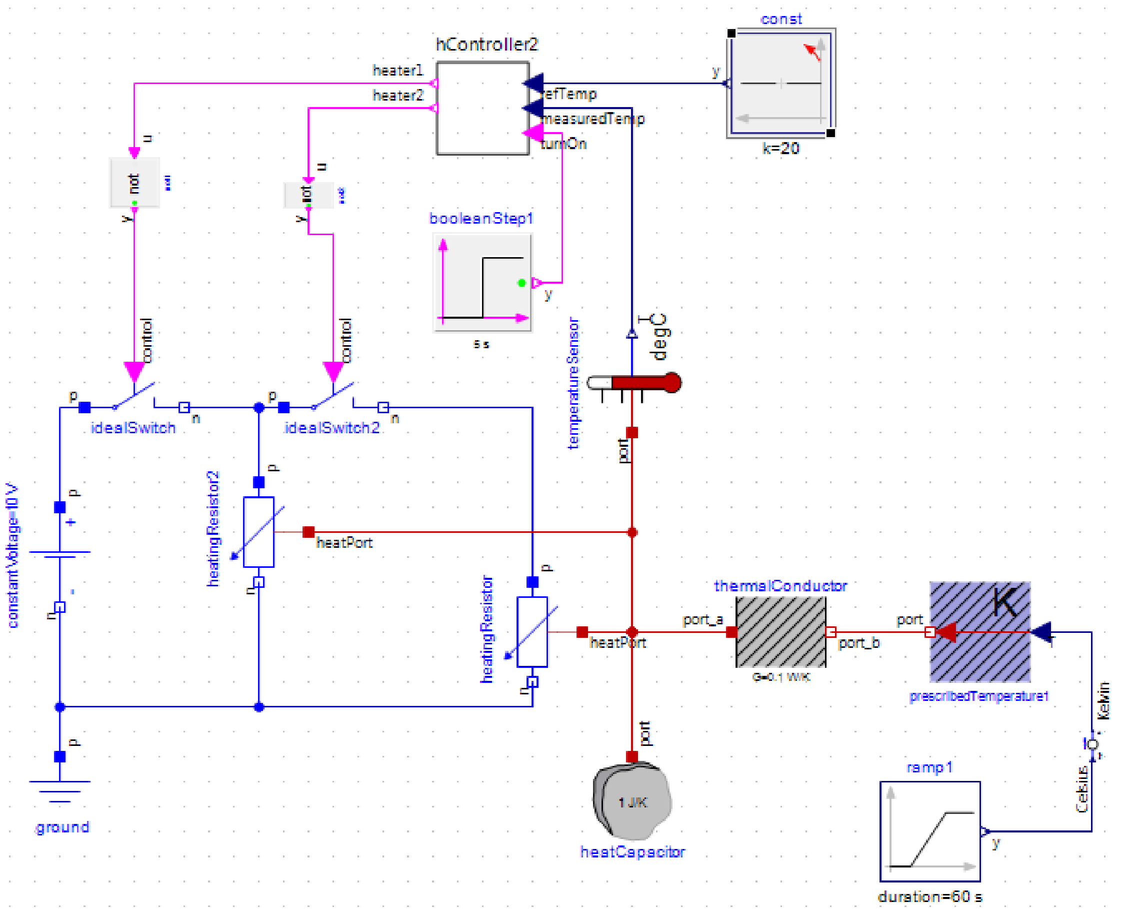

Figure 1: The heater model in SimulationX.

this type defines single-precision floating-point numbers, while the FMI standard defines real numbers using double precision, it is translated into an FMI real type with a constrained range. Following the type definition are the specifications of the model variables. First are the three inputs, the first with a boolean type and the others with the constrained type. All inputs have input causality and discrete variability, except for `measuredTemp`, which is continuous. Following those are the two output booleans, and the three parameters.

In order to let the plugin know how to treat the various block elements, we defined a UML profile containing a set of stereotypes. For example, the «FMUParameter» stereotype denotes the three block attributes that are to be treated as parameters (Figure 3). The signal `activeHeaters` is not exported to the FMU, as indicated by the stereotype «FMUIgnore» annotating the block attribute of the same name. The model of Figure 1 defines how these inputs and outputs are connected to the rest of the model when the exported FMU is imported into the SimulationX environment. Inside SimulationX, the imported FMUs are represented as regular Modelica blocks, so they can be naturally included in any Modelica model.

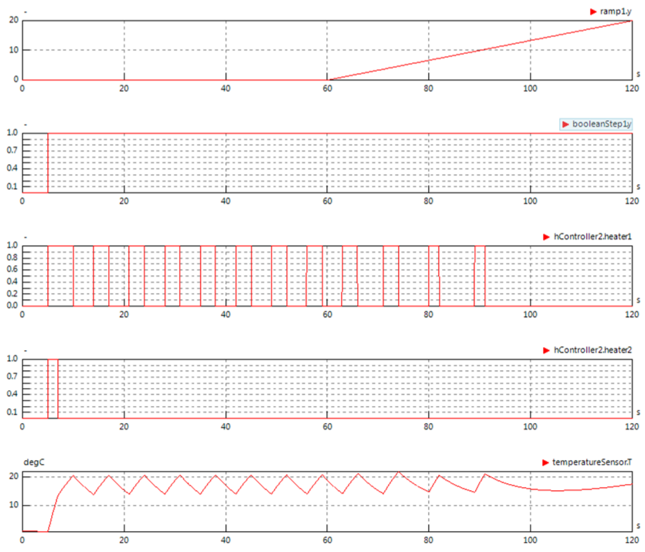

Figure 5: Simulation results with the Rhapsody FMU in SimulationX.

Figure 5 shows the results of the simulation using the exported controller FMU in SimulationX. The top graph shows the `ramp1` variable from the block at the

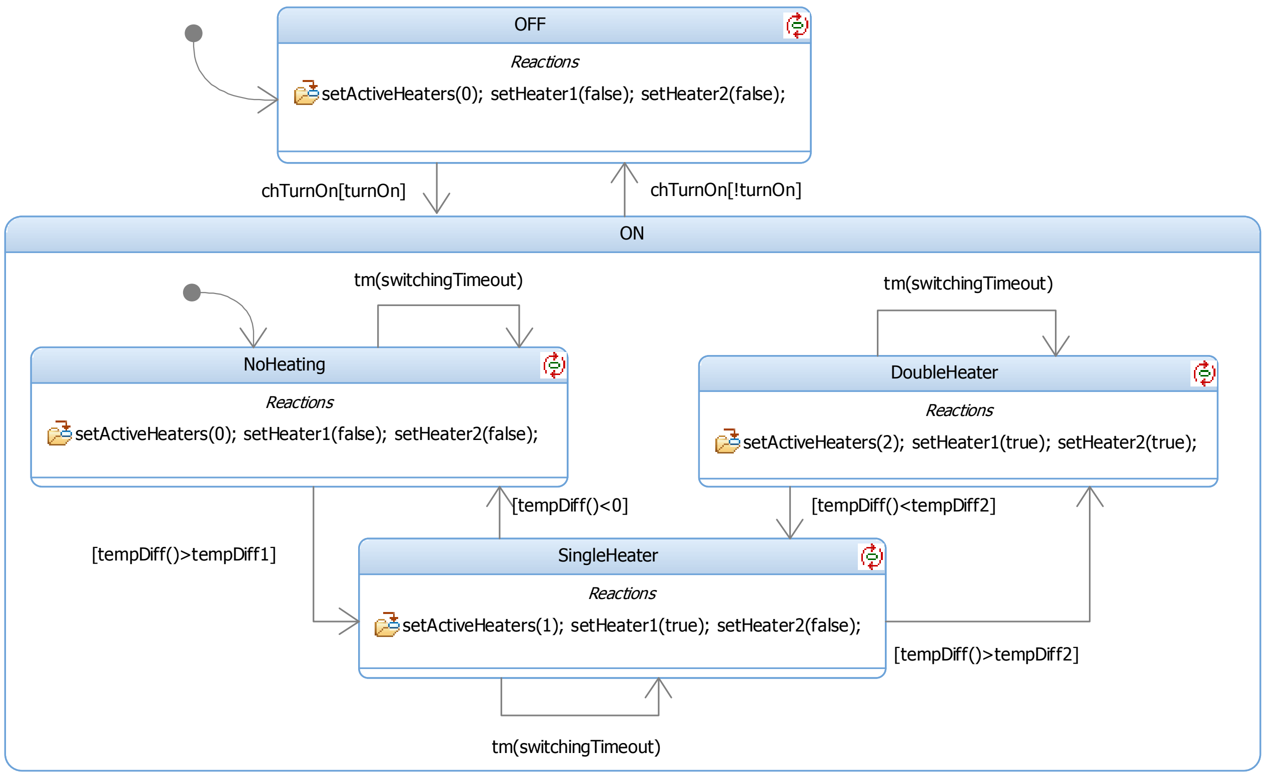

Figure 2: The heater controller in Rhapsody.

top right of Figure 1; this variable represents the ambient temperature, which starts at 0 and climbs to 20 degrees during the second minute. The second graph shows the on/off switch, with is turned on 5 seconds after the simulation starts. The next two graphs show the signals that the controller sends to activate the heaters. When the switch is initially turned on, the measured temperature is zero, and both heaters are activated. The second heater is deactivated after a few seconds, while the first heater is turned on and off according to the measured temperature. The bottom graph shows the values measured by the temperature sensor; the value climbs from 0 to a little over 20 degrees, then seesaws as a result of the activation of the heaters. Toward the end of the second minute, once the ambient temperature is high enough, both heaters are turned off, and the sensed temperature climbs slowly, as expected.

3 FMI for SysML

This section describes the FMU export functionality we developed for Rhapsody as a plugin, and discusses the design decisions and their implications. The FMI standard defines two export modes: *model exchange* and *co-simulation*. The major difference between them is that model exchange requires a hosting tool to provide an ordinary differential equation (ODE) solver to perform the simulation, and therefore the models needs to expose all internal equations. Rhapsody models are discrete and do not need an ODE solver, so co-simulaton would be appropriate; this in turn could enable the use of the FMU in environments that do not provide ODE solvers. Unfortunately, FMI for co-simulation cannot handle discrete events efficiently [2]. For example, the co-simulation API, unlike the model exchange API, does not provide a way to report the next event time. We have therefore chosen to use the model-exchange mode for our plugin. In any case, most of the points discussed in this paper apply to co-simulation as well.

The elements of the SysML block that are exported currently limited to atomic flow ports and attributes. As we saw in the example, input flow ports of the SysML block are exposed as inputs of the FMU, and output flow ports become outputs. Values of attributes of the block that have corresponding flow ports are considered by Rhapsody as storage for the values of the ports, and are therefore represented by the same input or output variables. Other attributes will become FMI discrete internal variables; if they are annotated with «FMUParameter», they will become FMI parameters. Initial values of attributes will be translated into

```
<TypeDefinitions>
  <Type name="real32_Type">
    <RealType min="-3.4028234663852886E38" max="3.4028234663852886E38"/>
  </Type>
</TypeDefinitions>
<ModelVariables>
  <ScalarVariable name="turnOn" valueReference="2" variability="discrete" causality="input">
    <Boolean start="false"/>
  </ScalarVariable>
  <ScalarVariable name="refTemp" valueReference="0" variability="discrete" causality="input">
    <Real declaredType="real32_Type" start="0.0"/>
  </ScalarVariable>
  <ScalarVariable name="measuredTemp" valueReference="1" variability="continuous" causality="input">
    <Real declaredType="real32_Type" start="0.0"/>
  </ScalarVariable>
  <ScalarVariable name="heater1" valueReference="0" variability="discrete" causality="output">
    <Boolean start="false"/>
  </ScalarVariable>
  <ScalarVariable name="heater2" valueReference="1" variability="discrete" causality="output">
    <Boolean start="false"/>
  </ScalarVariable>
  <ScalarVariable name="switchingTimeout" valueReference="0" variability="parameter"
       causality="internal">
    <Integer start="1000"/>
  </ScalarVariable>
  <ScalarVariable name="tempDiff1" valueReference="2" variability="parameter" causality="internal">
    <Real declaredType="real32_Type" start="5.0"/>
  </ScalarVariable>
  <ScalarVariable name="tempDiff2" valueReference="3" variability="parameter" causality="internal">
    <Real declaredType="real32_Type" start="10.0"/>
  </ScalarVariable>
</ModelVariables>
```

Figure 4: The `modelDescription.xml` file of the generated FMU (excerpt).

a start value of the corresponding FMU variable. However, the «FMUIgnore» stereotype will prevent any element from being exposed. This gives the user fine control over what parts of the block are to be exposed externally.

Not all SysML elements are currently supported. In particular, the following are challenging, because of mismatches between SysML and the FMI standard:

- Bi-directional flow ports, since the FMI standard does not allow variables that are both inputs and outputs. This restriction may be lifted in future, by artificially creating two variables, one for each direction of a bi-directional port. However, there are various issues such as naming of ports and variables, and synchronizing the values of the input and output variables correctly.

- SysML standard ports, since they carry SysML events (these are not to be confused with FMI events), which are not supported by the FMI standard.

The FMU export functionality is based on the FMU SDK provided by QTronic,[9] which provides a skeleton implementation of the FMI API. The implementation of the exported FMU consists of the code normally generated by Rhapsody for the block, with an additional wrapper that adapts it to the requirements of the FMI standard. The FMU export process consists of the main steps described in Algorithm 1.

The first step collects the various elements to be exposed according to the rules described above; it also checks for illegal or unsupported combinations and reports them to the user (such reports can be suppressed by using the «FMUIgnore» stereotype). The second step creates the `modelDescription.xml` file, which describes the interface of the generated FMU.

[9]`http://www.qtronic.de/en/fmusdk.html`.

Algorithm 1 Generate FMU Wrapper

```
1. Analyze the SysML model to identify input and
   output ports and internal varibles, discover and
   report errors.
2. Generate the FMU model description file.
3. Apply the standard Rhapsody code generation.
4. Generate the code for the FMU wrapper.
5. Compile and package the FMU.
```

Algorithm 2 fmiEventUpdate

```
1. Set Rhapsody time based on the time received in
   the last fmiSetTime call.
2. Set the block's input variables based on the pre-
   vious set of fmiSetXXX calls.
3. Invoke the Rhapsody-generated code to execute a
   behavioral step.
4. Update the wrapper's variables corresponding to
   the FMU outputs.
5. Set the next event time to the earliest timeout ac-
   tive in the block.
```

The third step invokes Rhapsody's code-generation facilities to create the implementation of the behavior of the block to be exported. The fourth step creates the code that adapts Rhapsody's implementation to the FMI standard. This wrapper code is discussed in most of the rest of this section. Finally, the code is compiled and packaged in the `.fmu` file.

3.1 The FMU Wrapper

The FMU Wrapper generated in step 4 of the FMU export algorithm supports the FMI interface (currently version 1.0 for Model Exchange) and translates it to the Rhapsody interface. The wrapper keeps a set of variables corresponding to the FMU variables, together with a set of other internal variables (for example, the current simulation time). Most of the FMI functions are implemented by reading or setting these variables; the real work is done by the `fmiEventUpdate` function. The wrapper algorithm implements this function as described by Algorithm 2.

This algorithm seems quite straightforward, but it hides many subtle semantic issues. When a block is exported as an FMU, its behavior can be changed by the way that the hosting tool delivers variable changes to the wrapper, using the FMI interface calls, and by the way that the wrapper handles these calls. While the former is out of the control of the Rhapsody FMU export plugin, the latter behavior is, and there are a number of different ways to create the wrapper, each of which yields somewhat different semantics. There are two issues that need to be addressed. The first is communication: the way inputs are transferred to the FMU and outputs are received from it. The second is scheduling: when communication takes place, and how much activity the FMU encapsulating the SysML block allows the block to perform before it considers the step to be completed.

Ideally, Rhapsody blocks exported as FMUs would retain their behavioral semantics, and blocks written without consideration for their context could be used as FMUs. At the very least, the naive expectation could be that if two or more Rhapsody blocks are exported as FMUs out of the same model, and are connected in the hosting environment in exactly the same way they were connected in the Rhapsody model, their behavior will not change. However, context does matter, in simulation as well as in physical realizations. Full preservation of semantics between two exported blocks is challenging; see Section 3.2. In the context of non-Rhapsody models, the different semantics of the FMI standard and of Rhapsody create other difficulties, as discussed in Sections 3.3–3.5.

3.2 Quoted Out of Context

In this section we consider the case of two blocks from a single Rhapsody model, both of which are exported as FMUs, and connected in the external tool in the same way as they were in the original model. How does the implementation of the wrapper change the semantics of the joint model?

The Rhapsody semantics [7] describes the behavior of a statechart as consisting of a series of steps; each step may consist of several state changes, and may produce several variable-change and other events. There is a strict order between these events.

Changing the value of a SysML flow port in Rhapsody might also create an internal change event (such as `chTurnOn` in Figure 2); this event can trigger one or more transitions. The value of the attribute corresponding to the port is changed immediately. However, all events in Rhapsody are sent asynchronously; events sent to each port (from whatever source) are queued, and delivered in order. In particular, updating several variables consecutively creates one event for each, and these are handled one by one. The order is therefore significant; the block may perform arbitrary actions in response to each event; for example, a typical response to an event is for the statechart is to move to a new state. Such actions can change how the block responds to the next event.

As mentioned above, the FMI standard allows changing any number of variables in a single event-processing cycle, and the changes are semantically simultaneous. This is a consequence of the synchronous semantics on which the FMI standard is based. A Rhapsody block can perform a sequence of discrete changes in response to a single event. For example, a single transition might include a series of actions that sequentially change a number of outputs. This implies that, in order to ensure that the order of the corresponding Rhapsody events is preserved, these discrete changes must be delivered one by one to the FMU interface. However, this could cause an inconsistecy between the values, if several related variables (such as current and resistance) are not changed simultaneously.

The wrapper can implement this strategy by ending step 3 of Algorithm 2 inside each of the setter functions for the output variables in the Rhapsody-generated code, and continuing with steps 4–5. In the next event-processing cycle, the wrapper would allow Rhapsody to continue from the point it was stopped. All of these events, except for the last one, will require another event-iteration cycle by returning `fmiFalse` in the `iteractionConverged` field of the return value of the `fmiEventUpdate` function; this does not advance the simulation time.

This strategy allows other FMUs to generate new inputs after each output is reported. These inputs can be delivered by the wrapper to Rhapsody immediately; that would be consistent with the Rhapsody semantics, since it represents an execution of the Rhapsody model in which each block is run in a separate thread and output generation happens to be fully synchronized between threads. However, Rhapsody semantics allows other behaviors; for example, those where one thread is significantly faster than the others.

In this strategy, the exported FMU behaves in accordance with the original semantics of the SysML blocks in Rhapsody. Any strategy that batches consecutive outputs may violate the semantics, since the generated FMU can be used in a context in which the order of these output updates is significant. However, as mentioned above, the Rhapsody semantics may expose inconsistent value. Furthermore, this strategy could potentially be very inefficient, forcing the whole simulation to receive and react to each discrete change separately. Because of these issues, the current implementation of the FMU wrapper delivers all variable-change events simultaneously; the modeler of the SysML block should be aware of this and introduce small delays between sequential changes when the order is important. This strategy takes the other extreme, in that it allows the block to perform all its possible behavior in a single cycle, and only stops when it reaches a wait for some timeout or event.

A related issue is the treatment of multiple changes to the same variable. In the example, the command to turn the first heater on is given by the signal `heater1`; the command is recognized when the signal changes value from `false` to `true`. It is possible that some path in the statechart contains a series of transitions in the same step, containing the action `setVar(false)` followed by `setVar(true)`. In Rhapsody, both actions will be communicated to the connected element, resulting in two changes in the value of the signal, with two corresponding change events, causing the command to be recognized. However, if this signal is exported as an FMU output, the behavior will depend on how the wrapper treats multiple changes to the same variable. In the implemented version of the wrapper, the second change will override the first, causing the signal to retain its previous value without change. The other block will therefore receive only the last value update, which contains the variable's original value; this might result in different behavior.

In the other strategy, both changes will be delivered separately, and the command will be recognized. A third scheduling strategy, in the middle between these two extremes, is to present changes to different variables together, as in the implemented version, but separate changes to the same variable. This can be done by having the wrapper divide the stream of events received from Rhapsody into segments, each starting with a change to a variable that already has another change in a previous segment. All the events in a single segment will be presented at once, but different segments will be presented separately.

On input (step 2 of Algorithm 2), the wrapper must be ready to accept simultaneous changes of discrete variables, even if it only produces them one at a time, since inputs may be provided by other types of FMUs. There are a number of ways to treat multiple such events. The first is for the wrapper to issue the events

Table 1: Two behaviors of the rounding strategy.

FMI Time *(sec.)*	Rhapsody Time *(msec.)*	x_1	x_2	x_3	y
Scenario 1					
0.0000	0	1	2	3	2
0.0050	5	-1	2	3	2
0.0100	10	1	2	3	2
Scenario 2					
0.0000	0	1	2	3	2
0.0050	5	-1	2	3	2
0.0096	10	-1	2	-3	-2
0.0100	10	1	2	3	-2

to Rhapsody in an arbitrary order, and require modelers of SysML blocks to be exported to ensure that the order does not matter; this is the option currently taken by the plugin. A second option is to impose a specific order on the events; this order can be specified by a SysML stereotype.

3.3 A Question of Time

In the current FMI standard, time is measured as a double-precision floating-point value in units of seconds, whereas in Rhapsody time is represented as an integer, in units of milliseconds. This mismatch is another cause for semantic incompatibilities. (The same problem would exist even if the standard used integral units at a different time resolution, such as microseconds.) How should the FMU wrapper handle `fmiSetTime` calls that set the time between two Rhapsody clock ticks?

Of course, a perfect match of the time-lines is impossible. An obvious candidate strategy is to round times to the nearest integral value. However, this strategy causes strange phenomena, where the addition of an unrelated `fmiEventUpdate` call can completely change the behavior. For example, consider a SysML block with three discrete inputs, x_1, x_2, and x_3, and one discrete output, y. The behavior of the block is very simple; periodically every 10 milliseconds, it updates y to be either x_2 or $-x_2$, depending on whether $x_1 > 0$ or not. Input x_3 is not used at all and should not influence this behavior in any way.

Table 1 displays two possible behaviors of this block, under the rounding strategy. In the first scenario, y is updated at time 0.0, and again at time 0.01. The change of x_1 at time 0.005 does not update the value of y, since the block is still waiting for its timeout. The second scenario starts in the same way, except that at FMI time 0.0096, variable x_3 gets a new value. Because this time is rounded to Rhapsody time 10, the block is activated and updates the value of y. When x_1 is changed back to 1 at FMI time 0.01, the block is already waiting for its next timeout, at (Rhapsody) time 20, and y is not updated again. The result is a different value for y, due to a spurious update of x_3. For continuous systems, due to numerical issues, such differences might be acceptable. For discrete system, working at precise clocks, however, this type of behavior is obviously unacceptable.

Because of this issue, the FMU export plugin uses truncation rather than rounding. With truncation, such undesirable behaviors cannot occur, since intermediate event updates such as the one in the example will not trigger the FMU's timeout and will not advance its internal clock. In the example, FMI time 0.0096 will be translated to Rhapsody time 9, will not trigger the timeout transition, and will not change the value of y.

It is common in continuous systems to consider a discrete unit of time strictly as a sampling interval, by ignoring all changes that occur between two sampling points except for the last one. However, this approach can result in counterintuitive behavior for discrete models; for example, a slight shift in the timing of a discrete signal change can cause it to be ignored.

3.4 Types

A Rhapsody model can employ the full type system of the target language (which, in this paper, we consider to be C). The FMI standard defines a different set of types; these only contain scalar types (real, integer, boolean, string, and enumeration). Each of these can be customized; for example, real and integer types can have an associated range, as shown in the type definition of Figure 4. The FMU export plugin attempts to define the closest possible FMI type for the C type used in the block. Obviously, integral C types are expressed as FMI integers, and floating-point types (`float`, `double`) as FMI reals. Ranges are applied in the FMI types based on the ranges of the C types; however, not all types can be accurately represented in this way. For example, in the 32-bit FMI platform, `fmiInteger` is defined as a C `int`; this means that the C type `unsigned int` has a wider range than that admitted by an FMI integer, which is always signed.

3.5 Discrete or Continuous?

Rhapsody is based on a discrete-time model; triggers for transitions between states are all discrete events, and variables are modified in a discrete way (that is, signals are piecewise constant). Modelers in Rhapsody therefore think of all variables as discrete. However, when a Rhapsody FMU is used in a hybrid model, some of its inputs may be connected to continuous signals. This places restrictions on the way such inputs may be used in the statechart.

Transitions in a statechart are activated by *triggers*, which are discrete events. Triggers include variable-update events (such as `chTurnOn` in Figure 2, which signals a change in the `turnOn` input). The time of a change event for a continuous variable is not well defined, because at a call of `fmiEventUpdate` all FMI variables are updated, whether relevant or not (as in the second scenario of Table 1). This can cause the behavior of the statechart to become unpredictable. Modelers must therefore avoid change events for continuous inputs. The use of continuous inputs in other places is not restricted. For example, checking the value of an input in a transition guard (such as `tempDiff()<0`) does not cause a problem, even if the input is continuous, since the guard is only evaluated at timeout events. The same holds for using a continuous input in an action.

3.6 Guidelines for Exportable Blocks

Based on the previous discussion, the following points should be considered when designing a Rhapsody model to be exported as an FMU. First, change events must not be used for continuous signals, although their values can be freely used otherwise. Second, if the order of changes in output variables is important, non-zero delay should be introduced between changes. Third, non-zero delay should be introduced between changes of the same variable if all intermediate values need to be observed. Finally, to preserve communication semantics between Rhapsody blocks or include features that are not yet supported by the FMU export functionality, the composition of the blocks should be exported as a single FMU.

4 Related work

Modeling and simulation of hybrid systems is an active research topic [5, 3]. Carloni et al. [4] provide a detailed analysis and comparison of the semantics of commonly-used tools. They conclude that there is a strong need to allow integration of different tools, and suggest leveraging the Hybrid Systems Interchange Format (HSIF) to mediate model semantics between tools. This approach is very different from the FMI code-generation-based approach used in our work, since FMI is mainly focused on the standardization of a model execution API, and exposes only the model information required for this purpose (for example, whether variables are discrete or continuous, and dependencies between variables).

Other interesting approaches for integrating SysML and Modelica are based on extensions of SysML such as the SysML-Modelica Transformation standard,[10] or of ModelicaML [10] where Modelica could be described using the UML profile extension mechanism. Specifically, Schamai et al. [11] suggest a formal approach to modeling UML statecharts using Modelica, and highlight various semantic differences. For example, in ModelicaML, all available events are processed in parallel in the next evaluation of the state machine, while in Rhapsody statechart events are queued and processed in order according to the UML run-to-completion semantics [7].

In practice, changing languages or tools is a major undertaking, and users are reluctant to do so. Our work therefore concentrates on using the popular Rhapsody tool in new contexts. As discussed in Section 3, this requires some attention to modeling details that might differ in other contexts, but there is no need to change the tool itself. This is similar to the approach taken by Sakairi et al. [9] to integrate Rhapsody and Simulink®, except that they use a proprietary S-function interface.[11] Because we use the FMI standard, our approach is not limited to integration with a single tool or language, but can work with any FMI-complaint simulator.

Pohlmann et al. [8] export FMUs for MechatronicUML [1] instead of Rhapsody SysML. They do not describe any semantic differences between MechatronicUML and the generated FMU. They do say that "a discrete port implements an array of message queues," since communication in MechatronicUML is asynchronous; it seems, therefore, that the same issues described in Section 3.2 are relevant there as well.

5 Conclusions

The FMI standard can be used for hybrid simulation of systems modeled using several tools. We described the Rhapsody plugin that exports FMUs enapsulating SysML blocks whose behavior is defined using statecharts, and demonstrated it on an example that

[10] `http://www.omg.org/spec/SyM/1.0`.

[11] `http://www.mathworks.com/help/simulink/sfg/what-is-an-s-function.html`.

combines the SysML model with a Modelica model. We highlighted subtle but important semantic differences between Rhapsody and FMI, causing the compositional behavior of the FMUs to differ from that of the original blocks, and provided guidelines that will prevent such behavioral differences. Since different tools come with their own semantics, we expect that such mismatches are common, especially when connecting continuous-time with discrete models, and that our guidelines will generalize to many such cases.

Acknowledgments

We are grateful to Sergey Zolotnitsky for his help with the implementation of the plugin. We thank ITI GmbH for their assistance with SimulationX.

References

[1] S. Becker, C. Brenner, C. Brink, S. Dziwok, C. Heinzemann, U. Pohlmann, W. Schäfer, J. Suck, O. Sudmann, and R. Löffler. The MechatronicUML design method – process, syntax, and semantics. Technical Report tr-ri-12-326, Heinz Nixdorf Institute, University of Paderborn, 2012.

[2] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter. Determinate composition of FMUs for co-simulation. In *Proc. Int'l Conf. Embedded Software (EMSOFT)*, pages 1–12, 2013.

[3] D. Broman, E. A. Lee, S. Tripakis, and M. Törngren. Viewpoints, formalisms, languages, and tools for cyber-physical systems. In *Proc. 6th Int'l Workshop on Multi-Paradigm Modeling*, 2012.

[4] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, 1(1–2), 2006.

[5] P. Derler, E. A. Lee, and A. L. Sangiovanni-Vincentelli. Modeling cyber-physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Computer Programming*, pages 231–274, 1987.

[7] D. Harel and H. Kugler. The Rhapsody semantics of statecharts (or, on the executable core of the UML). In *Integration of Software Specification Techniques for Applications in Engineering*, pages 325–354. Springer, 2004.

[8] U. Pohlmann, W. Schäfer, H. Reddehase, J. Röckemann, and R. Wagner. Generating functional mockup units from software specifications. In *Proc. 9th Int'l Modelica Conf.*, pages 765–774, 2012.

[9] T. Sakairi, E. Palachi, C. Cohen, Y. Hatsutori, J. Shimizu, and H. Miyashita. Designing a control system using SysML and Simulink. In *Proc. SICE Annual Conf.*, pages 2011–2017, 2012.

[10] W. Schamai. Modelica modeling language (ModelicaML): A UML profile for Modelica. Technical report, Linköping University, 2009.

[11] W. Schamai, U. Pohlmann, P. Fritzson, C. J. J. Paredis, P. Helle, and C. Strobel. Execution of UML state machines using Modelica. In *Third Int'l Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT)*, pages 1–10, 2010.