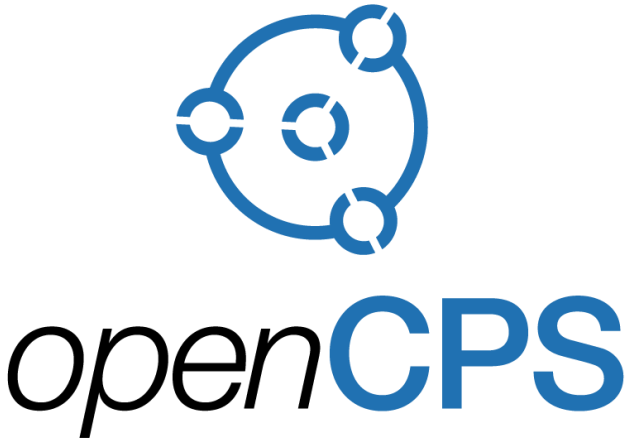
D2.2	**Interoperability of the standards Modelica-UML-FMI**
Access[1]:	**PU**
Type[2]:	**Report**
Version:	**0.3**
Due Dates[3]:	**M12, M24**



Open Cyber-Physical System Model-Driven Certified Development

Executive summary[4]:

M12 version of D2.2 enabled the specification of a precise semantics for UML state machines and to identify semantics differences between UML and xtUML state machines. The purpose of such work was twofold: (1) enable state machines to be used in the specification of simulation models described in UML (2) Evaluate how Executable UML semantics could be specialized to account for semantics associated to xtUML. This deliverable established the basics to make sure that both UML and xtUML could be used in a simulation process.

M24 version of D2.2 provides an exploratory analysis of different strategies to integrate simulation models specified using UML and xtUML in a co-simulation process. In particular, the deliverable specifies two families of approach enabling these kind of simulation models to be exported in an FMU (Functional Mockup Unit). The first kind of approach advocates that such integration could be made using a profile and an extension of the semantics of the language used to specify the simulation model. The second kind of approach advocates for a wrapping based strategy.

[1] Access classification as per definitions in PCA; PU = Public, CO = Confidential. Access classification per deliverable stated in FPP.

[2] Deliverable type according to FPP, note that all non-report deliverables must be accompanied by a deliverable report.

[3] Due month(s) according to FPP.

[4] It is mandatory to provide an executive summary for each deliverable.

Deliverable Contributors:

	Name	Organization	Primary role in project	Main Author(s)[5]
Deliverable Leader[6]	Jérémie TATIBOUET	CEA	Task Leader	X
Contributing Author(s)[7]	Ákos HORVATH	IncQuery Labs	Contributor	X
	Zoltan GERA	ELTE-Soft	Contributor	X
	Boldizsár NEMETH	ELTE-Soft	Contributor	X
	Dániel SEGESDI	IncQuery Labs	Contributor	X
	Sébastien REVOL	CEA	Contributor	X
	Gergely Dévai	ELTE-Soft	Contributor	X
Internal Reviewer(s)[8]	Ákos HORVATH	IncQuery Labs	Contributor	X
	Sébastien REVOL	CEA	Contributor	X

Document History:

Version	Date	Reason for Change	Status[9]
0.1	10/11/2017	Initial version of the deliverable	Draft
0.2	17/11/2017	Contribution and review from Akos	Draft
0.3	17/11/2017	Minor editorial corrections	Released

[5] Indicate Main Author(s) with an "X" in this column.
[6] Deliverable leader according to FPP, role definition in PCA.
[7] Person(s) from contributing partners for the deliverable, expected contributing partners stated in FPP.
[8] Typically, person(s) with appropriate expertise to assess deliverable structure and quality.
[9] Status = "Draft", "In Review", "Released".

CONTENTS

ABBREVIATIONS....3
1 INTRODUCTION....4
1.1 FMI and UML / xtUML....4
1.2 Problem statement....4
2 TEST CASE....5
3 FMI INTEGRATION WITH UML AND XTUML....6
3.1 FMU Structure....6
3.2 Specify FMI Structure with UML / xtUML....6
3.2.1 UML....7
3.2.2 xtUML....9
3.3 Formalize Interactions between an FMU and the Master....9
3.3.1 Approach N°1: Extend UML Semantics....9
3.3.2 Approach N°2: Wrap Models into a Higher Level Component....14
3.4 Comparisons....24
3.5 Issues to be resolved....27
3.5.1 Roll-Back Support....27
3.5.2 Inputs Consumption Support....28
3.5.3 Outputs Production Support....29
3.5.4 Time Support....31
4 CONCLUSIONS....33
TABLE OF FIGURES....35
REFERENCES....36

ABBREVIATIONS

List of abbreviations/acronyms used in document:

Abbreviation	Definition
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
UML	Unified Modelling Language
FUML	Semantics of a Foundational Subset for Executable UML Models
PSCS	Precise Semantics of UML Composite Structures
PSSM	Precise Semantics of UML State Machines
RTC	Run to Completion
SW	Software

1 INTRODUCTION

1.1 FMI and UML / xtUML

FMI [1] is a standard defining how to couple two or more simulation models in a co-simulation environment. The simulation models may be specified with different languages whose support is provided in different tools. However, the FMI standard was designed to support the simulation of continuous time modeling languages, thus typical SW modeling languages based on discrete event-based formalisms cannot be easily mapped to the constructs defined in FMU.

Within the context of OpenCPS, the three different platforms used to design complex cyber-physical systems for various engineering domains are: Open Modelica, Bridge Point and Papyrus. All of these platforms are based on different languages, where Open Modelica relies on the Modelica language a standard for continuous time modeling, Bridge Point relies on xtUML and Papyrus on UML as well as all UML-based languages (i.e., languages formalized as profiles), the latter two used for SW modeling with event based formalism.

UML [2] and xtUML [3] are heavily used in industry to design control systems. In the classical design process, these systems can be formalized in a sufficient level of detail in order to make them executable or to enable the generation of an executable code. Since the models are executable it is possible to export them into FMU in order to use them in co-simulation. The interest here is to evaluate the different mapping opportunities of UML/xtUML models into FMU in order to further the quality of multi-domain simulations of cyber-physical systems.

1.2 Problem statement

The export of simulation models specified in UML or xtUML into FMU used in a co-simulation immediately raise two issues:

1. How can systems formalized with these languages be exported into FMU? How are the exported software components identified? What are the variables of these components being exposed as FMU variables? How are these variables identified and formalized into the model?

2. While the communications between the master and FMUs involved in a co-simulation process are formalized in the FMI [1] specification, the translation of these communications into a form that can be understood by the underlying simulation model formalized either with UML or xtUML remains unspecified. As an example, lets consider the assignment of an FMU variable. From the master algorithm point of view, the set action merely corresponds to a call to the `fmiSetXXX(…)` API function. However, on the other side, depending on the way the variable is formalized into the original simulation model, it may be necessary to convert that call to a signal sending or an operation call through a port. This kind of event base communication depending on the form of the simulation model may imply the triggering of computations while a `doStep` request has not already been received from the master.

The first objective of this deliverable is to clarify how systems specified with UML and xtUML shall be designed (i.e., identification of the main component, identification of the exposed variables) in order to make possible their export into an FMU. The second objective is to

formalize the translation of the master algorithm interactions with an FMU containing a simulation model specified with UML or xtUML into communications that can be understood by this latter. The hard constraint here is to ensure that the translated communications will preserve the semantics initially intended by the calls performed by the master.
The outline of the document is organized as follows: section 2 presents the test case used to evaluate the different approaches to integrate UML and xtUML simulation models in FMUs. Section 3 identify the modeling constraints applying on each approach and precisely how the interactions with the master will be performed.

2 TEST CASE

Figure 1 shows a composite assembly of three FMUs. The `Control Interface` FMU corresponds to the part of the system enabling the user to power on or off the control of the temperature level in a room. The `Thermostat` FMU is in charge of deciding the strategy to apply in order to maintain the temperature in the room. To do so, it takes as inputs the status of the control interface (either ON or OFF) as well as the current temperature in the room. Based on its decision, the `Thermostat` FMU forwards the order to turn the heating system ON or OFF to the `Room` FMU.

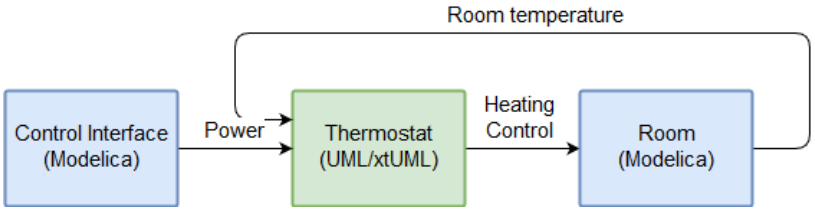


Figure 1 - Test Case Environment

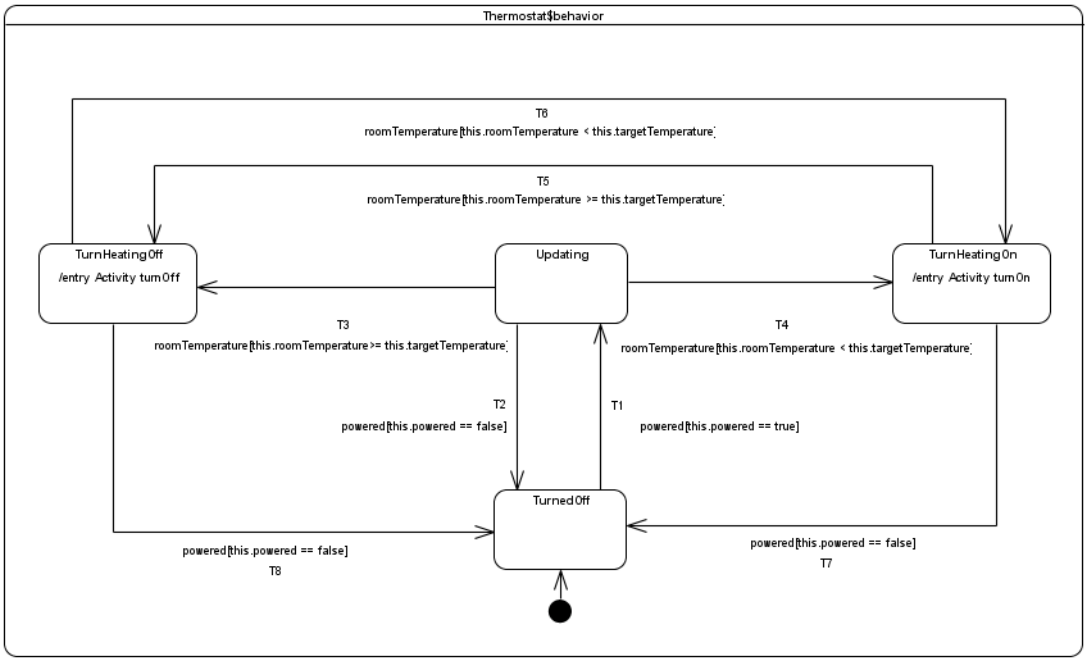


Figure 2 - Behavior specification of Thermostat FMU

While the `Control Interface` FMU and the `Room` FMU are specified using Open Modelica, the `Thermostat` FMU is specified using UML or xtUML. The behavior of the `Thermostat` FMU is defined as a state machine. An example of behavioral specification for is shown in Figure 2.

The state machine is split into four states and eight transitions. Sate `TurnedOff` is one entered by the `Thermostat` when the system starts. It indicates the situation where no temperature control is performed. The state `Updating` indicates that the `Thermostat` is on and shall control the room temperature. To do so it requires to receive at least once the current temperature room. At this point, it has the opportunity to move either to states `TurnHeatingOff` or `TurnHeatingOn`.

One can notice that transitions in that test case are triggered. The triggering of these latter is can only be done upon the acceptance of signal event occurrences (i.e., `powered` and `roomTemperature`).

3 FMI INTEGRATION WITH UML AND XTUML

3.1 FMU Structure

An FMU is described as a black box component exposing variables (see section 4.1.1 in [1]). The component is mainly used to materialize the software artefact manipulated by the master. Variables identify the interaction points at which the master can post and retrieve data from the FMU.

Both the component and its variables expose FMI specific information. As an example, a variable is not only defined by a `type` constraining data that can be hold by this latter but also with a `variability` and `causality`. The `variability` identifies the time dependency of the variable to time (e.g., discrete or continuous). Finally, the `causality` enables to specify if the variable is an `input` (i.e., the value hold by this variable can be provided from another model or slave), an `output` (i.e., the value hold by this variable can be provided to another model or slave) a `parameter`, a `calculatedParameter`, `local`, or `independent`.

While both UML and xtUML already provide reusable concepts to model a FMU, they do not provide the capability to capture information that are specific to the FMI standard (see section 2.2 in [1]).

3.2 Specify FMI Structure with UML / xtUML

Both UML and xtUML provide concepts that can be reused to specify both the component to be exposed through the FMU as well as the variables attached to that component. Sections 3.2.1 and 3.2.2 identify in both languages the concepts to be reused. These sections also identity how to capture FMI specific information.

3.2.1 UML

3.2.1.1 Component

UML provides the concept of `Class` (see section 11.8.3 in [2]). This concept is heavily used in system design to specify classification of object types exposing features and operations. Such object type may also be attached to behaviors intended to define the computations that have to be performed by any instance of the object type.

The `Class` concept looks appropriate to model the component corresponding to an FMU for three reasons:

1. It has the capability to own structural features (see section 11.8.3.6 in [2]) which can be used to model components variables.
2. It has the capability to own operations (see section 11.8.3.6 in [2]) which can be called to request computations to be performed on the component.
3. It has the capability to own a classifier behavior (see section 10.5.1.5 in [2]) which can be used to specify how to handle events received by the component.

3.2.1.2 Variables

UML provides concepts of `Property` (see section 9.9.17 in [2]) and `Port` (see section 11.8.14 in [2]). Both can be used to specify the variables of an FMU component. Indeed, they are `TypedElement` (see section 7.8.22 in [2]) hence they can be associated to a type that will constrain the type of information that can flow through them. In [1] section 2.2.7 in states that the type used to type variables can only primitives or enumerations.

3.2.1.3 FMI specific information

The FMI domain specific information put on a FMU component and its variables (e.g., `causality` meta-property, FMU `canInterpolateInputs` meta-property – see section 4.3.2 in [1]) cannot be natively captured by `Class`, `Property` and `Port` concepts. This issue suggests to use a UML profile for FMI that will define the necessary extensions to the UML concepts in order to capture FMI domain specific information.

An excerpt of such profile is given in Figure 3. Stereotypes `FMU`, and `ScalarVariable` formalize information that need to be captured specifically for FMI variables and the component holding these latter.

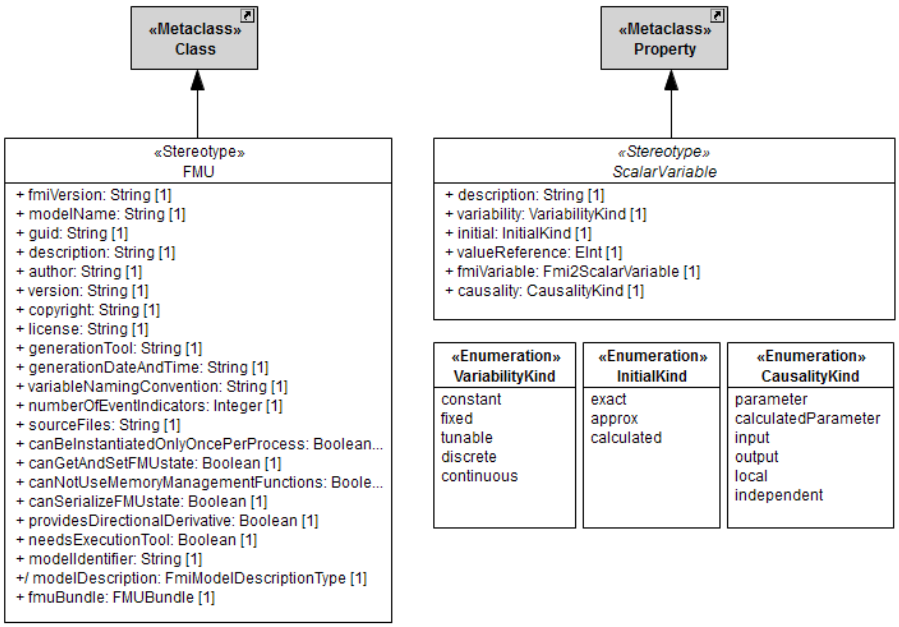


Figure 3 - Excerpt of a UML profile for FMI

Using a regular UML model with the FMI profile applied, an FMU can be described as shown in Figure 4. The class `Thermostat` is an FMU since it has stereotype `FMU` (see Figure 3) applied. FMU Variables are the ports displayed on right and left sides of the FMU. Each port is typed by a UML primitive type and has the stereotype `ScalarVariable` applied. This stereotype enables each port playing the role of an FMU variable to expose information about the way the variable is initialized (see `initial` property in Figure 3), its `causality` as well as its `variability`.

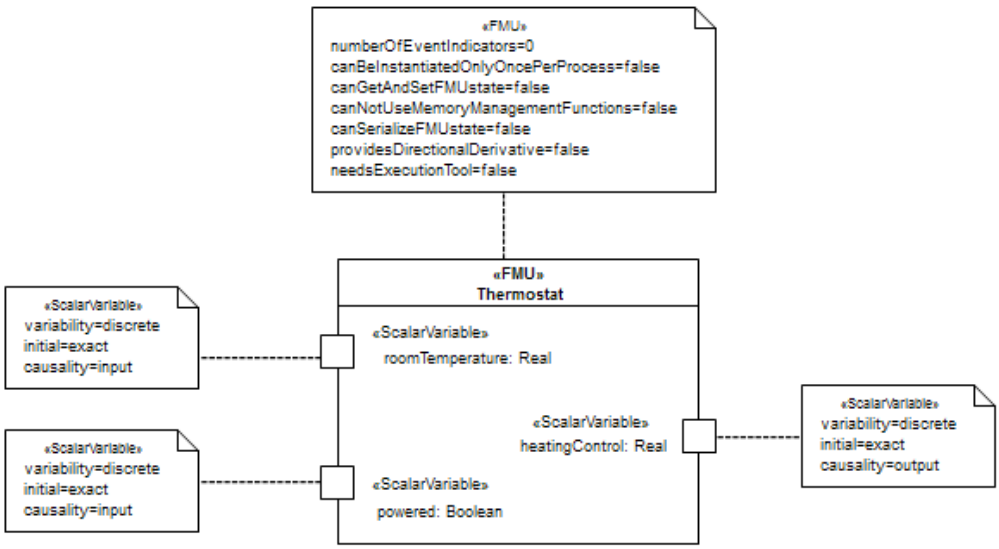


Figure 4 - Thermostat FMU defined with UML and the FMI profile

3.2.2 xtUML

3.2.2.1 Component

xtUML provides the concept of `Component`. `Components` formulate the basis of xtUML based system design and they are used to specify and encapsulate classes defining certain aspects and behavior of the system.

The direct mapping of xtUML `Components` to FMU Components is a straightforward approach as both are used for the same purpose in their respective languages: to hide internal details and execution semantics from the outside world, and to provide a well-defined communication means to the outside world.

3.2.2.2 Variables

As UML, xtUML also provides the concepts of `Property` and `Port`. Both can be used as an information flow between the FMU variables and the xtUML model. Further details on the different mapping strategies of FMU variables to xtUML `Properties` and `Ports` are described in Section 3.3.2.

3.2.2.3 FMI Specific Information

In case of the xtUML to FMU mapping, we opted to follow a different approach compared to the profile based extension used in case of UML. The main reason for our option was to provide a Bridgepoint agnostic solution that can work with any xtUML modeling environment out of the box. For this reason, all FMU specific domain information are encoded into separate Eclipse Modeling Framework models that drive the FMU wrapper generation process. The defined EMF models and the realization of the FMU wrapper generator are described in Annex 1 [4].

3.3 Formalize Interactions between an FMU and the Master

This section describes two approaches to provide the capability to a simulation model defined either in UML or xtUML to be exported in a FMU and responds to the master interactions.

3.3.1 Approach N°1: Extend UML Semantics

The approach presented in this section proposes to extend the UML base semantics in order to enable the usage of simulation model specified in UML in the FMI context. Section 3.3.1.1 identifies the modeling constraint that applies to a simulation model specified using this approach. Section 3.3.1.2 presents the proposed extensions to the UML base semantics and rationalize these extensions to ensure the capability of the simulation to be executed and to account for requests received from the master. Section 3.3.1.3 describes how the interactions (set, doStep and get) between the master and the simulation work in the context of the extended UML semantics.

3.3.1.1 Modeling Constraints

In this approach, the component to be exported in an FMU is always expected to be specified as a nonhierarchical active class (i.e., a class with a classifier behavior and the meta-property

`isActive` set to true). This class can have one to many ports which play the role of FMU variables. Each port is required to be typed by a primitive type or an enumeration.

The classifier behavior attached to the class modeling the FMU component is required to be either an activity or a state machine. In addition, triggers used on transitions or accept event actions can only be for `ChangeEvent` (see section 13.4.4 in [2]) and `TimeEvent` (see section 13.4.10 in [2]). This constraint implies that the classifier behavior is only allowed to accept event occurrence related to the change on variable value and the elapsing of time.

Finally, the model containing the specification of this simulation model shall always have the FMI profile applied. The FMU component defined through the active class will have the stereotype `FMU` (see Figure 3 in section 3.2.1.3) applied. Each port will have the stereotype `ScalarVariable` (see Figure 3 in section 3.2.1.3) applied. If defined according to these constraints the model will have a form similar to the one presented in Figure 4.

3.3.1.2 Semantics Extensions

UML semantics is precise and formal for the subset of the syntax including classes, composite structures, activities and state machines. This implies that any model built using this subset can by construction be executed.

Unfortunately, an FMU conforming to the modeling constraints specified in section 3.3.1.1 cannot be also conformant with the executable UML subset. Indeed, to be executed the FMU will require that semantics for `ChangeEvent` (see section 13.4.4 in [2]) and `TimeEvent` (see section 13.4.10 in [2]) is defined.

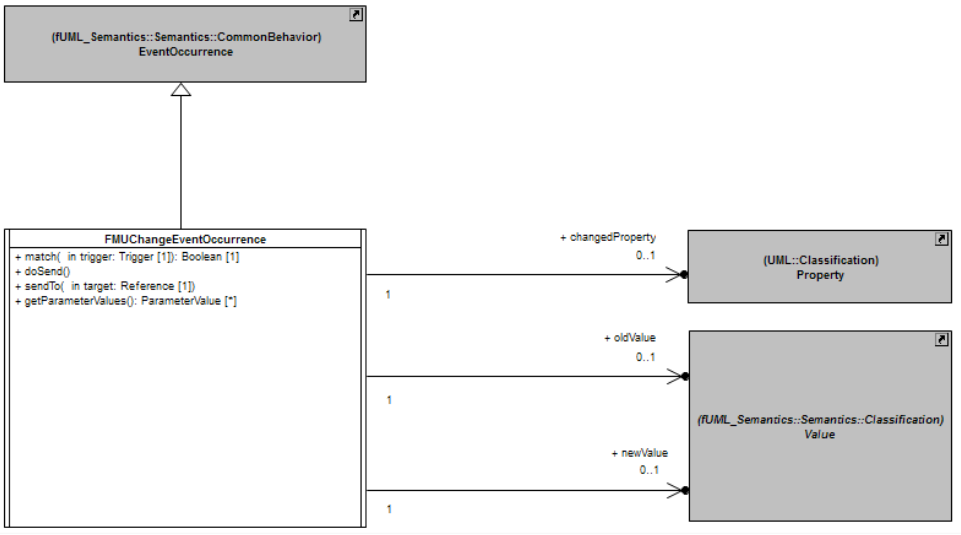


Figure 5 - Semantics for ChangeEvent

Figure 5 shows the extension introduced to [5] in order to provide a support for change events. Semantics for this event type is formalized as a specialization of class `EventOccurrence` (see section 8.4.3.2.6 in [5]). The `FMUChangeEventOccurrence` class overrides the semantics of the `match` operation. A change event occurrence matches a trigger for a change event when the changed property is the same property than the one specified in the change

expression. Such event is generated only when the value of an observed property changes and the new value is different from the new value.

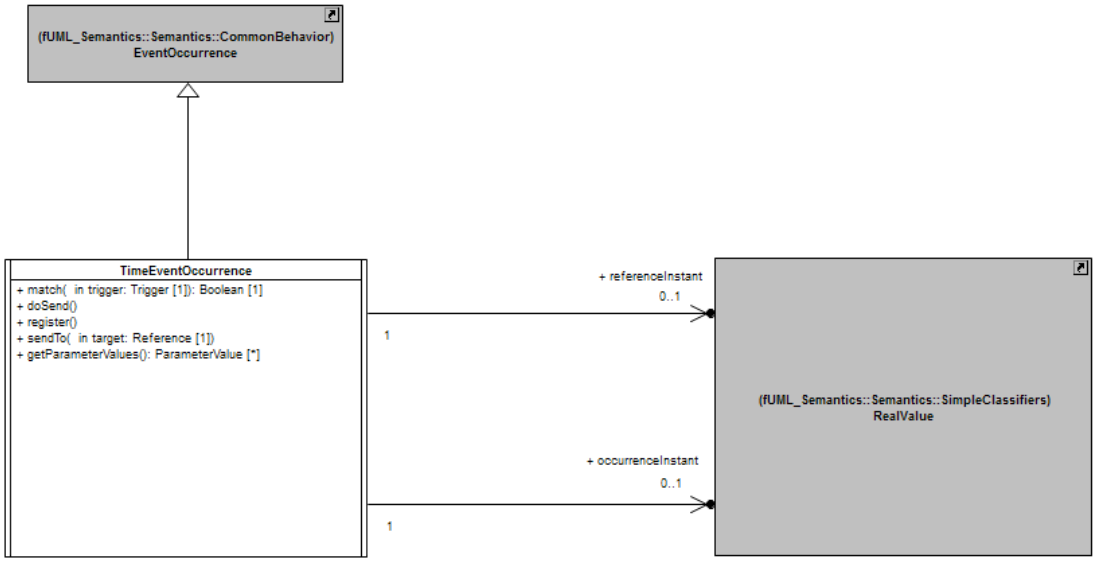


Figure 6 - Semantics for TimeEvent

Figure 6 shows the extension introduced to [5] in order to provide a support for time events. Semantics for this event type is formalized as a specialization of class `EventOccurrence` (see section 8.4.3.2.6 in [5]). The `TimeEventOccurence` class overrides the `match` operation. Such event occurrences only match a trigger for a time event when the time at which the time event occurrence was produced matches the time specification evaluated from the time expression attached to a trigger.

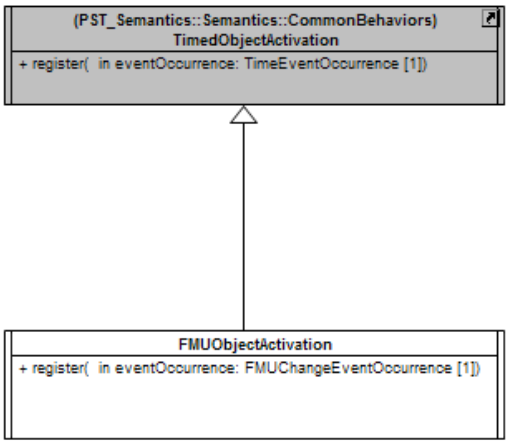


Figure 7 - Semantics for FMUObjectActivation

The introduction of a support `ChangeEvent` (see section 13.4.4 in [2]) and `TimeEvent` (see section 13.4.10 in [2]) requires the semantics defined for an object activation to also be specialized. Such extension is shown in Figure 7. A new type of object activation that is specific to FMU object is defined, this object activation enables the capability for a change event occurrence to be registered at the event pool. One can notice that an `FMUObjectActivation` specializes `TimedObjectActivation` that already enables the capability to register a time event occurrence at the event pool.

To use the newly defined object activation, a new type of object is defined: `FMUObject`. This class is shown in Figure 8.

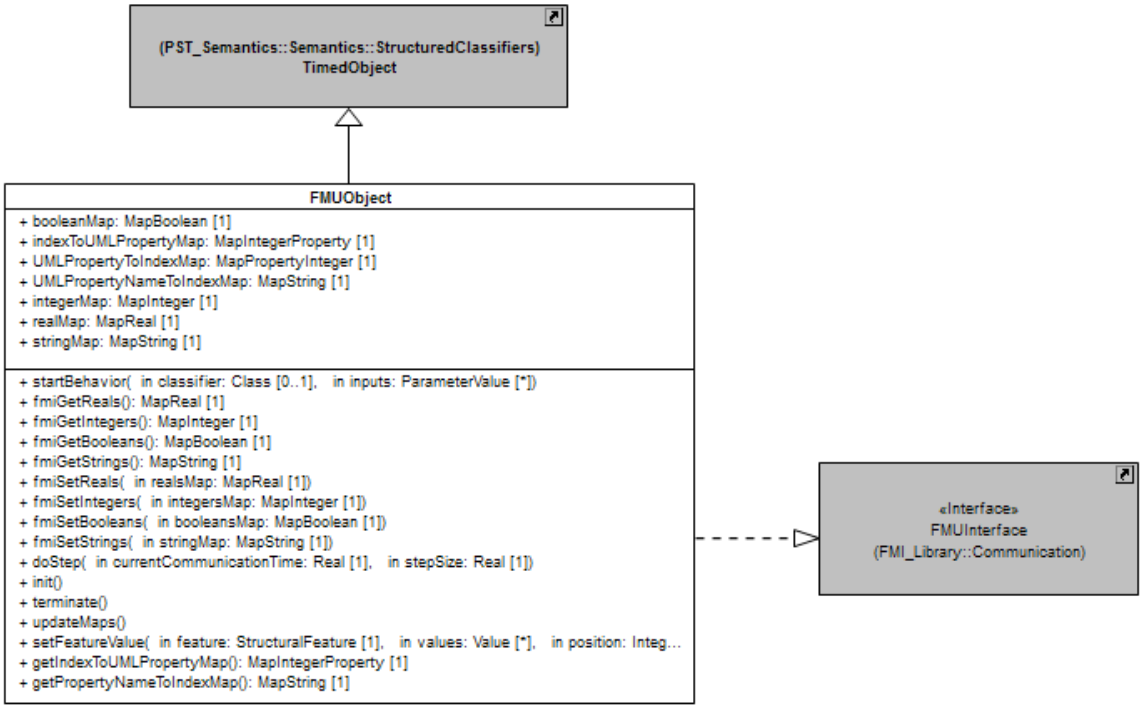


Figure 8 - FMU Object Class

The second purpose of defining a new type of object is to enable any instance of an FMU represented via this object to be manipulated directly via the master controlling the execution the execution of a graph of FMUs. To make this possible, an `FMUObject` implements the `FMUInterface` (see Figure 8). This interface provides the operations `fmiGetXXX`, `fmiSetXXX` and `doStep`. In this configuration, any instance of an FMU available at the locus (i.e., abstraction of a physical memory) can interact with any master algorithm capable of handling objects of type `FMUInterface`.

3.3.1.3 Master Interactions

This section explains how master calls are handled by FMUs in which the exported model is specified in UML. Sections 3.3.1.3.1, 3.3.1.3.2 and 3.3.1.3.3 focus on the major interactions occurring between the master and an FMU: variables assignment, triggering of doStep and variables reading.

3.3.1.3.1 Set variables values

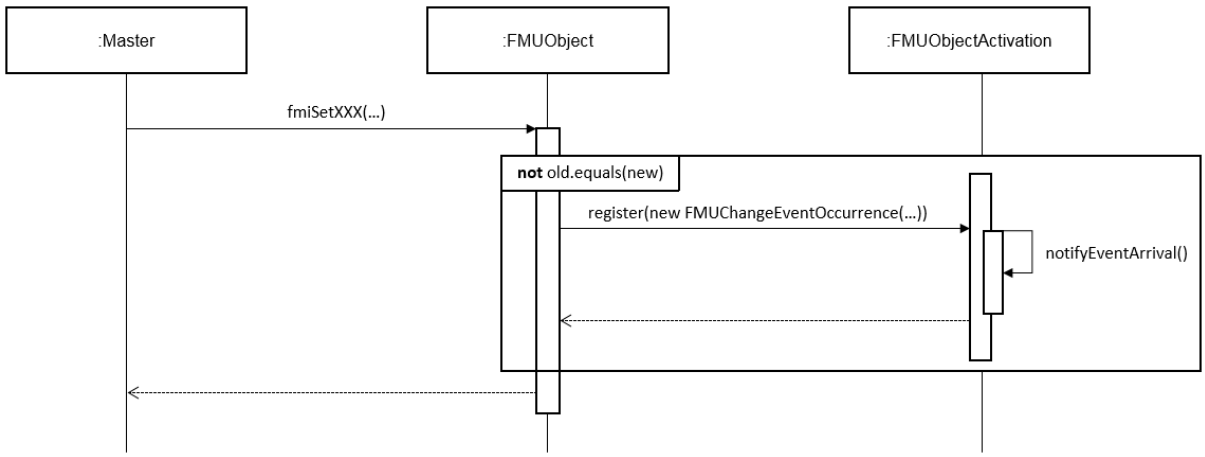


Figure 9 - Set a variable of an FMU Object

The sequence model presented in Figure 9 presents the consequences of setting a variable of an FMU object. When a variable is about to be assigned, this implies a new change event occurrence is registered at the event pool of the FMU if (and only if) the assigned value is not equal to the old value. Equality between objects is formally defined in section 8.3.3.2.19 in [5].

3.3.1.3.2 Perform a doStep

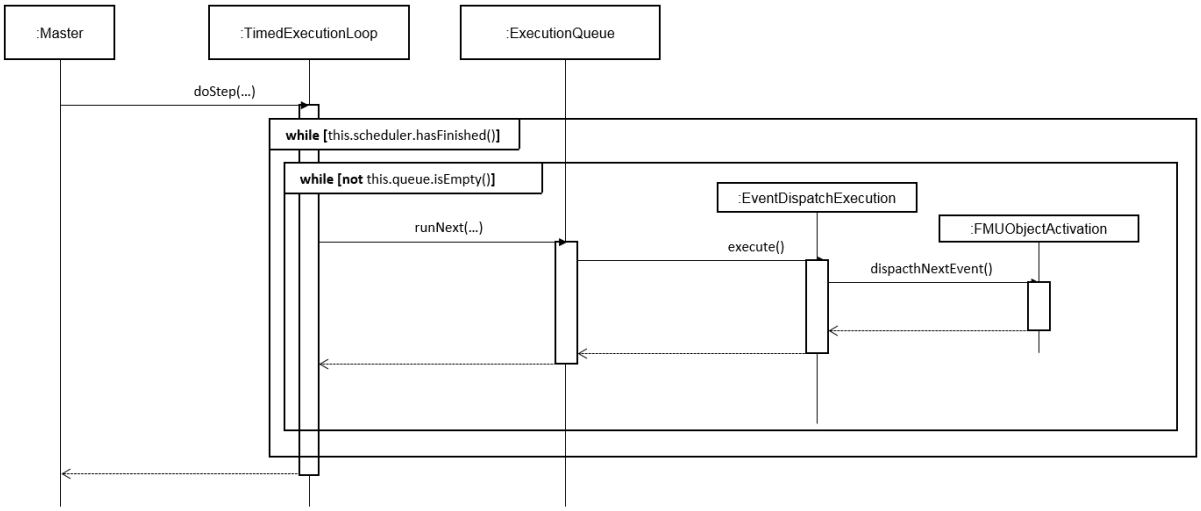


Figure 10 - Request a doStep on an FMU

The sequence model presented in Figure 10 shows the impact of a doStep request on a FMU. The principle is simple: the execution progresses according to the UML semantics until the time marking the end of the doStep is reached. In order to let the execution to progress, events of active objects are dispatched. When no event can anymore be dispatched in any active object then the time can progress. If the current time corresponds to the time at which the FMI step shall end, then the step terminates which implies the master is released and can request a doStep to another FMU.

3.3.1.3.3 Get variables values

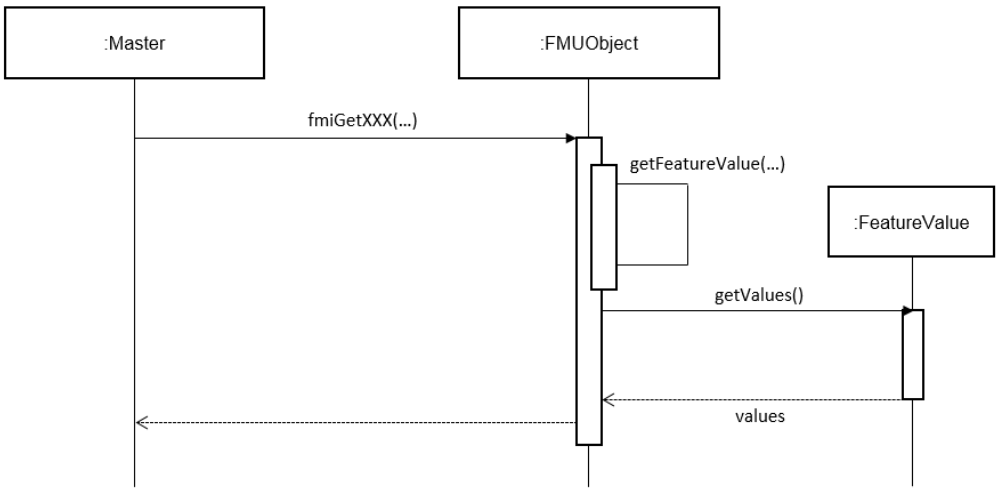


Figure 11 - Get a FMU variable value

The sequence model presented in Figure 11 shows how a request to retrieve a variable value is handled by an FMU object. The object gets the feature value (i.e., an object maintaining the relationship between) corresponding to the variable that needs to be read. As soon as this feature value is retrieved the values hold by this latter can accessed by the master.

3.3.2 Approach N°2: Wrap Models into a Higher Level Component

The approach proposed in this section is based on the idea that in order to execute a UML model in an FMU then this latter needs to be encapsulated in a higher level component. This component will then provide wrappers in charge of the translation of master calls into communications that can be natively handled by the simulation model. In order to execute a deeper evaluation of these FMU wrapper approaches, we have developed prototype generators:

- The xtUML to FMU wrapper generator is based on the Bridgepoint xtUML code generator and exemplifies the *"variables as class attribute"s* and the *"port typed by interfaces"* variants described in Section 3.3.2.2.1 and 3.3.2.2.3, respectively.
- The UML to FMU wrapper generator is based on the txtUML and exemplifies the a*ttributes of distinguished signals* variant described in Section 3.3.2.2.2.

The summary of our findings is discussed in Section 3.4, while the implementation details of the prototype xtUML and UML wrapper generators are discussed in Annex 1 [4] and Annex 2 [6], respectively.

3.3.2.1 Model Wrapping

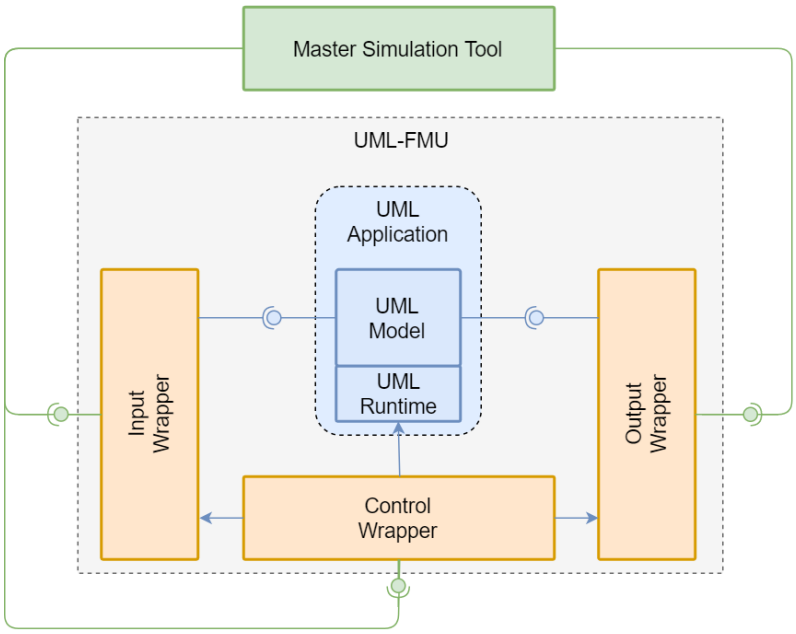


Figure 12 - Proposal to wrap a UML application model in a FMU

Figure 12 shows the general architecture of the wrapping approach. In that architecture, the simulation model specified in UML (see the blue component on the figure) is not directly exposed to master calls. Indeed, these calls are first handled by the `InputWrapper` and the `OutputWrapper`.

- The `InputWrapper` is in charge of the translation of master calls aiming to set variables of the wrapped UML simulation model. The translation consists in transforming the `fmiSetXXX` calls into messages or operation calls that can be understood by the simulation model. These messages or operation calls when received by the simulation model will then imply an update of the targeted variable.
- The `OutputWrapper` is in charge of the translation of master calls aiming to read variables of the wrapped UML simulation model. The translation consists in transforming the `fmiGetXXX` calls into messages or operations calls that can be understood by the simulation model. These messages or operation calls when received by the simulation model will then enable the wrapper to get access to the targeted variable. At this point, the variable value will be communicated back to the master.

Figure 12 also set the focus on a third component: the `ControlWrapper`. This component is in charge of handling the remaining FMI functions (i.e. lifecycle and simulation control functions). It is also responsible for scheduling the execution of the UML application and possibly modifying the states of the other wrapper components (e.g. resetting them between simulation steps).

3.3.2.2 Modeling Constrains

This section identifies the modeling constraints applying on the application design for each variant of the wrapping approach. These modeling constraints are respectively detailed in sections 3.3.2.2.1, 3.3.2.2.2 and 3.3.2.2.3.

3.3.2.2.1 Variables as Class Attributes

This solution uses the attributes of a selected UML class as the exposed variables of the FMU. The selected class must be a singleton, so that each exposed variable can be matched to a single value during runtime, or the class must expose static (class-level) attributes for the same purpose. Note that output variables can be marked as calculated using the `isDerived` (see section 9.9.17.5 in [2]) attribute of a `Property` (see section 9.9.17 in [2]). This implies that the value of an output depends one to many other variables values.

The behavior of the class is expected to be a state machine. This state machine can only react to the reception of a specific signal: `DoStep`. This signal will be available in a model library and be exclusively sent by the master. Such constraint implies the state machine can only have transitions with a trigger referencing a signal event for the `DoStep` signal. Guards placed on transitions will implement the conditions allowing the state machine to switch from one state to the other. A state machine conforming to these modeling constraints is shown in Figure 13 (this state machine is a refined version of the one presented in Figure 2).

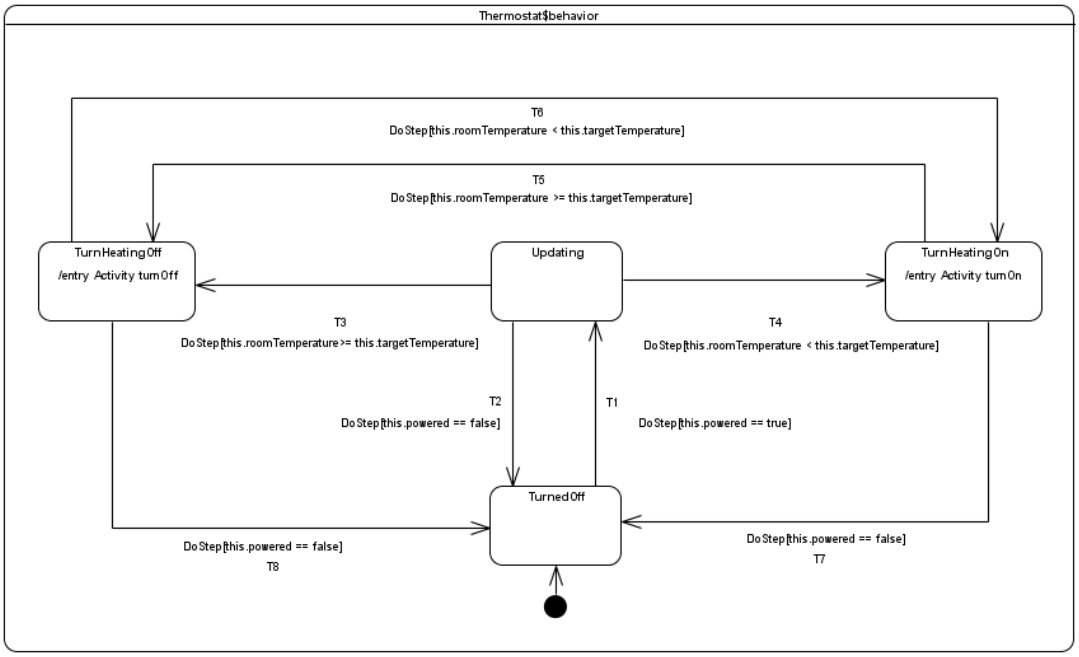


Figure 13 - Every transition is triggered by `DoStep`

It is very important to note that with this configuration it is only possible for the state machine defining the behavior of the class to traverse at most one transition per FMI step. This means one step requested by the master implies at most one RTC step to be performed in the state machine.

3.3.2.2.2 Attributes of Distinguished Signals

This solution proposes to group input values provided to an FMU and output values produced by an FMU in signal attributes. The role of these signals is to enable the passing of data from the input wrapper to the UML application and also from the UML application to the output wrapper. In this approach, it is not mandatory for the UML application to maintain internal properties holding the values provided as inputs.

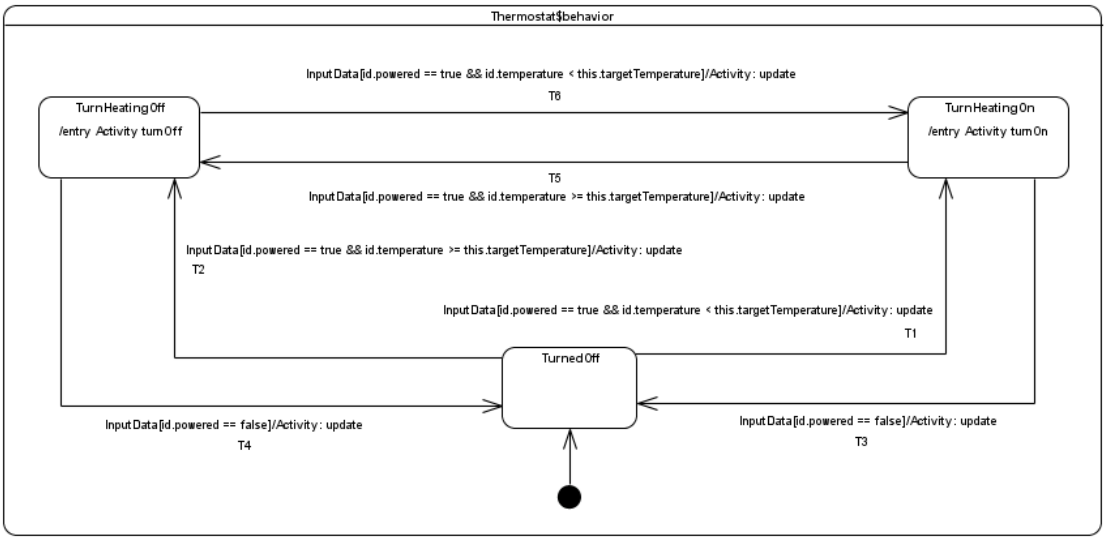


Figure 14 – Every transition is triggered by `InputData`

The behavior attached to the UML application is intended to be a state machine. Figure 14 shows the state machine that may implement the `Thermostat` behavior. Such state machine can only have transitions triggered by the acceptance of an `InputData` signal.

The signature (i.e., signal attributes) of the signal is aligned with the number of inputs that are expected to be received by the UML application. Figure 15 shows the specification of the `InputData` signal in the context of the `Thermostat` example. This signal has two attributes that represents inputs (i.e., current room temperature and power status) that need to be taken into account by the UML application. Figure 15 also shows the specification of the `OutputData` signal. This signal has a single attribute representing the output produced by the UML application.

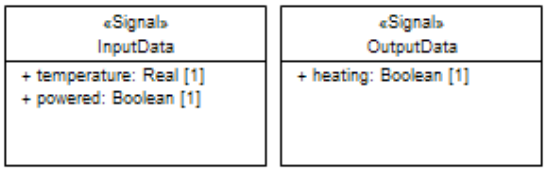


Figure 15 - Input and Output signals

It is important to note that with this approach, the UML application is intended to define a reception for the `InputData` signal. This signal is the only one that when accepted can make the state machine defining the UML application behavior to evolve. In addition, the `OutputData` signal is the only type of signal that can flow out of the UML application.

3.3.2.2.3 Port Typed by Interfaces

In this solution, the provided features (i.e., messages that can be accepted by the class) represent the input variables and the required features (i.e., messages that can be sent by the class) represent the output variables. This implies that each attribute in signal that can be received by the UML application is considered as an input variable of the FMU. In addition, attributes of

signals that can be received by the UML application are considered as outputs of the FMU. By mapping FMI variables on signals that can be sent and received, any UML model can be used through its original interface; no other design constraints apply on the model.

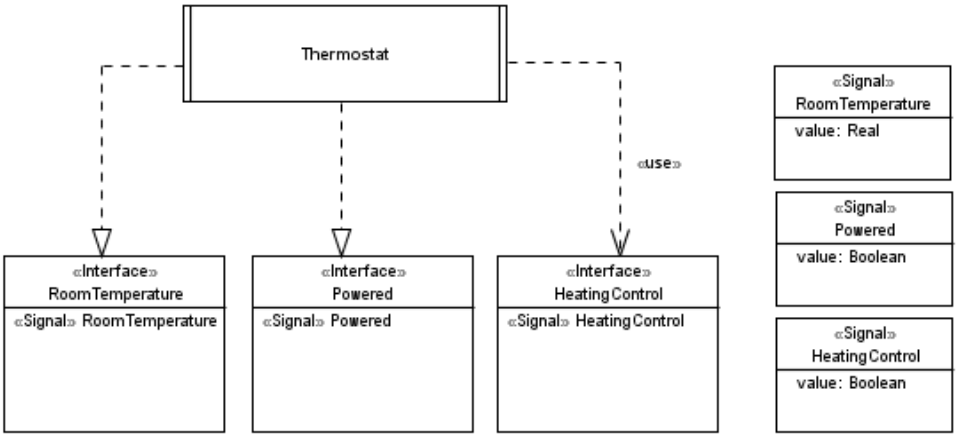


Figure 16 - Provided and Required Signals

Figure 16 shows signals that can be sent and received by the `Thermostat`. Each signal (see right hand side of the figure) defines an attribute describing the type of data that need to be provided either as an input or an output. In the context of the `Thermostat` example, the current room temperature and the power status corresponds to inputs while the heating order corresponds to the output.

The behavior attached to the UML application is intended to be a state machine. As there are no constraints regarding the communications that can be received and emitted by the UML application then no change is required to the original state machine. Hence, this state machine corresponds to the one presented in Figure 2.

3.3.2.3 Master Interactions

This section defines for each variant of the wrapping approach their interactions with the master when setting input variables, requesting a doStep and getting output variables. These interactions are respectively presented in sections 3.3.2.3.1, 3.3.2.3.2 and 3.3.2.3.3.

3.3.2.3.1 Variables as Class Attributes

Wrapping

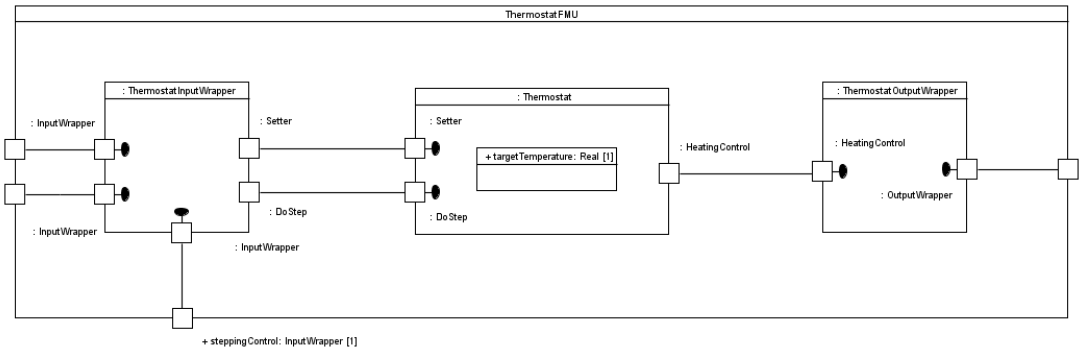


Figure 17 - Class Attributes Wrapped Model

Figure 17 shows an application of the wrapping strategy on a UML application whose main applicative component exposes its internal attributes as variable of the FMU. As for the general architecture, interactions with the master algorithm are handled by the input and output wrappers. Communications between the input wrapper and the UML application are always synchronous. This is also true for communications occurring between the UML application and the output wrapper. These communications are formalized as call to operations. The called operations are the setters and getters provided by the UML application for its internal properties.

Set Variables

Figure 18 shows that when the master requests to set a variable then this request is translated by the wrapper into an operation call dispatched on the class instance representing the main application component at runtime. After this call was dispatched the value of the property in the class instance is updated and the master `fmiSetXXX` requests can terminate.

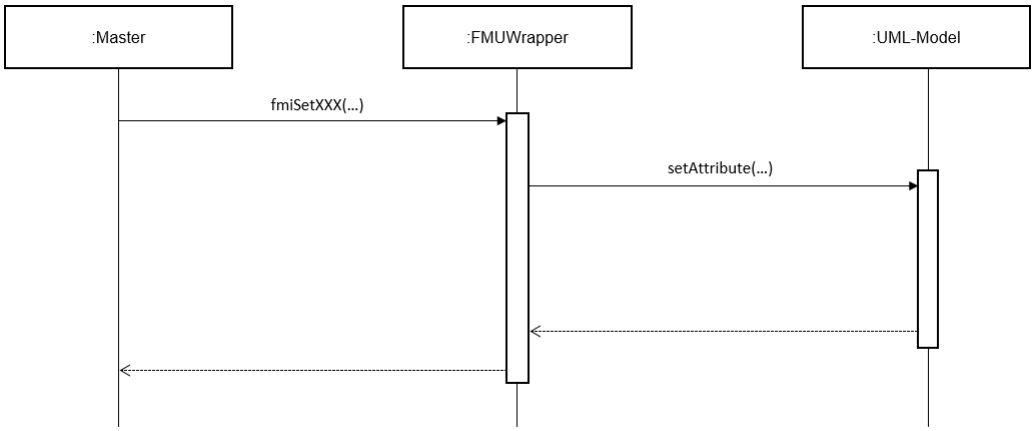


Figure 18 - Set variable – “Class attributes” approach

Do Step

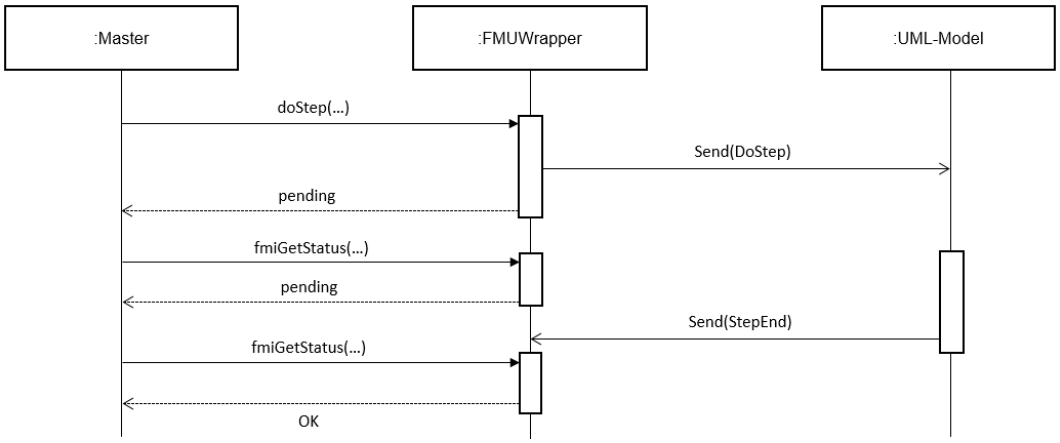


Figure 19 – Request a doStep – “Class attributes” approach

Figure 19 shows that when the master requests a doStep to be performed then the wrapper receiving the call sends a `DoStep` signal event occurrence to the class instance representing the main application component at runtime. After the signal was sent, the wrapper waits for the

notification indicating the end of the step on the application side. While the wrapper is waiting for the application notification, the master can also poll its status to check if the step is ended or not for the FMU. In this situation, the status will remain pending until the application responds to the wrapper. As soon as the application has responded, the status will be updated and it will be possible for the master to read that status.

Get Variables

Figure 20 shows that when the master requests to read a variable then this request is translated by the wrapper into an operation call dispatched on the class instance representing the main application component at runtime. After this call was dispatched the value of the property in the class instance is returned and the master `fmiGetXXX` request of can terminate. At this point the master knows the value that was read.

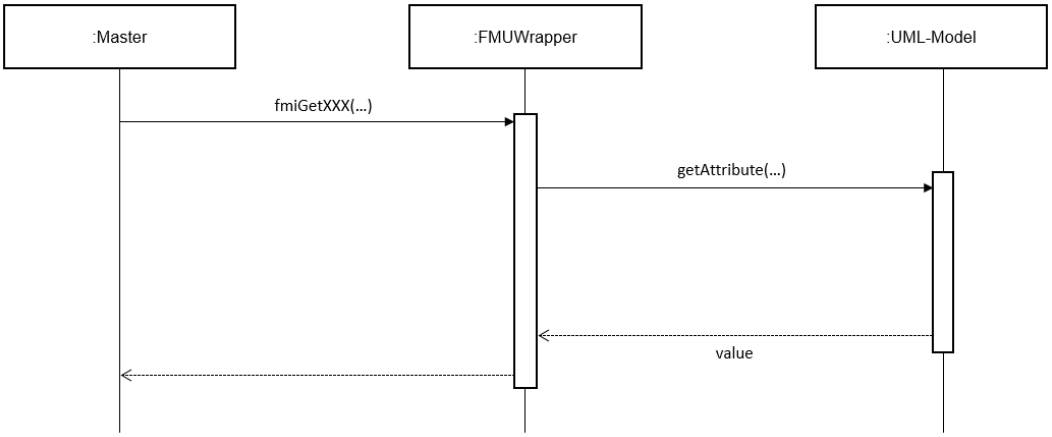


Figure 20 – Get variable – “Class attributes” approach

3.3.2.3.2 Variables as Signal Attributes

Wrapping

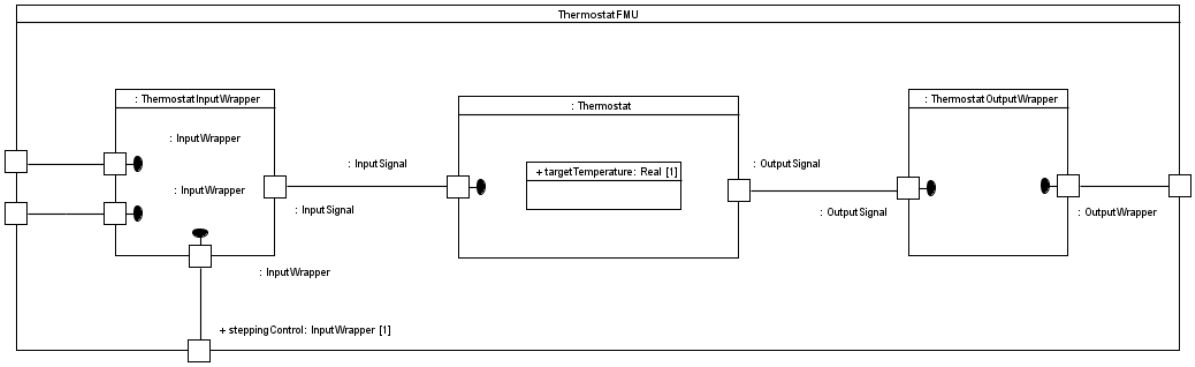


Figure 21 - Variable as Signal Attributes - Wrapped Model

Figure 21 shows an application of the wrapping strategy on a UML application designed according to the modeling constraints specified in section 3.3.2.2.2. In this instantiation of the general wrapping architecture, the input and output wrappers communicates with the UML application thanks to signal passing. It exists at most one output port for the input wrapper. This

port enables `InputData` signals to be sent from the input wrapper to the UML application. It also exists at most one input port for the output wrapper. This port enables `OutputData` signals to be received by the output wrapper from the UML application.

Set Variables

Setting of the FMU variables for this approach is different from the one presented in Figure 18. Indeed, in this approach when an `fmiSetXXX` operation call is received by the input wrapper from the master, the input wrapper updates the value of an internal property used to store the value. The update of the input is performed thanks to a call to a setter provided by the input wrapper. Such a behavior is presented in Figure 22.

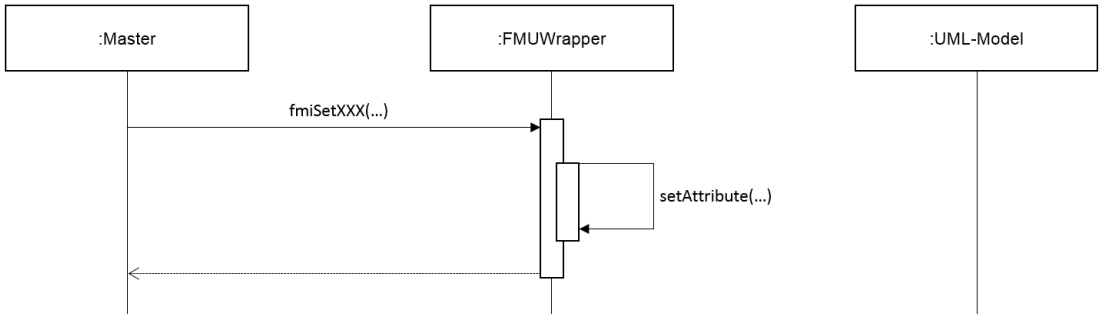


Figure 22 - Set Variable – “Signal attributes” approach

Do Step

Figure 23 shows how a `doStep` operation call is handled by the input wrapper. When received, the operation call triggers the sending of specific signal `InputSignal` to the UML application. Values associated to the event occurrence are the input values received by the FMU during the variable setting phase. When the event occurrence is accepted by the UML application a RTC is performed in the state machine defining its behavior that may lead to the sending of an output signal to the output wrapper.

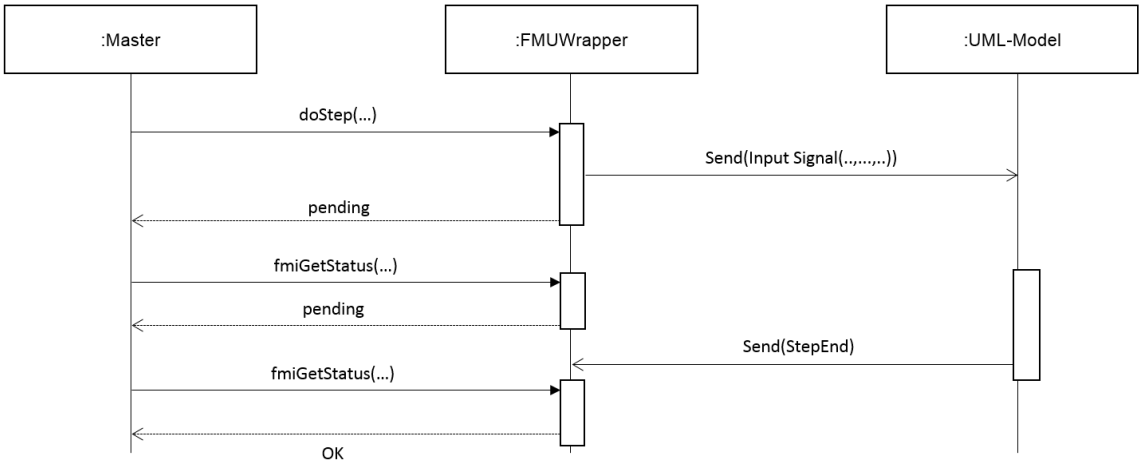


Figure 23 - Do Step - "Signal attributes" approach

Get Variables

Figure 24 shows how a request to get an output is handled by the output wrapper. When received, the `fmiGetXXX` call triggers the execution of the reading of an output stored at the output wrapper. This reading is performed thanks to the execution of the getter corresponding to the target variable.

It is important to note that variables values maintained by the output wrapper can only be updated upon the reception of an `OutputSignal` event occurrence. Values hold by the signal attributes are used to updates the output wrapper variables values.

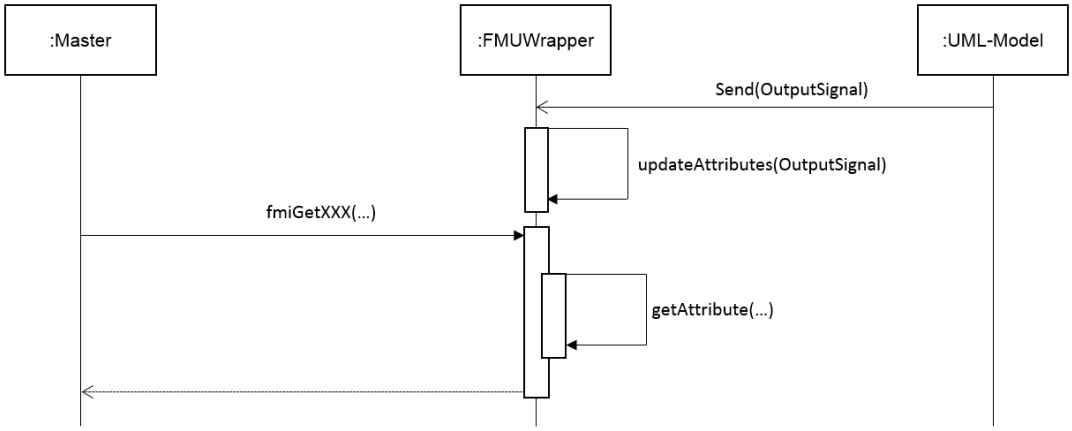


Figure 24 – Get Variable - "Signal attributes" approach

3.3.2.3.3 Ports Typed by Interfaces

Wrapping

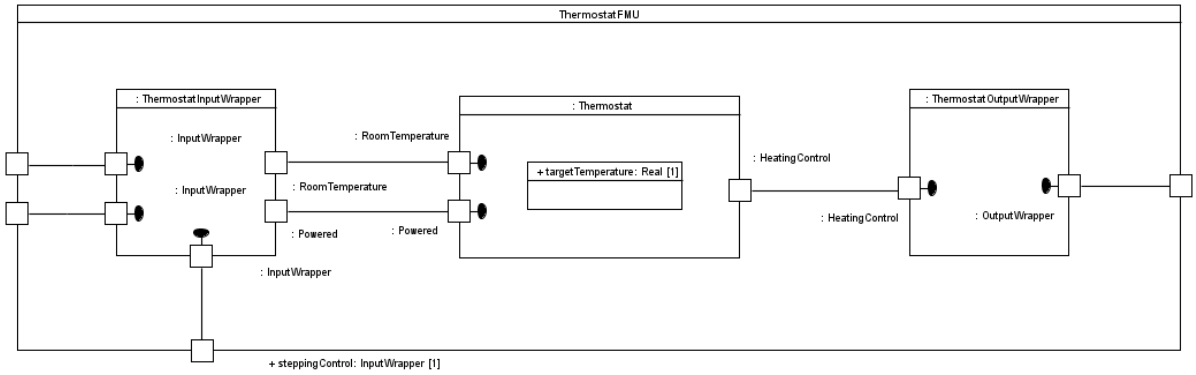


Figure 25 - Ports Typed by Interfaces - Wrapped Model

Figure 25 shows an application of the wrapping strategy on a UML application designed according to the modeling constraints specified in section 3.3.2.2.3.

In this instantiation of the general wrapping architecture, the input and output wrappers communicates with the UML application thanks to signal passing. Passed signals are those originally intended to be received by the UML application. This implies the input wrapper output ports will require the interfaces provided by the UML application input ports while the output wrappers input ports will provide the interfaces required by the UML application output ports.

Set Variables

Figure 26 shows how a call to the `fmiSetXXX` operation is handled by the input wrapper. When such call is received it is translated as call to the `cacheValue` operation. This operation is provided by an `InputCache` object which is in charge of the storage of input variables values.

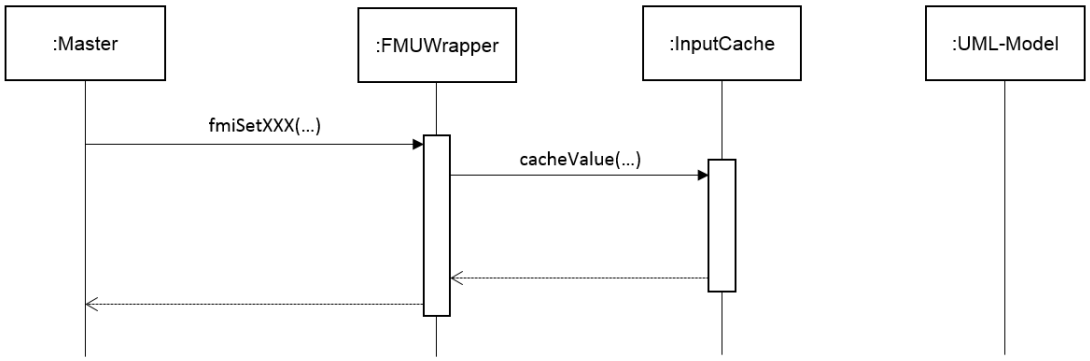


Figure 26 - Set Variable - "Ports typed by interfaces" approach

Do Step

Figure 27 shows how a call to the `doStep` operation is handled by the input wrapper. When such call is received, the wrapper queries cached value that are required to be sent to the UML application. Each value is then encapsulated in a signal event occurrence and sent to the UML application. This latter dispatches and accepts received events occurrences until the event pool gets emptied. When the event pool is empty and the UML application is in stable state (i.e., there are not anymore possibilities to continue the execution) it sends a `StepEnd` signal to the input wrapper. This signal indicates the end of the simulation step on the UML application side. The master has then the possibility to account for the completion of the step.

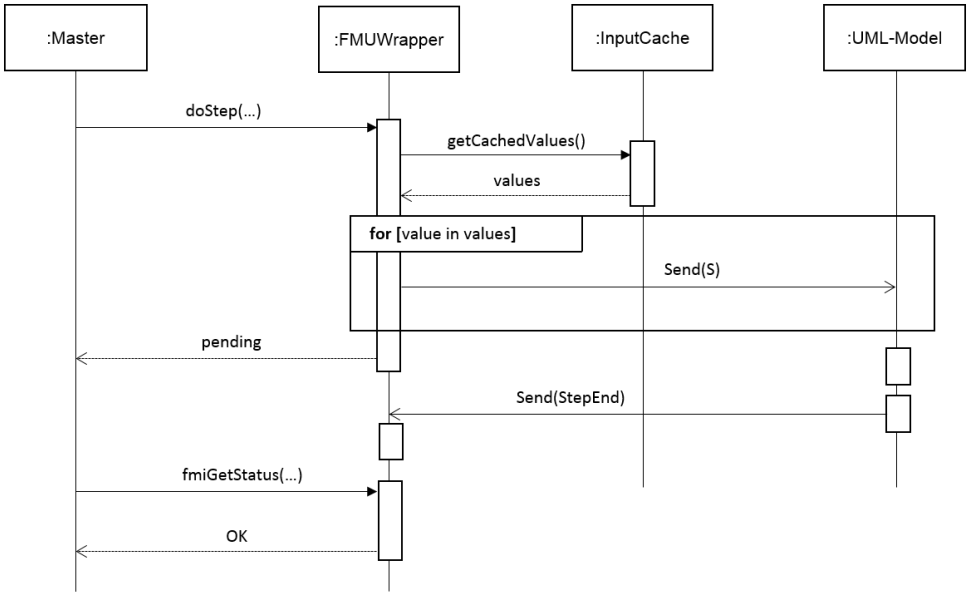


Figure 27 – Do Step - "Port typed by interfaces" approach

Get Variables

Figure 28 shows how a call to the `fmiGetXXX` operation is handled by the output wrapper. When such call is received it is translated into a call to the `getCachedValue` operation provided by the `OuputCache`. This call returns the value attached to the target variable. It is important to note that the value maintained by the `OutputCache` for a specific variable can only have been updated if the output wrapper received a signal from the UML application specifying the new value.

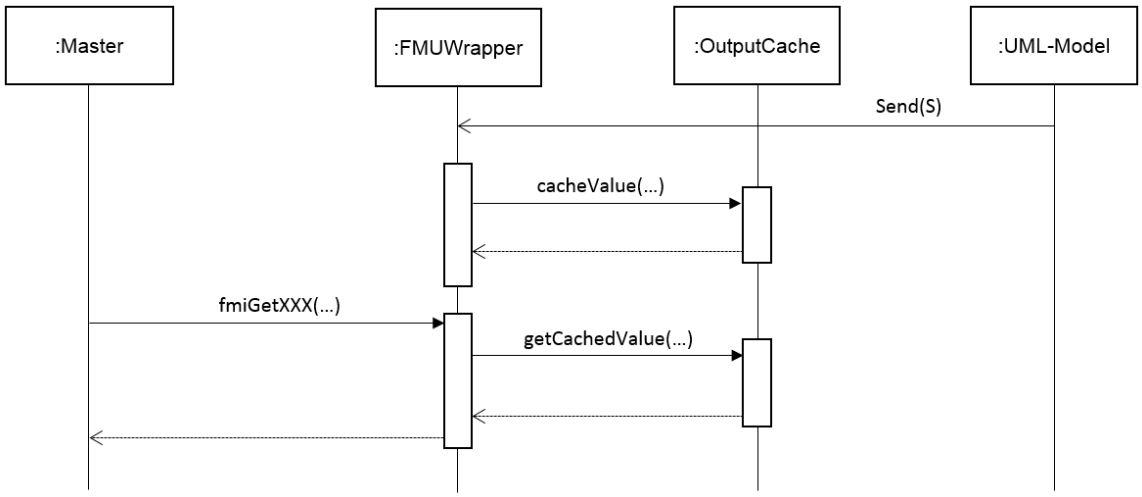


Figure 28 - Get Variable - "Ports typed by interfaces" approach

Sections 3.3.1 and 3.3.2 presented two approaches to integrate system specification described with UML / xtUML into an FMU. The approach presented in section 3.3.1 proposes to handle the FMU integration thanks to extension of the semantics defined for UML. Conversely, the approach presented in section 3.3.2 proposes to handle the FMU integration through the wrapping of the original UML application into an higher level component providing interfaces directly compatible with those provided by a master algorithm. Three different variants of the second approach are proposed.

Next section presents a comparison of the four different solutions.

3.4 Comparisons

The purpose of this comparison is to determine the approach offering the best trade-off between the integration quality and the tooling required to be developed in order to make the integration usable. The evaluation criteria are the following:

1. **Compatibility**
   - Specify the compatibility of the different approaches with the languages (i.e., UML and xtUML) used to design the application to be exported in the FMU.
   - Values:
     - UML – xtUML – UML / xtUML
2. **Constraints on the original design**
   - Specify if the approach used to integrate the application in a FMU requires the original design to be refactored.

- Values:
  - YES – NO

3. **Efforts to perform the changes on the original design**
   - Evaluate the efforts required to change the original design in order to conform to the modeling constraints implied by the chosen approach.
   - Values:
     - WEAK – AVERGAGE – SIGNIFICANT - NONE
4. **Capability to automate the changes on the original design**
   - Specify if the changes to be applied on the original design in order to conform to the constraints implied by a chosen approach can be automated.
   - Values:
     - YES – NO
5. **Efforts to automate the changes on the original design**
   - Evaluate the difficulty to implement tools to automate the changes to be performed on the original design per approach.
   - Values:
     - WEAK – AVERAGE – SIGNIFICANT – NONE

	Semantic Extension	Variables as Class attributes	Variables as Signal Attributes	Ports typed by interfaces
Compatibility	UML. This approach was not evaluated on xtUML.	UML / xtUML	UML / xtUML	UML / xtUML
Constraints on the original design	YES. It is mandatory for the designer to use the FMI profile to configure the FMU. In addition, triggers on transitions can only be for change events and time events.	YES. It is mandatory for the designer to define getters and setters for all properties that are desired to be exposed as FMU variables.	YES. The application can only have a single input and output port. The input port enables receptions of InputData signals while the output enables receptions of OutputData signal.	NO.
Effort to perform the changes on the original design	AVERAGE. FMU stereotypes need to be applied on the application. In addition the state machine transition shall be refined.	SIGNIFICANT. Getters and setters for FMI variable are required to be provided. The behavior shall be refined to only account for DoStep signals.	SIGNIFICANT. The behavior shall be refined to only account for InputData signal events and send variable updates as OutputData signal events..	NONE.
Capability to automate the changes on the original design	YES.	YES	YES	NO
Effort to automate the changes on the original design	WEAK. For instance, Papyrus already provides the capability to use the FMI profile. Few more accelerators are required to make the application of stereotypes on multiple model elements in a single click.	AVERGAGE. It requires the definition of an in-place model transformation.	AVERAGE. It requires the definition of an in-place model transformation.	NONE.

The comparison sets the focus on the wrapping approach, in particular on the "*Port typed by interfaces*" variant. Indeed, this variant of the wrapping approach is compatible with both UML languages and does not require the original model to be changed (i.e., the wrappers are designed to communicate only via messages originally supported by the application).

However, it is important to mention that even if the "*Port typed by interfaces*" does not require changes in the original UML model, integrating the approach into a multi FMU simulation scenario can be cumbersome as processing the UML signals mapped to the FMU variables requires additional data transformations.

3.5 Issues to be resolved

Although this document specifies the basics to integrate simulation models described with UML or xtUML in FMU, some issues have been identified by the T2.2 task force. This issues still need to be discussed and resolved in order to enable a full usage of the UML and xtUML simulation models in the FMI context. These issues and their proposed resolutions (if any) are described in the following sections.

3.5.1 Roll-Back Support

3.5.1.1 Issue Description

An FMU supporting rollback has the capability to refuse a step requested by the master (e.g. due to an incompatible communication step size). This implies either `fmi2Error` or `fmi2Discard` status are returned to the master after a `doStep` call was received and computed by the FMU. The difficulty here is that the FMU state may have changed. Hence, before redoing a step it is required to restore the FMU to its previous state (i.e., the one known before the failure of the first `doStep`).

3.5.1.2 Issue Discussion / Resolution

At runtime, the different states of the application designed in UML or xtUML is not maintained. Hence it is not possible (at least currently) to allow an FMU encapsulating a simulation model designed in UML or xtUML to support rollback.

In order to provide a support for rollback it is required to build a trace model along the simulation of UML and xtUML models. Such models will be used to keep track of the states of the simulation models before and after each `doStep`. In this way, it will be possible to restore the state of an FMU encapsulating a simulation model specified either with UML or xtUML after the failure or the discarding of a doStep.

It is important to note that the definition and the implementation of such tracing capability is a significant work. In addition, the possibility to define a common trace meta-model for UML and xtUML is required to be evaluated. However, it can be anticipated that for the behavioral aspects, the trace models for UML and xtUML will be different. Indeed, in UML activities are used while it is not the case for xtUML which replaces them with the possibility to specify low level computations a textual action code.

3.5.2 Inputs Consumption Support

3.5.2.1 Issue Description

In approaches presented in sections 3.3.1 and 3.3.2.2.3 FMU inputs are provided in separated event occurrences. Indeed, in the approach presented in section 3.3.1 each time an input is provided a change event occurrence is placed at the event pool of the UML application. In the approach presented in section 3.3.2.2.3, all inputs are forwarded to the UML application as signal event occurrences. Each input is placed in a different signal event occurrence.

At runtime, the master algorithm expects the FMU to consume all inputs at the same time (i.e., at the beginning of a step). However, this is not what happen if modeling approaches presented in sections 3.3.1 and 3.3.2.2.3 are used. Indeed, each event occurrence corresponding to an input will trigger a RTC step that may modify the state of the UML application. Hence, as the state of the application may be modified by the acceptance of an event occurrence then the order in which these event occurrences are accepted has a great impact on the behavior of the application.

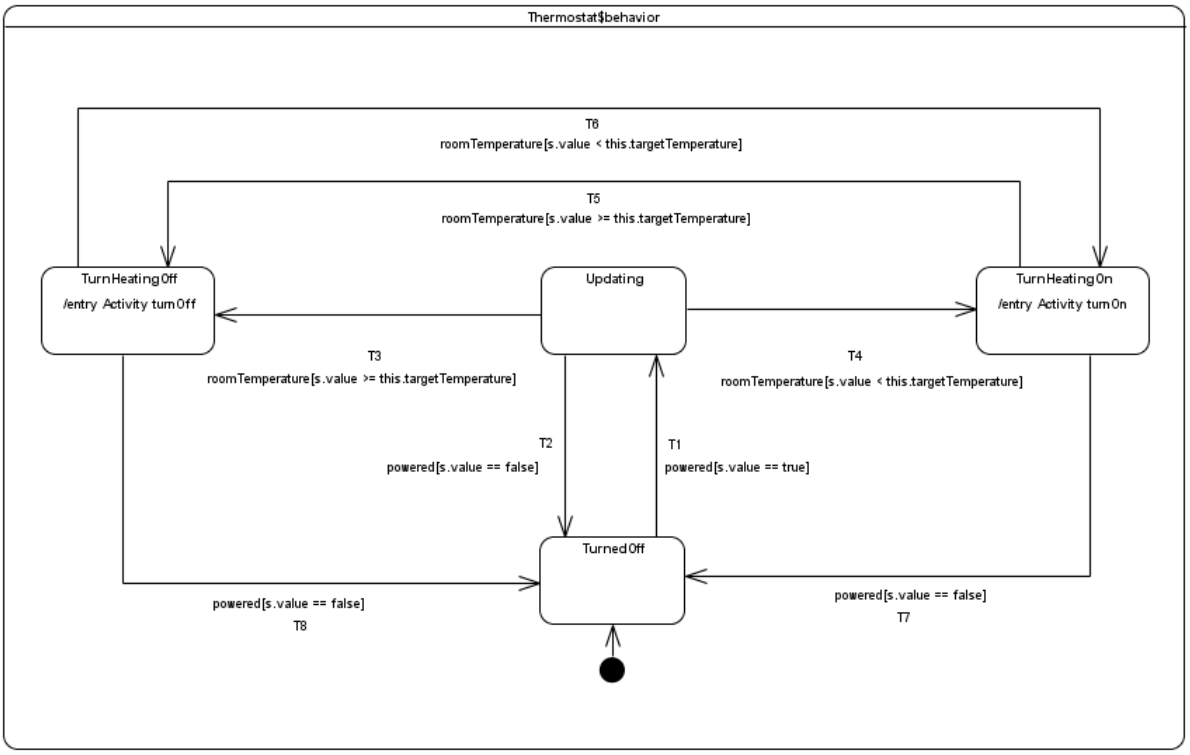


Figure 29 - Thermostat Behavior - "Ports typed by interfaces" approach

The issue can be highlighted through the example presented in Figure 29. Let consider the state machine is in state `TurnedOff` and the two different ordering of the event occurrences received by the `Thermostat`:

1. {RoomTemperature{value=19}, Powered{value=true}}
   - In such situation, the state machine can accept the `Powered` event occurrence and traverse transition `T1`. Next depending on the specified target temperature, either `T3` or `T4` can be traversed upon the acceptance of `RoomTemperature`.
2. {Powered{value=true}, RoomTemperature{value=19}}

- In such situation, the state machine is not in a configuration allowing the acceptance of the `RoomTemperature` event occurrence. Per the UML semantics, an event that is extracted from the event pool and cannot be consumed is lost. At this point the state machine remains in state `TurnOff`. When the `Powered` event occurrence is accepted then `T1` is traversed and the state machine reaches the state `Updating`.

The ordering has an impact on the behavior execution since with the two different event pools presented above, the state machine does not reach the same configuration when all events occurrences have been dispatched.

3.5.2.2 Issue Discussion / Resolution

UML and xtUML semantics are precise: an event occurrence, if accepted triggers a RTC step in the behavior of the object that accepted this latter. The issue here is inherent to the way the UML application is notified about input variables updates.

Approach presented in sections 3.3.2.3.1 and 3.3.2.3.2 only partially resolve this issue. Indeed, the approach presented in section 3.3.2.3.1 suggests that input variables are updated synchronously (via calls to setters) within the context of the UML application. Hence all variables are available locally when the `doStep` is triggered by the acceptance of a `DoStep` signal. The approach presented in section 3.3.2.3.2 suggests that variables updates and triggering of a `doStep` are combined. Hence, a step is triggered by the acceptance of single signal `InputData`. This signal event occurrence also contains all input values. These latter become available in the same RTC step.

However approaches presented in sections 3.3.2.3.1 and 3.3.2.3.2 both require the behavior of the UML application to be changed to receive a signal that is not natively used by this latter (i.e., `DoStep` and `InputData`). The best trade-off between the preservation of the of the communications natively supported by the UML application and the presence of a specific signal to trigger a step might be to:

1. Ensure all variable updates have been received first.
2. Make sure the UML application send to itself a user defined signal enabling the actual FMI step to be performed.

An alternative to this might to rely on deferred event semantics (see section 14.2.3.4.4 in [2]). It is important to note that both solutions are clearly hypothetical and require to be evaluated in order to make sure they are applicable.

3.5.3 Outputs Production Support

3.5.3.1 Issue Description

In UML or xtUML multiple messages can be propagated through output ports of a composite structure along a single RTC step. However, in the FMI context, if the output variables of an FMU are updated multiple times then only the latest values will be available at the time at which the master will propagate the output. This is an important issue since values may have been lost and so no have been communicated to the environment of the FMU.

This issue is highlighted thanks to the *Feeder* example. In this example two discrete UML models are connected in a FMI co-simulation. Their interactions are serialized to two variables between the two FMUs.

One of the models is a source and the other one is a sink. The interaction between them is the following: The `Sink` asks for a given amount of data, and the source responds with the given number of signals each one containing one unit of information. If the UML models are connected in a non-FMU simulation the result is as expected, all of the sent messages arrive. However, in an FMU environment, when using the "*Keep the last signal*[10]" strategy, for each interaction only the last data message is kept. For this reason, the behavior of the system will not be the expected.

```
@From(Running.class)@To(Running.class)@Trigger(RequestSignal.class)
class ReceiveResponse extends Transition {
@Override
public void effect() {
RequestSignal trigger = getTrigger(RequestSignal.class);
for (int i = 0; i < trigger.amount; i++) {
      Action.send(new ResponseSignal(3),
assoc(SinkSourceAssoc.sink.class).selectAny());
}}}
```

Another example from the `Sink` part:

```
@From(Running.class)@To(Running.class)@Trigger(ResponseSignal.class)
class ReceivedLastResponse extends Transition {

@Override
public boolean guard() {
return remaining <= 1;
}

@Override
public void effect() {
ResponseSignal trigger = getTrigger(ResponseSignal.class);
Action.log("" + trigger.data);
Action.log("RECEIVED " + requested + " MESSAGES");
++requested;
remaining = requested;
Action.send(new RequestSignal(requested),
assoc(Assoc.source.class).selectAny());
}}
```

3.5.3.2 Issue Discussion / Resolution

Three different strategies were discussed by the T2.2 task force in order resolve this issue. These strategies are presented in sections 3.5.3.2.1, 3.5.3.2.2 and 3.5.3.2.3

[10] This strategy ignores all messages but the last one. That is, the wrapper overwrites the output variables whenever an output signal arrives.

3.5.3.2.1 Report on Error

In this case, an error is reported upon the arrival of the second signal of the same type in the same simulation step. This solution is safe and simple but it only works well with UML models that behave according to this limitation.

3.5.3.2.2 Arrays of Values

This solution is a generalization of the one in section 3.5.3.2.1. For each signal type *s*, an upper bound ub_s is defined: The model is allowed to send signals of type *s* at most ub_s times within one simulation step. For the signal type *s*, there are several exported variables: An integer n_s, which tells how many signals of type *s* have been sent during the simulation step, and for each attribute *a* of *s*, there are ub_s exported variables, $a_1, a_2, \ldots, a_{ub}$, to hold the values of the signal attribute in the signal instances that have been sent. When a simulation step completes, then only the first n_s variables, namely $a_1, a_2, \ldots, a_n$ contain relevant values. If no output signal of type *s* appeared in the simulation step, then $n_s=0$, and none of the $a_1, a_2, \ldots, a_{ub}$ variables hold relevant data. If the upper bound of a signal is violated, then an error is raised and the co-simulation is stopped. This encoding is similar to a variable sized bounded vector, which is encoded using an integer (length) and a partially filled buffer.

3.5.3.2.3 Rollback

This solution assumes that UML models are enriched with time information, and the simulators of the UML models support roll-back (see issue on rollback support in section 3.5.1), that is, going back to previous states in the execution. If these requirements are fulfilled, then a UML-FMU detecting more than one signal of the same type can inform the master simulation tool that it is in a un stable state, and ask for smaller execution step. All the co-simulated models will then be rolled back to the beginning of the simulation step. The smaller step size will then give a chance to the problematic FMU to emit at most one signal (of each type), so that the problem gets resolved.

3.5.4 Time Support

3.5.4.1 Issue Description

The integration of UML and xtUML models in FMU requires that interpreters for these models enable the capability to account for time (i.e. to have the capability to maintain a representation of the simulated time). The purpose of having this representation of time is to have the capability to control that steps performed on the UML application side are constrained by the step size imposed by the master algorithm when it requests a `doStep`.

Both xtUML and UML provide the capability to specify time constraints on the model (e.g., information on the model specifying the execution during of model elements or dates at which a time event shall be received). However:

1. Time in UML has no formal semantics (i.e., the time semantics is not defined in a way that allows time to be represented during a simulation). Indeed, time is not part of the UML subset considered by the Executable UML specifications: fUML [5], PSCS [7] and PSSM [8].
2. Time has semantics in xtUML but its representation is based on real-world time rather than an abstraction of time. This works fine with models that are meant to be executed

in a real environment, but during a simulation it would cause problems. To be able to simulate time in an xtUML model, it has to be based on the simulation time.

Both issues are required to be resolved in order to make sure that modeled timings can be accurately simulated and to prevent a simulation models to execute more tasks than allowed in the time frame specified by the master algorithm for the requested step.

3.5.4.2 Issue Discussion / Resolution

3.5.4.2.1 xtUML

In order to have the possibility to represent simulated time, the proposal is to substitute the existing timers with modified ones that increment the time based on the simulation steps, the same timed events can then be used as in the original implementations. The events delayed to a certain point in time will occur as soon as the simulation time passes that timestamp.

In order to ensure that simulated computation time fits into the constraint imposed by the step size requested by the master, it is proposed to introduce the notion of *remaining execution duration* of operations, which has to be maintained for the next operation. If the operation cannot be completed in the current step, then the remaining duration of the simulation step should be subtracted from the remaining duration of the operation, and the event processing should be paused. The operation should be examined again in the next step. The operation should only be executed in the simulation step it can be completed in, to make sure it doesn't yield results before it should be able to.

Note that this method also provides solution to the problem of infinitely running execution loops. If a model generates internal events during its execution, then the event queue might never be empty, causing the run-to-completion method to become an infinite loop. By halting event processing if a certain amount of work is completed, this cannot happen. Even if the execution times of operations are not provided in the model, a special timer could be introduced, that ends the simulation step and pauses the execution loop after a certain amount of time, to avoid infinitely running simulation step.

3.5.4.2.2 UML

In order to represent time during simulation semantics of time in UML was formalized. This extension is defined as an extension of the Executable UML standards: fUML [5], PSCS [7] and PSSM [8]. In particular it integrates a support for `TimeEvent` (see section 13.4.10 in [2]). However, it is important to note that this semantics are not normative. Hence support for time semantics in UML models is currently specific to Moka.

As specified in section 3.3.1, Moka provides an extension of the Executable UML semantics that enables the integration between FMI and UML. This extension combined to the one adding the capability to account for time during simulation guarantee that a UML model executed in an FMU will not execute more operations than allowed in a step requested by the master during a doStep. Indeed, when the master requests a step, the discrete clock handling the simulation time is notified (i.e., it receives a notification about the time at which the simulation step shall end). If the behavior of the UML application suspends for time events, then if these events are timestamped with a date that is on the future of the end of the FMI step then these events will not be dispatched. If these events are not dispatched, then they cannot trigger any RTC step in

the UML application. It is important to note that the aforementioned behavior make the assumption that the designer of the application provided specifications regarding timing. If not, then everything is executed in zero-time meaning that a simulation step will end only when all events in the pool of the UML application have been dispatched. This situation may obviously lead to situations where the requested FMI step never ends.

It is also important to note that this handling of time and FMI integration through a semantics extension is specific to Moka. The same model in a different tool may be executed in a different manner if it does not conform to the same semantics extensions.

4 CONCLUSIONS

This document describes two families of approaches to integrate simulation models specified in UML or xtUML in FMUs.

The first approach (see section 3.3.1) advocates for an integration based on semantics extensions. While this approach applies well for the UML context, it does not fit with xtUML whose semantics is not intended to be extended. In addition, this approach is currently specific to the implementation provided in Papyrus Moka.

The second approach (see section 3.3.2) advocates in favor of an integration based on wrapping strategy. The core idea is to have input and output wrappers responsible for translating master requests in communications that can be understood by the simulation model (i.e., the UML or xtUML application). Three variants of this approach were proposed.

- Two of them (see sections 3.3.2.2.1 and 3.3.2.2.2) impose strong constraints regarding the refinement to be applied on the original simulation model in order to make it usable in the context of an FMU. In particular, they both suggest the usage of domain specific signals to control the execution of the application behavior. These approaches will likely not be accepted by the final users if the refinement of the original model cannot be made automatically. While the automation could be made, it requires a development effort that cannot be neglected.

- The third variant (see section 3.3.2.2.3) has the advantage to preserve the original design from being changed while still enabling its integration within an FMU. The core idea is to make sure that wrappers enabling the communications with the simulation model only uses communication natively handled by this latter. However, one significant issue with this approach is the way inputs and outputs are "*provided to*" the simulation model. Inputs are sent as signal events occurrences. These event occurrences are sent asynchronously and received in a certain order. The order in which the event occurrences are dispatched has a great impact the behavior realized in the simulation model (see issue presented in section 3.5.2).

In presents situation, the approach based on the wrapping strategy is applicable both to UML and xtUML. In addition, the third variant "Port typed by interfaces" is one minimizing the impact on the original design. Hence, the task T2.2 task force promotes the usage of this approach to integrated UML and xtUML with the FMI standard. However, the acceptability of the solution is strongly conditioned by the availability of the tooling enabling designers to

automatically produce the input and the output wrappers in charge of making the translations between the master requests and the communications with the underlying simulation model.

Further developments to better integrate FMI with UML and xtUML should address issues presented in section 3.5.

TABLE OF FIGURES

Figure 1 - Test Case Environment....5
Figure 2 - Behavior specification of Thermostat FMU....5
Figure 3 - Excerpt of a UML profile for FMI....8
Figure 4 - Thermostat FMU defined with UML and the FMI profile....8
Figure 5 - Semantics for ChangeEvent....10
Figure 6 - Semantics for TimeEvent....11
Figure 7 - Semantics for FMUObjectActivation....11
Figure 8 - FMU Object Class....12
Figure 9 - Set a variable of an FMU Object....13
Figure 10 - Request a doStep on an FMU....13
Figure 11 - Get a FMU variable value....14
Figure 12 - Proposal to wrap a UML application model in a FMU....15
Figure 13 - Every transition is triggered by `DoStep`....16
Figure 14 – Every transition is triggered by `InputData`....17
Figure 15 - Input and Output signals....17
Figure 16 - Provided and Required Signals....18
Figure 17 - Class Attributes Wrapped Model....18
Figure 18 - Set variable – “Class attributes” approach....19
Figure 19 – Request a doStep – “Class attributes” approach....19
Figure 20 – Get variable – “Class attributes” approach....20
Figure 21 - Variable as Signal Attributes - Wrapped Model....20
Figure 22 - Set Variable – “Signal attributes” approach....21
Figure 23 - Do Step - "Signal attributes" approach....21
Figure 24 – Get Variable - "Signal attributes" approach....22
Figure 25 - Ports Typed by Interfaces - Wrapped Model....22
Figure 26 - Set Variable - "Ports typed by interfaces" approach....23
Figure 27 – Do Step - "Port typed by interfaces" approach....23
Figure 28 - Get Variable - "Ports typed by interfaces" approach....24
Figure 29 - Thermostat Behavior - "Ports typed by interfaces" approach....28

REFERENCES

[1] MODELISAR Consortium / Modelica Association Project, "Functional Mock-up Interface for Model Exchange and Co-Simulation," 2014.

[2] OMG, "OMG Unified Modeling Language (UML)," 2015.

[3] OneFact, "Executable, translatable UML with BridgePoint," [Online]. Available: https://xtuml.org/.

[4] OpenCPS Consortium, "Annex 1 of D2.2: xtUML to FMU prototype generator".

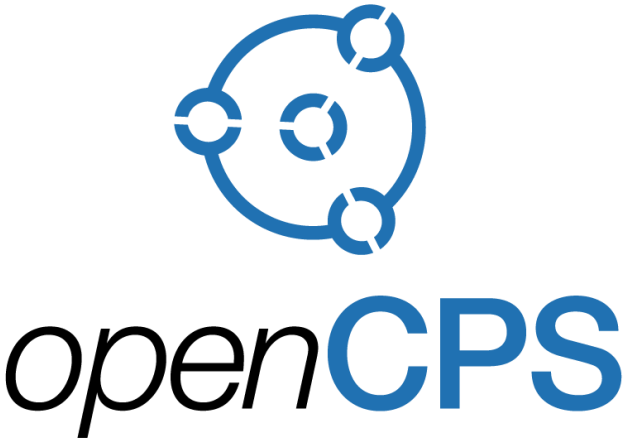
[5] OMG, "Semantics of a Foundational Subset for Executable (fUML)," 2016.

[6] OpenCPS Consortium, "Annex 2 of D2.2: The txtUML modeling tool".

[7] OMG, "Precise Semantics for UML Composite Structures (PSCS)," 2015.

[8] OMG, "Precise Semantics for UML State Machines," 2015.

Annex 1 of D2.2	xtUML to FMU Prototype Generator
Access[1]:	**PU**
Type[2]:	**Prototype**
Version:	**1.1**
Due Dates[3]:	**M24**



Open Cyber-Physical System Model-Driven Certified Development

Executive summary[4]:

Annex 1 of D2.2 focuses on the developed xtUML FMU wrapper generator prototype that helped us to understand the different options available for mapping xtUML methods and events to FMU variables. Our findings about the different wrapping based approaches are summarized in ection 3.3 of D2.2.

1Access classification as per definitions in PCA; PU = Public, CO = Confidential. Access classification per deliverable stated in FPP.
2 Deliverable type according to FPP, note that all non-report deliverables must be accompanied by a deliverable report.
3 Due month(s) according to FPP.
4 It is mandatory to provide an executive summary for each deliverable.

Deliverable Contributors:

	Name	Organisation	Primary role in project	Main Author(s)[5]
Deliverable Leader[6]	Ákos Horváth	IncQuery Labs	Task leader (for the annex)	X
Contributing Author(s)[7]	Rebeka Farkas	IncQuery Labs	Contributor	X
	Krisztián Mócsai	IncQuery Labs	Contributor	X
	Zoltán Ujhelyi	IncQuery Labs	Contributor	X
	Dániel Segesdi	IncQuery Labs	Contributor	X
Internal Reviewer(s)[8]				

Document History:

Version	Date	Reason for Change	Status[9]
0.1	10/11/2017	First Draft Version	Draft
1.0	17/11/2017	Final version based on feedback	Final
1.1	17/11/2017	Apply remarks for Magnus	Final

5 Indicate Main Author(s) with an "X" in this column.
6 Deliverable leader according to FPP, role definition in PCA.
7 Person(s) from contributing partners for the deliverable, expected contributing partners stated in FPP.
8Typically person(s) with appropriate expertise to assess deliverable structure and quality.
9Status = "Draft", "In Review", "Released".

CONTENTS

CONTENTS 3
Abbreviations 3
1 Overview 4
1.1 Introduction 4
1.2 xtUML and BridgePoint 4
1.3 FMI and FMU 4
1.4 xtUML transformation to FMU 4
2 ARChITECTURE 6
1.5 Environment 6
1.6 Models 6
1.7 Project structure 9
User GuIDE 15
2.1 Overview 15
2.2 Environment setup 15
2.3 Usage 25
2.4 Description of the used xtUML model 32
3 SUMMARY 33

ABBREVIATIONS

List of abbreviations/acronyms used in document:

Abbreviation	Definition
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
M&S	Modelling and Simulation
N/A	Not Applicable
SotA	State of the Art
TBD	To Be Defined

1 OVERVIEW

1.1 Introduction

The goal of this sub-project is provide support for transforming xtUML models to standardized FMU which has to be capable of co-simulation with other FMUs.

The document is structured as follows. Section 1 provides some background information on the used technologies and standards and explains the basics of our solution of the xtUML to FMU transformation, Section 2 presents the architecture of our implementation, including the used technologies, the models we work on and explains each steps of the transformation, Section 3 provides a user guide that explains how to use the software from the installation phase, and provides an example and Section 4 concludes this document.

1.2 xtUML and BridgePoint

Executable and translatable UML (xtUML) accelerates the development of real-time, embedded and technical software systems. Proven and well defined, xtUML is a fully automated methodology utilizing a UML notation. The most common tool for xtUML models is called BridgePoint (BP).

For more information of xtUML and BridgePoint please visit this website.

1.3 FMI and FMU

The Functional Mock-up Interface (FMI) is a tool independent approach for model exchange (ME) and co-simulation (CS), and on the way to become the industry standard for exchange of models and cross-company collaboration. Its main purpose is to share and reuse simulation artifacts among a wide variety of tools and environments, by putting the model specifications into a simple compressed file called Functional Mockup Unit (FMU). The FMU contains a model description in XML format, source written in C and/or binaries ready to run and optional components such as documentation, model logo, etc.

For more information of FMI please visit this website.

1.4 xtUML transformation to FMU

The FMI standard was created with continuous-time models in mind, which makes this transformation a non-trivial task.

The solution of the transformation is generate wrapper in C code to the executable code, which provides the connection interface of xtUML and the interface of FMI standard. The structure of this wrapper is depicted in Figure 1. The inner components of such an FMU wrapper can be grouped into three parts:

1. The input-wrapper which translates incoming FMI variables into actions to be performed on the underlying UML model.

2. The output-wrapper which translates certain actions and changes of the UML model into outgoing FMI variables.
3. The control-wrapper which can handle the remaining FMI functions (i.e. lifecycle and simulation control functions). This component is responsible for scheduling the execution of the UML application and possibly modifying the states of the other wrapper components (e.g. resetting them between simulation steps).

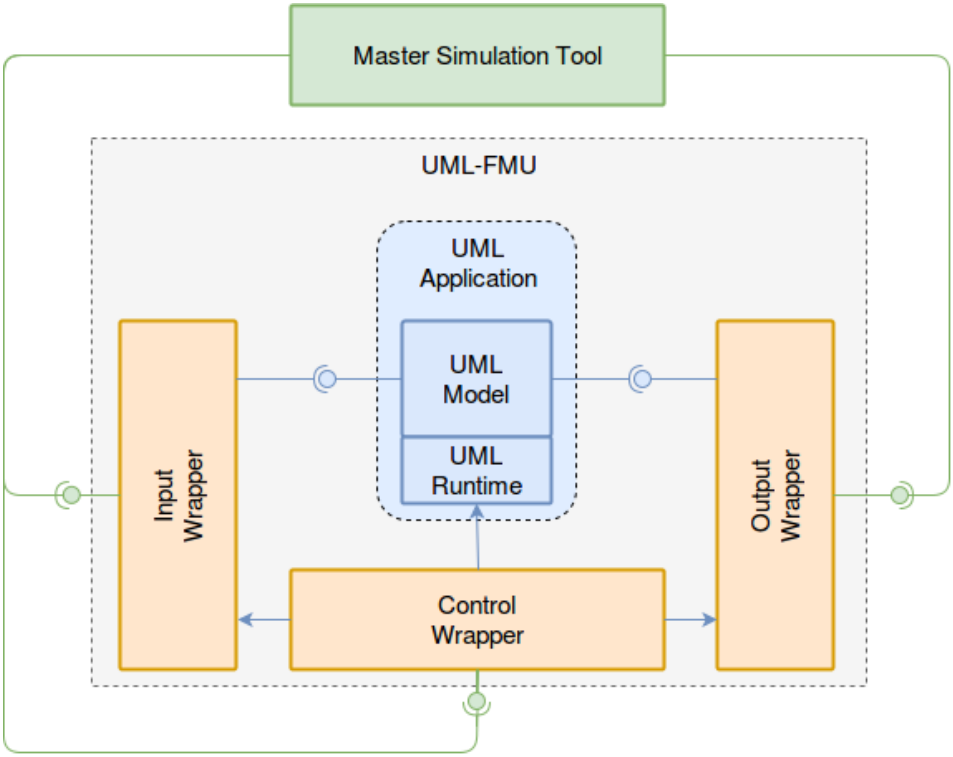


Figure 1 FMU wrapper

Generating this wrapper is refined to 5 sub-steps, which is can be found in Figure 2 signed with red circle. The detailing of this steps described in the Architecture section.

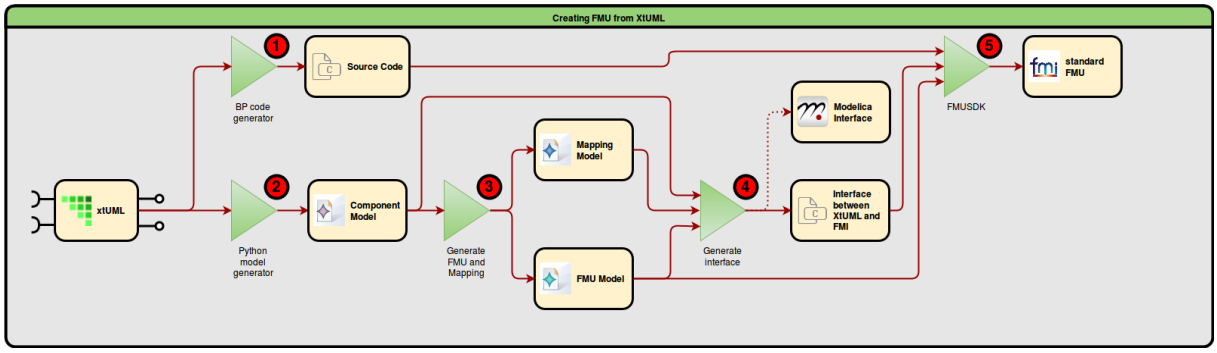


Figure 2. xtUML to FMU

2 ARCHITECTURE

1.5 Environment

1.5.1 Eclipse

Eclipse is a Java-based IDE with an extensible plug-in system, that allows the user to customize the environment as well as to develop and install their own plug-ins. For more information on Eclipse please visit the Eclipse website.

Many applications are based on Eclipse, including BridgePoint. This software is also based on an Eclipse.

1.5.2 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is also based on Eclipse. It provides a modeling environment, that allows the user to design class diagram-like metamodels and a code generator that generates Java structures (classes, interfaces, etc.) from the model. This improves the simplicity of creating complex data structures. For more information on Eclipse Modeling Framework please visit the EMF website.

This software uses EMF to represent xtUML models and transform them to FMUs.

1.5.3 BridgePoint

In BridgePoint you can create and edit xtUML models. BridgePoint has a code generator that creates executable code from the xtUML models to many languages. In this project we used and modified the C code generator of BridgePoint.

For more information of xtUML and BridgePoint please visit this website.

1.5.4 FMU SDK

The FMU SDK is a free software development kit provided by QTronic to demonstrate basic use of Functional Mockup Units (FMUs) for model exchange and for co-simulation as defined by the FMI Specification version 2.0 and 1.0. FMU SDK can also serve as a starting point for developing applications that create or process FMUs.

1.6 Models

1.6.1 FMU model

Our FMU metamodel is an EMF model that corresponds to the FMI standard. This is a big model, however, the important information about it is that at its lowest level it has the primitive variable *ScalarVariable* from which the FMU model is constructed.

For more information, please visit this website.

1.6.2 xtUML model

Project structure of xtUML project generally looks like next snippet.

```
RootPackage          //Root of Project
|
+-- component1       //Component is the main
|   |                //of the logical model
|   |                //Cardinality of Component is zero or more
|   +-- componentPackage
|       |
|       +-- classPackage
|       |   |
|       |   +-- class1
|       |       |
|       |       +-- instanceStateMachine
|       |       |
|       |       +-- classStateMachines
|       |
|       +-- port1         //Implementation of an interface
|       |                 //At this point implemented what
|       |                 //should happen at communication
|       +-- portK
|
+-- interface1            //Interfaces to make inner
|                         //or outer connections
|
+-- interfaceN
```

Components can use interfaces as both provided interfaces and required interfaces, which will appear as ports. Over these ports the component can connect to other *xtUML* or *OpenModelica* models. In xtUML interfaces can contain both signals and operations, but this project does not support the latter. However, signals can contain parameters, that are supported.

Because of this, the transformation from xtUML to FMU is not straightforward. In xtUML there are signals with zero or more parameters while an FMU only has the *ScalarVariable*. The other problem is that signals can be triggered more than once in a simulation step.

Our solution for these problems is demonstrated on the following example.

The signal1(Real param1, integer param2) should be mapped to the following variables:

- signal1Count [integer]
- signal1_param1_1 [real]

- signal1_param1_2 [real]
- …
- signal1_param1_N [real]
- signal1_param2_1 [integer]
- signal1_param2_2 [integer]
- …
- signal1_param2_N [integer]

1.6.3 xtComponent model

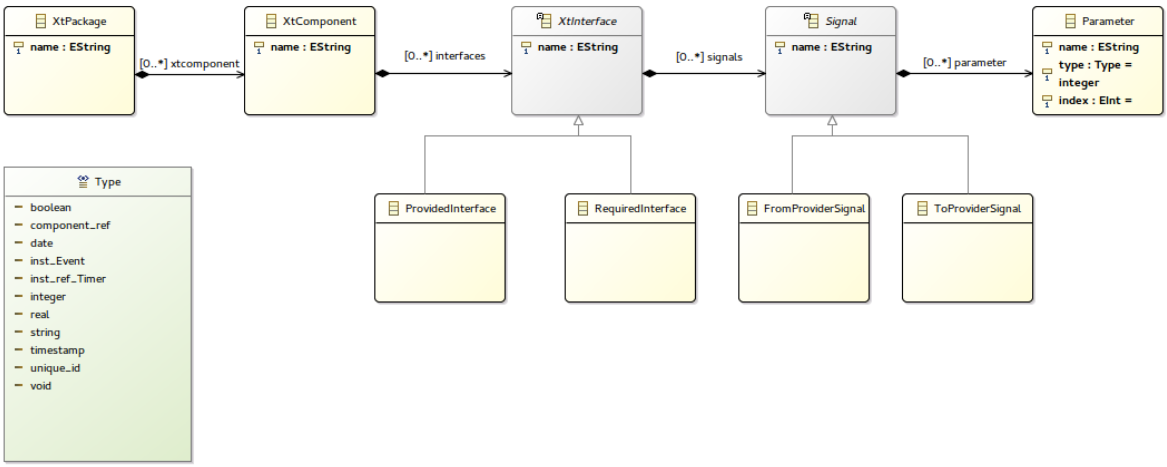


Figure 3. xtComponent model

An xtComponent model is the representation of an *xtUML* model under EMF, generated by the *emfGenFromXtUML.py* python script.

The root element of this model is the XtPackage element. The value of its name attribute is derived from the *xtUML* project's name.

The XtPackage has XtComponent elements, that have XtInterface elements. XtInterfaces are either provided (ProvidedInterface) or required (RequiredInterface). XtInterfaces have Signal elements, that are also of two types: FromProvider or ToProvider. Signals may have Parameter elements, that have name, type and index attributes. The index is required, because later at the *C code generation* phase the order of the parameters becomes important.

The data flow direction is defined by the XtInterface and Signal pair:

	ProvidedInterface	RequiredInterface
FromProviderSignal	Output	Input

	ProvidedInterface	RequiredInterface
ToProviderSignal	Input	Output

1.6.4 Mapping model

The mapping model maps between a *xtComponent* model and *FMU* model. The root element of this model is the Mapping element. This element has references to an xtComponent model and to an FMU model.

It has Link elements and a maxIndex attribute representing the number of links belong to one Parameter of the XtComponent. This attribute determines how many event can be handled at co-simulation.

A Link is either a ParameterLink or a SignalLink. Every Link refers to an FMU's parameter: a SignalLink refers to a Signal and a ParameterLink refers to a Parameter of the XtComponent, such that for each Parameter there are exactly maxIndex referring ParameterLink elements, indexed from 0 to maxIndex-1.

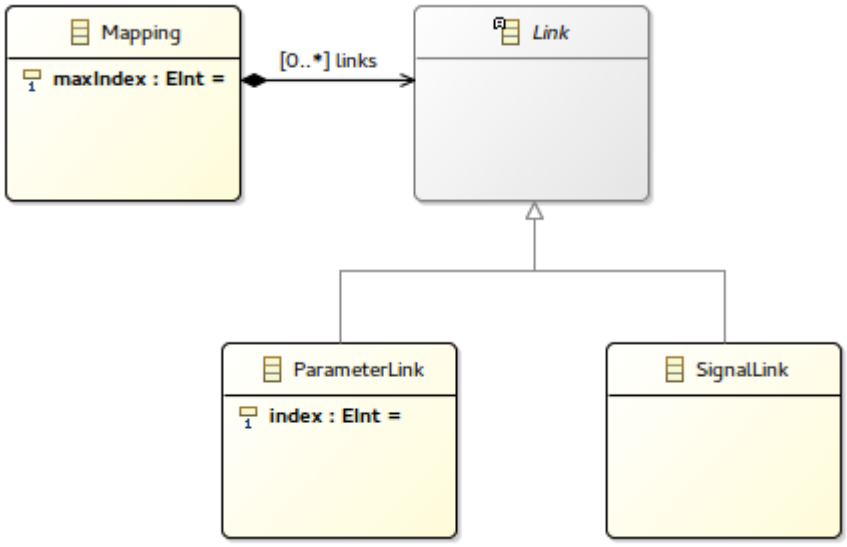


Figure 4. Mapping model

1.7 Project structure

This section presents the five steps that - as it was mentioned in the Overview section in Figure 2 - make up the transformation.

Table 1. Description of steps

Steps		Derive	Environment	Responsible Plugin
BP code generator	1	BP with modification	BP	plugins/com.incquerylabs.opencps.bp.mc.c.source/
Python code generator	2	IncQueryLabs (using BP's python libraries)	Terminal	simulator/pyxtuml/emfGenFromXtUML.py
Generate FMU and Mapping models	3	IncQueryLabs	Eclipse	plugins/com.incquerylabs.opencps.XtUML.mapping.transformation/
Generate interface	4	IncQueryLabs	Eclipse	plugins/com.incquerylabs.opencps.mapping.transformation/
FMUSDK interface	5	FMUSDK with modification	Terminal	simulator/fmuCreator/Makefile

1.7.1 BP code generator

The first step is to generate the C code, witch implements the behavior of the model.

A generator is already implemented in BP to generate an application in C code. In our implementation it modified to generate C code that can be wrappered to FMU. For example the generated code does not simulate time, but relies on outer sources to provide the simulation time.

1.7.2 Python code generator

The next step is to generate an EMF model, which is the representation of the xtUML model.

This step allows us to move the xtUML model to EMF environment so that other steps can use it later.

Table 2. Matching between xtUML and xtComponent

xtUML element	xtComponent element
rootPackage	XtPackage
component	XtComponent
port	XtInterface
signal	Signal
parameter	Parameter

1.7.3 Generate FMU and Mapping models

The next step is to generate an FMU model and a Mapping model, which has the connections between the Component model and FMU model.

This solves the problem mentioned in Section 2.2.2. The matching between xtComponent model and FMU model is listed in the following table.

Table 3. Matching between xtUML and xtComponent

xtComponent element	FMU element
XtPackage	---
XtComponent	---
XtInterface	---

Table 3. Matching between xtUML and xtComponent

xtComponent element	FMU element
Signal	ScalarVariable [Integer]
Parameter [type X]	ScalarVariable [type X]*

At this moment the cardinality of the ScalarVariable at the Parameter-ScalarVariable pair is fixed to 3.

1.7.4 Generate interface

The next step is to generate the wrapper C code and provide a modelica model.

1.7.4.1 Wrapper C code

First, the wrapper C code is generated that creates the connections between the source code and standard FMI functions. For example the param1_1 *ScalarVariable* derived from *signal1(Real param1, integer param2)* output signal, generates an *fmi2GetReal* method.

1.7.4.2 Modelica interface model (optional, not necessary for making FMU)

The Modelica model is an optional help to make data transformation automatically at Modelica. The Modelica can handle only one signal (as data) at one simulation step, hence the user has to define how to aggregate the parameters of the signal. There are already 6 options to that: *minimum, maximum, first, last, sum* and *avg*. The default method is *last*.

The output of the model can be multiple data, because if the receiver is xtUML-FMU, it can handle more signals and if the receiver is Modelica-FMU, it can chose what to do as described above. Therefor this interface Model collects how many times the output signal was triggered in one simulation step.

The following example shows the modelica model generated from a simple xtUML model. The xtUML has one component within one provided interface within 2 signals. The first signal's message direction is *from provider*, and second signal's message direction is *to provider*. The first signal is an output in the xtUML model, therefore it will appear as input at Modelica. For ease of understanding the first signal was named *InputSignal* and second *OutputSignal*. *InputSignal* has an Integer and a Real parameter and OutputSignal has a Boolean and an Integer parameter.

The generated transformed modelica model can be seen in Figure 5. In the picture you can see the signals appear as Boolean input and output. The inner logic counts how many times the OutputSignal is triggered and caches the parameters of the signal those moments.

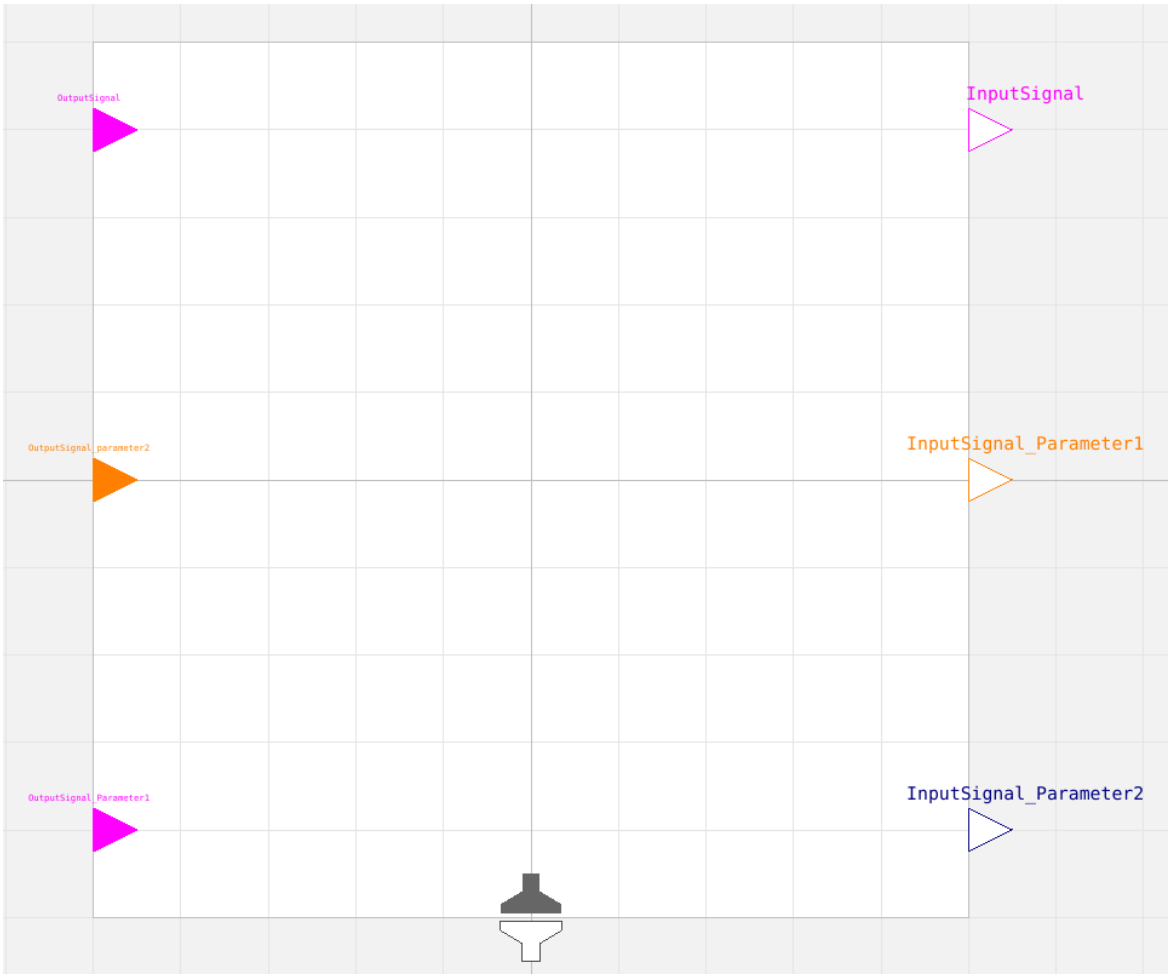


Figure 5. The generated transformer modelica model

The data transform interface is shown in Figure 6.

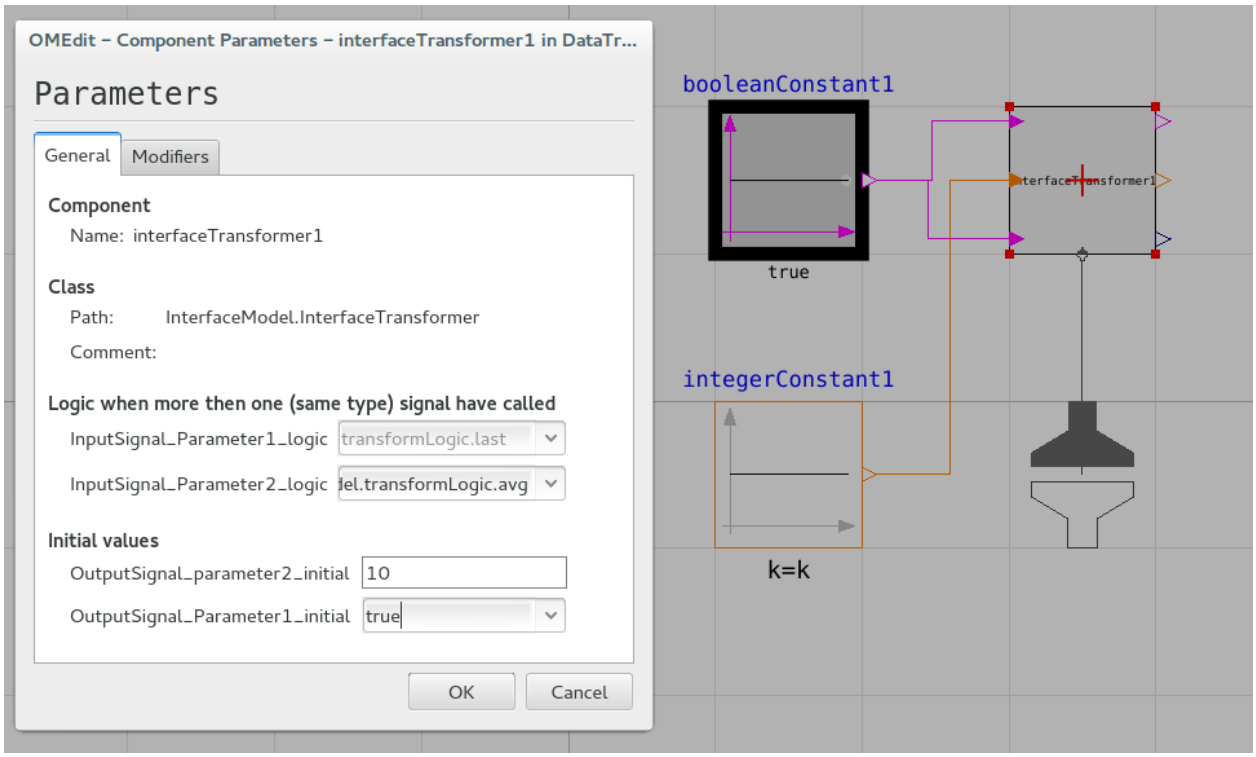


Figure 6. Data transform interface

You can find this example in repository at models/model_src/Modelica_interface.

1.7.5 FMU SDK interface

The final step is to generate a standard FMU. At this point all sources are available to make an FMU, and all that is left to do is to compile them together. This can be performed by FMU SDK.

USER GUIDE

2.1 Overview

2.2 Environment setup

2.2.1 Installation

The project environment requires Java to be installed. It can be downloaded from Oracle's website.

There is no update-site yet, where an Eclipse with the required plugins can be downloaded. Instead, this guide uses the Eclipse Installer, which can be downloaded from https://www.eclipse.org/downloads/.

The project also relies on Python to be installed. It can be downloaded from this website. The project work with both main version of Python (2.x and 3.x).

2.2.2 Eclipse setup

2.2.2.1 Downloading Eclipse

1. Run the Eclipse Installer in *advanced mode*. This can be achieved by clicking on the three horizontal lines in the upper right corner and selecting *advanced mode.*
2. The first task is to choose the Eclipse distribution to install.
    a. Make sure everything is up-to-date, by clicking on the *Install available updates* button, next to the *Back* and *Next* buttons on the bottom.
    b. From the list of products, choose *Eclipse Modeling Tools*.
    c. In the lower part of the screen select product version `Oxygen`, and make sure the settings are correct.
    d. Finally, hit *Next*.

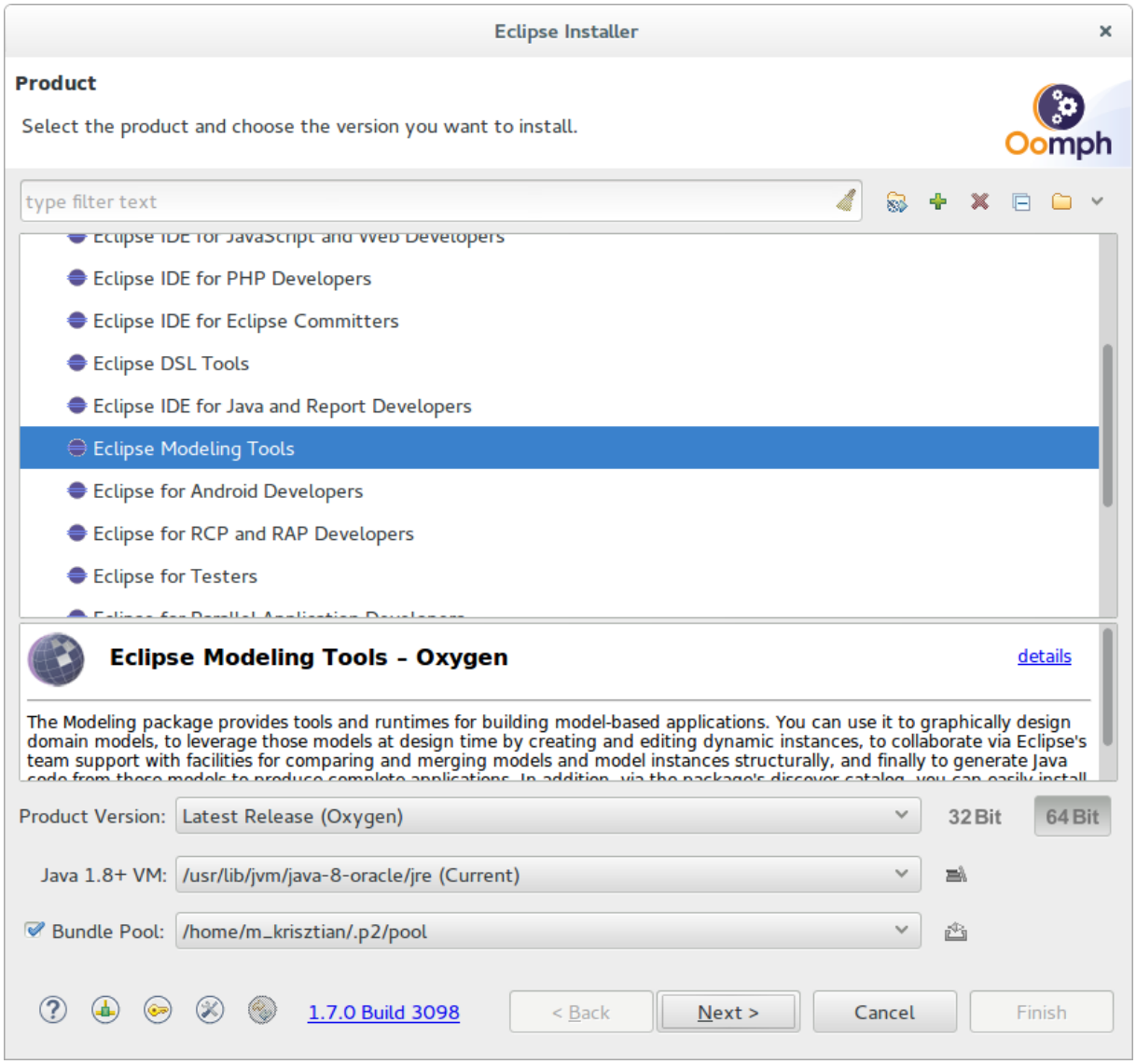


Figure 7. Eclipse Installer 1

3. The next task is to import the eclipse projects you will use.

   a. Click the green + button in the upper right corner.

   b. In the appearing window choose GitHub projects from the drop-down menu.

   c. The required *Resource URI* can be retrieved the following way.

      i. Make sure you are logged into GitHub and have an access to the repository.

      ii. Open TODO make public link link and copy the URL from the browser (it will have your token at the end).

      iii. Copy the link and paste it to the Eclipse Installer.

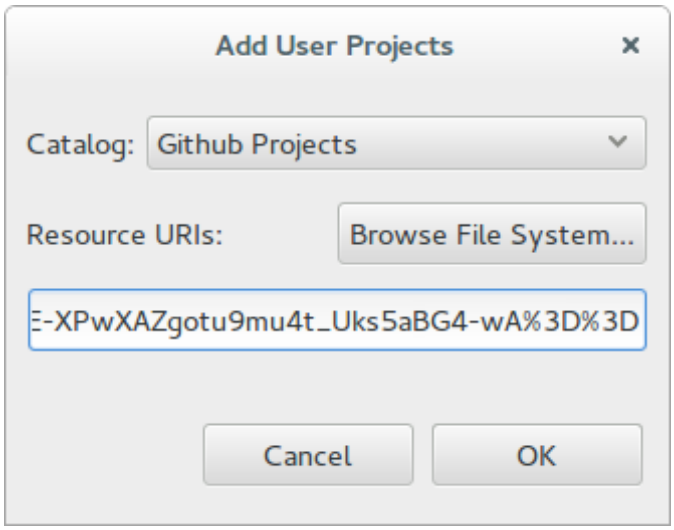


Figure 8. Eclipse Installer 2

d. Hit *OK*.

e. You'll see, that a new folder, named <*User*>, has been created in the GitHub category, which has a project named *opencps-hungary*. Select it.

f. Finally, hit *Next*.

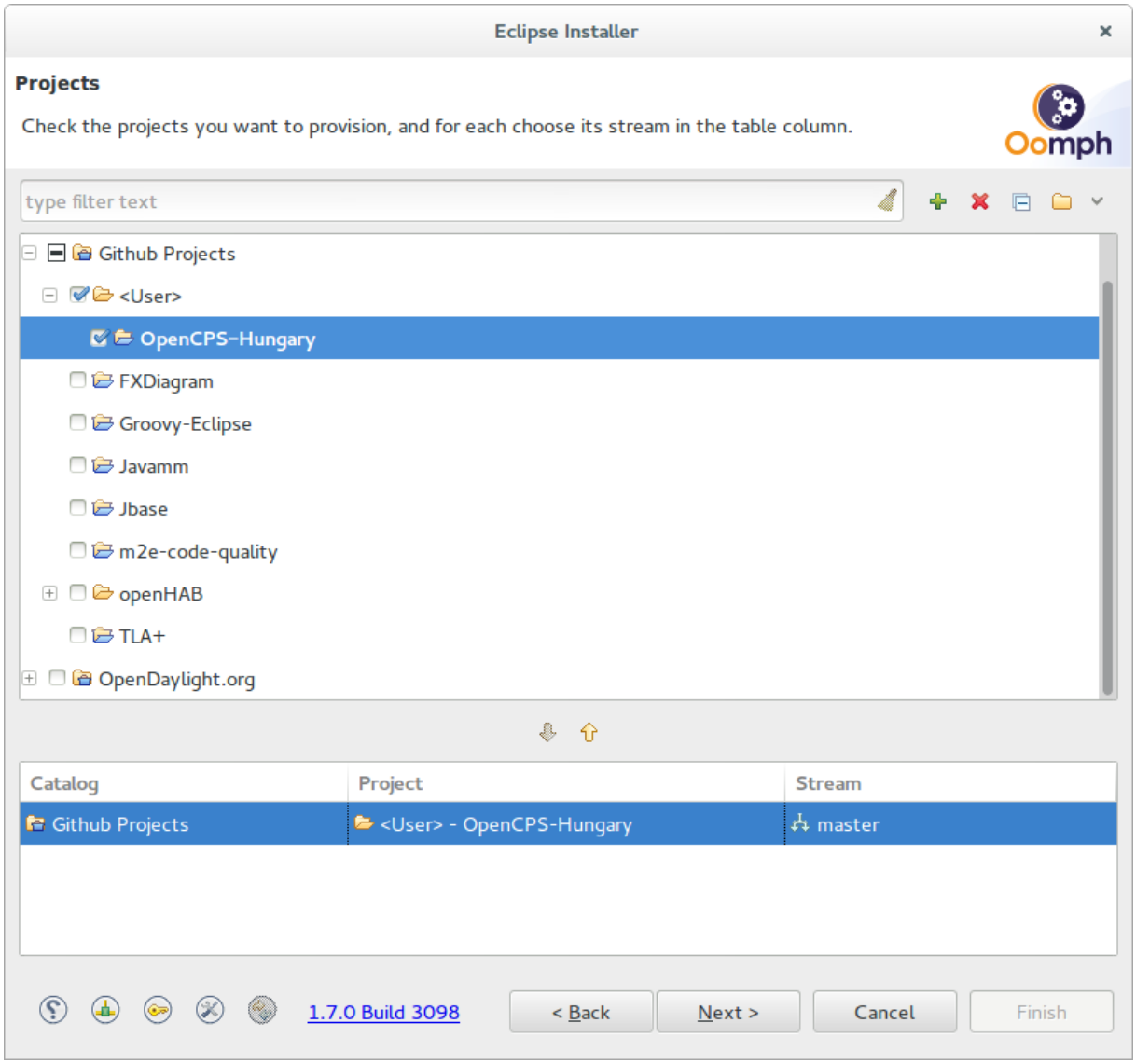


Figure 9. Eclipse Installer 3

4. The next task is to select where you want to keep the resources for Eclipse. If you select *Show all variables* you can set the paths of more resources in a more advanced way, otherwise the defaults will be set.

    a. Check *Show all variables*

    b. Choose a folder to install Eclipse.

    c. Set a *workspace* folder. Eclipse uses workspaces to organize the Eclipse projects you are working on, many settings for this application will be stored in the selected folder, however the projects themselves will be imported from the Git clone location.

    d. Browse your folders and set the one where you cloned the GitHub repository of this project.

e. Set the target platform to *Oxygen*.

f. Finally, hit *Next*.

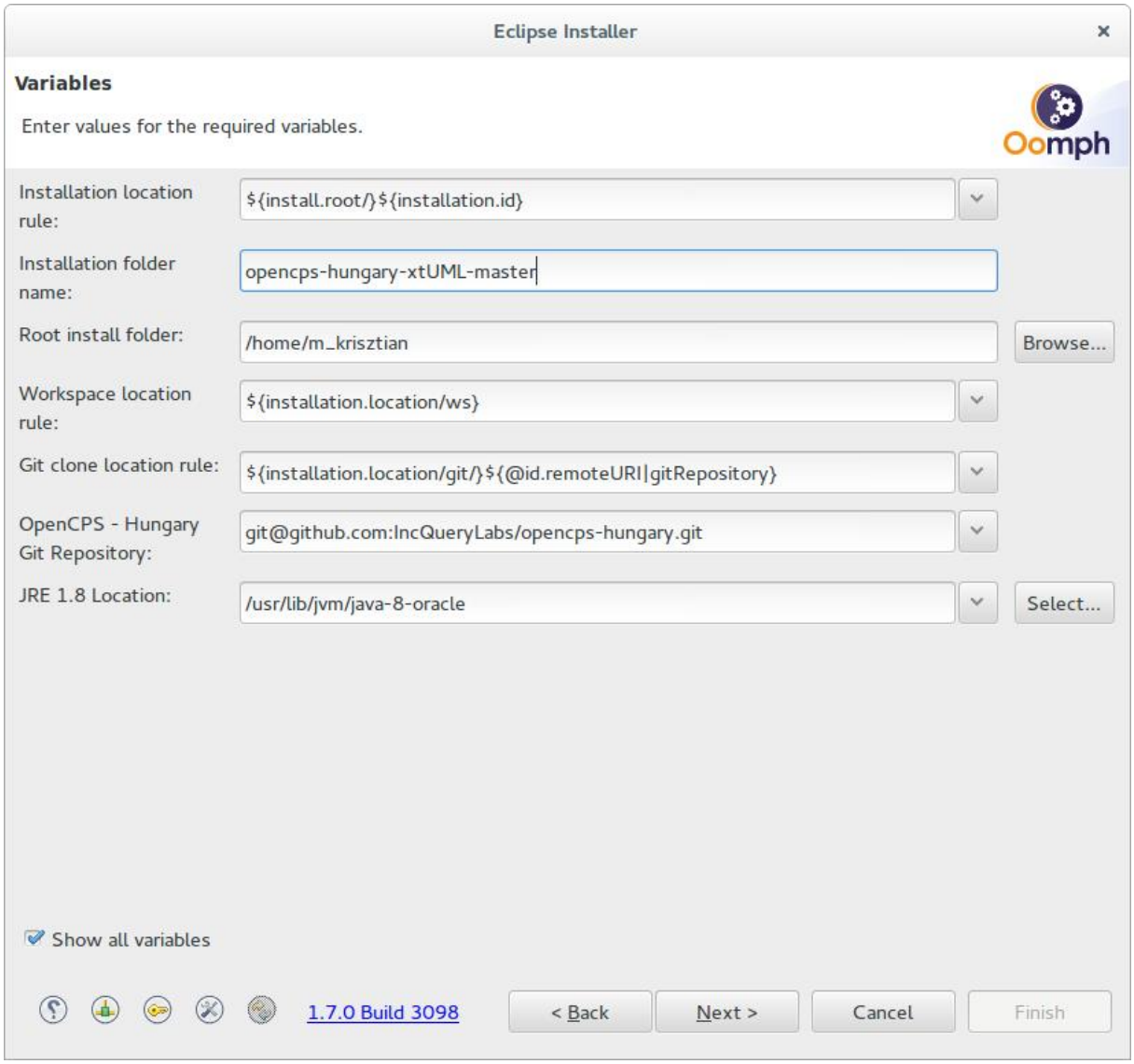


Figure 10. Eclipse Installer 4

5. In the appearing window, click *Finish* and the setup process begins. It migtht take up a few minutes, but once it's finished, it will open the newly installed Eclipse Oxygen automatically.

2.2.2.2 Resolving compile errors

After launching, Eclipse will need to take some time and import the projects. You can see the progress in the down right corner. Once it's done, there will be a lot of errors in the workspace (as you can see in the *Model Explorer* once you close the *Welcome* tab), but the following steps will resolve them.

1. First of all, you have to generate the model code from each genmodel (they are located in the model folder in each project), which are in the following projects:
    - **com.incquerylabs.opencps.fmu.from.xsd**
    - **com.incquerylabs.opencps.simulation.model**
    - **com.incquerylabs.opencps.xtComponent.model**
    - **com.incquerylabs.opencps.xtuml.mapping.model**

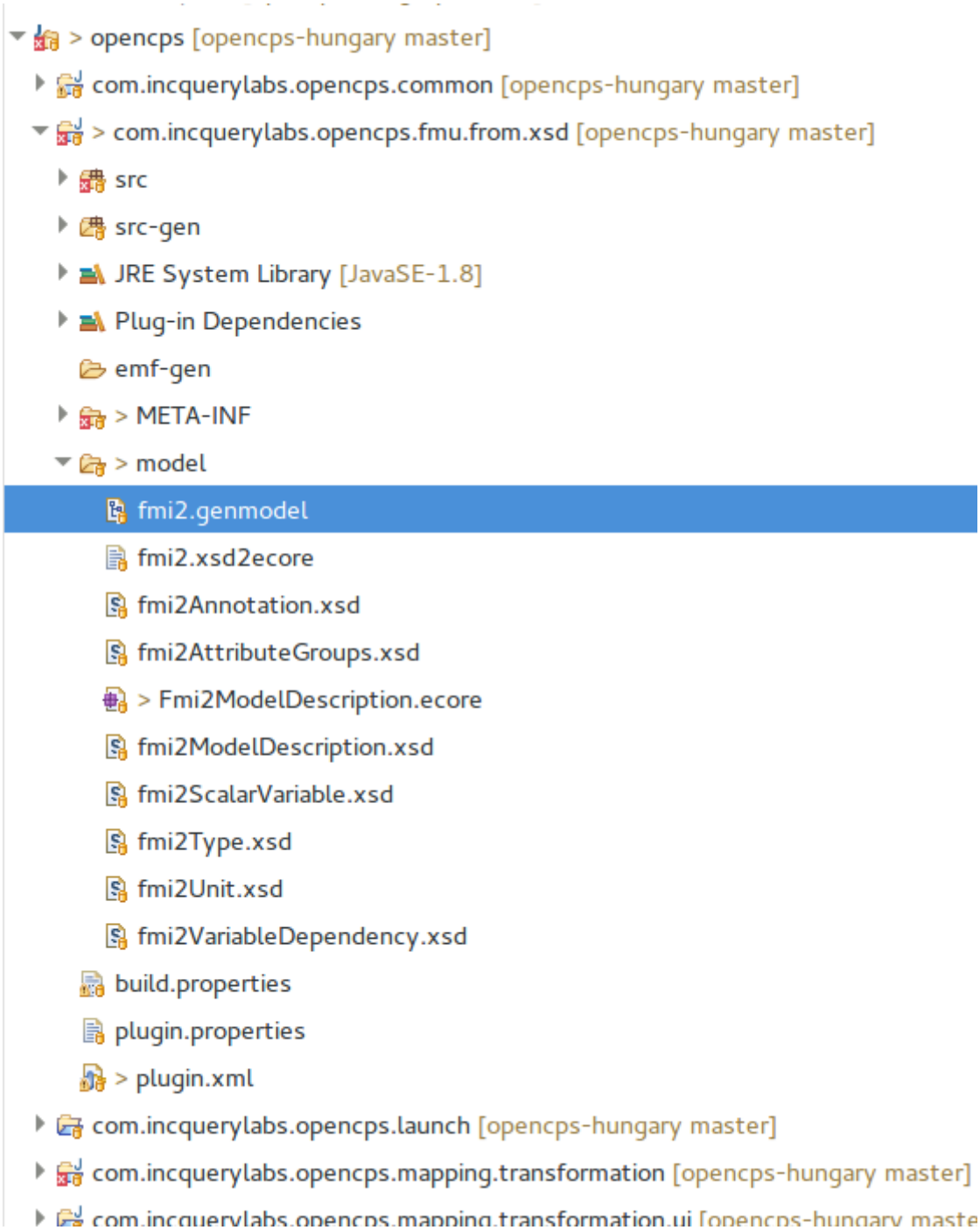


Figure 11. Genmodel in the Project

2. To generate model code, open a *.genmodel* file, right click on the root element, and click *Generate Model Code*. Wait until Eclipse builds the workspace and then do the same with *Generate Edit Code* and *Generate Editor Code*.

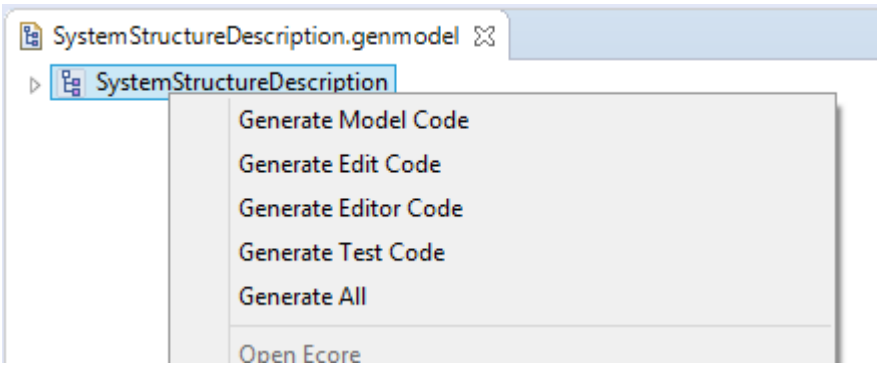


Figure 12. Generating from genmodel

A more simple solution is to simply click *Generate all*, and remove the generated test-related projects.

3. After you've done that, *clean* all projects.
4. That's it, you are all set now, the workspace should be errorless.

2.2.3 xtUML setup own code generator

First you need to import a project into BridgePoint. It is under opencps-hungary repository in plugins folder named com.incquerylabs.opencps.bp.mc.c.source.

Now, right click on your project witch contain the xtUML model and select `Properties`. Left side of the current open window select `Builders`. To create new builder press the `New...` button. Select `Program`, OK.

The build name could be `IQL_Model_Compiler` or what ever you want.

Location would be the xtumlmc_build.exe of the previously imported project's. Something like this:

```
${workspace_loc:/com.incquerylabs.opencps.bp.mc.c.source/mc3020/bin/xtumlmc_build.exe}
```

Working Directory would be the gen folder of the project:

```
${build_project}/gen
```

Arguments have to be the following:

```
-home "${workspace_loc:/com.incquerylabs.opencps.bp.mc.c.source/}" -l3s -e -d code_generation -O ../../src/
```

Click `OK`

Lastly, disable the original `Model Compiler`.

The result is illustrated in Figure 13.

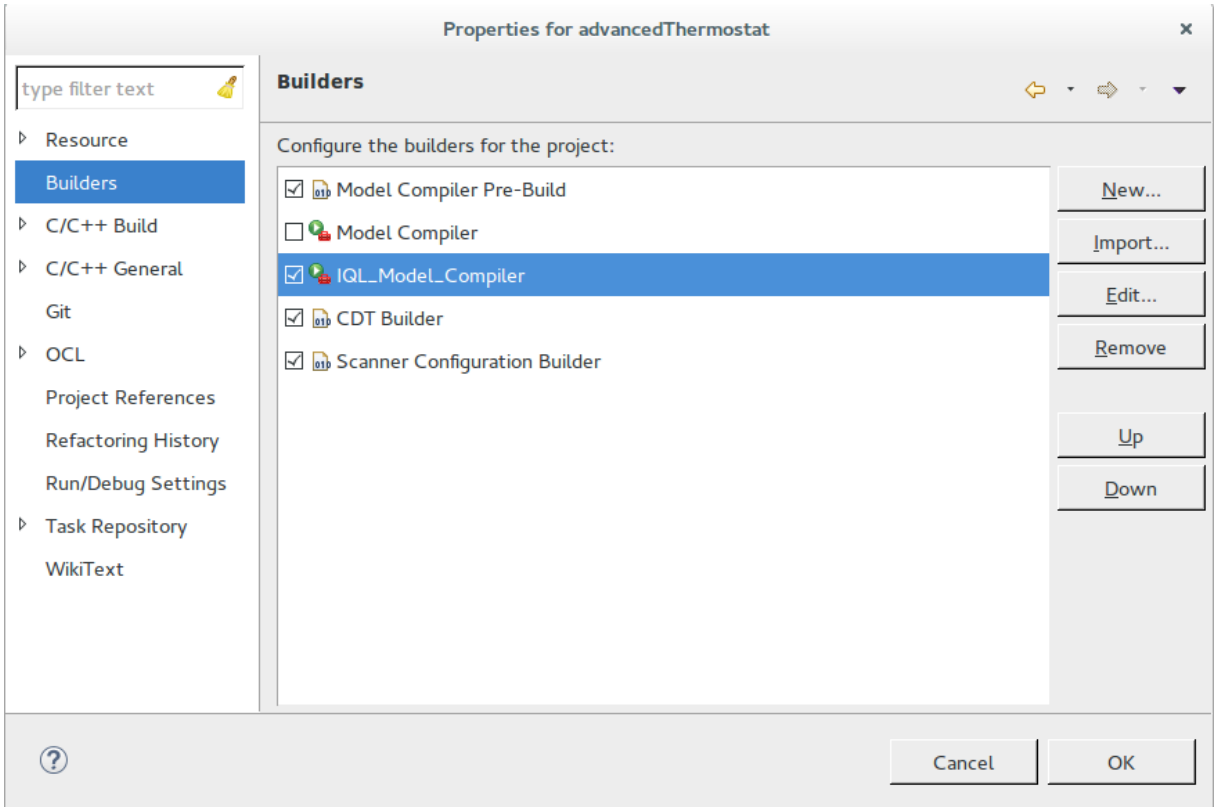


Figure 13. Build for xtUML project

2.2.4 Install python dependencies

2.2.4.1 In Linux

Download ply from this website, extract and install with the next commands.

```
$ cd ply-3.10 #The version may be different
$ sudo python setup.py install
```

Fetch the source code from github and install it manually:

```
$ git clone https://github.com/xtuml/pyxtuml.git
$ cd pyxtuml
$ sudo python setup.py install
```

2.2.4.2 In Windows

Add the path of Python to environmental variables is not necessary, but make more conforuble the useage of it. So we recommend to see this page. If you do not want to do that, you can write `"C:\Program Files (x86)\Python\python.exe"` insted `python` or something like this, depend where the python is installed.

Download ply from this website, extract and install with the next commands.

```
> cd ply-3.10 #The version may be different
```

```
> python setup.py install
```

Fetch the source code from github and install it manually:

```
> git clone https://github.com/xtuml/pyxtuml.git
> cd pyxtuml
> python setup.py install
```

2.2.5 Install Cygwin in Linux

This is a large collection of GNU and Open Source tools which provide functionality similar to a Linux distribution on Windows.

It is needed to be able to execute Makefiles in Windows.

It can be downloaded from here.

1. At installation you have to add all packages that are needed to make.
    a. At Select packages page choose view to `Category`.
    b. Search the word `make`.
    c. Choose install option on the right side of Devel.
    d. Next

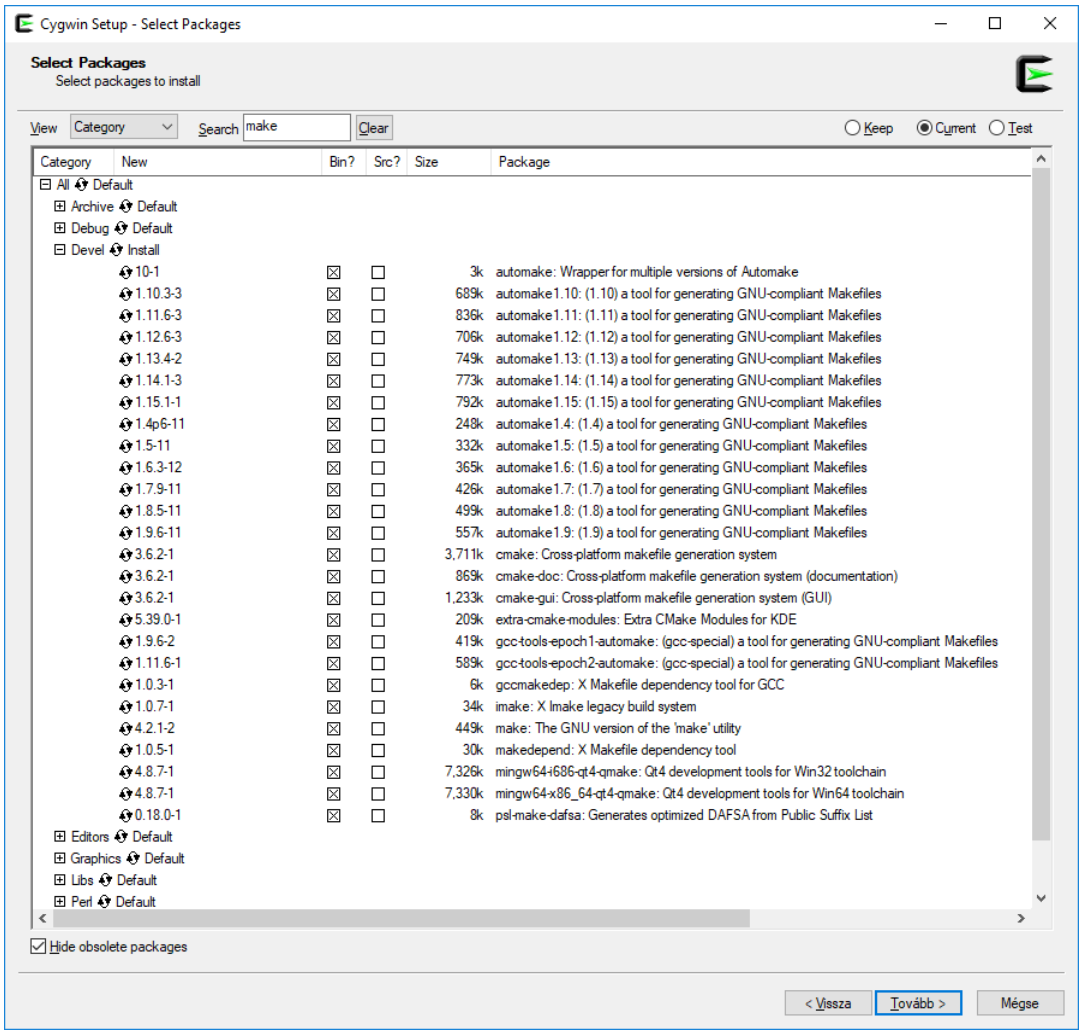


Figure 14. Cygwin packages

2.3 Usage

2.3.1 Generation of xtUML executable code (BridgePoint)

Generating the xtUML C code is done by building the xtUML projects (use `Project\Clean…`)

Move the generated source files into the correct location:

- Create a folder inside the directory `opencps-hungary/simulator/fmusdk2/fmu20/src/models`. It has to be named the same as the project in BridgePoint.
- Copy the generated code to this folder.

The result is generated into src folder, you can see it using Java perspective. Your project should look as illustrated in Figure 15.

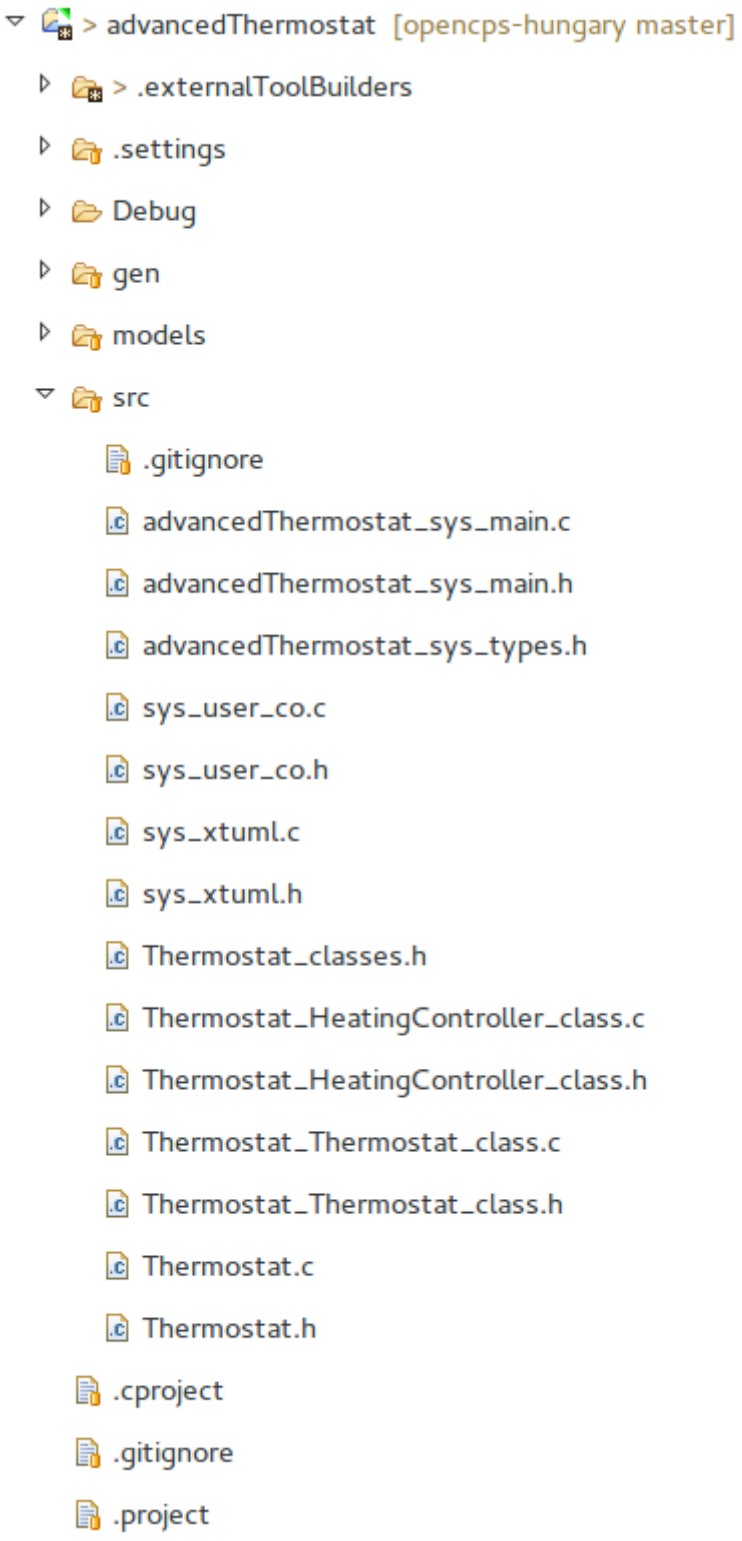


Figure 15. xtUML project after executable code generation

2.3.2 Generation of EMF model describing the xtUML interfaces (Python script)

- Merge all xtUML files

- Java perspective → in Package Explorer right click → Export… → xtUML Model

- Generate EMF model with the python script located at `opencps-hungary/simulator/pyxtuml` from the exported xtUML
  - From a terminal:
    - Command syntax: `python emfGenFromXtUML.py [File location]/[xtUML file] [Target location]`
    - The script has 2 arguments. The first is the path of the xtUML file with file name, the second is the target folder where the EMF file will be created.
  - From VS Code using a launch configuration:
    - Be sure the relative path in the `program` parameter is correct
    - The arguments are the same as in the terminal. **You probably have to overwrite it.**

```
{
   "version": "0.2.0",
   "configurations": [
      {
         "name": "Python",
         "type": "python",
         "pythonPath":"${config.python.pythonPath}",
         "request": "launch",
         "stopOnEntry": true,
         "console": "none",
         "program":
"${workspaceRoot}/pyxtuml/emfGenFromXtUML.py",
         "cwd": "${workspaceRoot}",

         "args":                    ["/home/course/git/opencps-
hungary/simulator/fmus/xtThermostat/thermostat.xtuml",
"/home/course/git/opencps-hungary/models/model_src/"],
         "debugOptions": [
            "WaitOnAbnormalExit",
            "WaitOnNormalExit",
            "RedirectOutput"
         ],
         "env": {"name":"value"}
      }
   ]
}
```

The outcome of this section is a xtComponent model and after opened in runtime-eclipse it looks as illustrated in Figure 16.

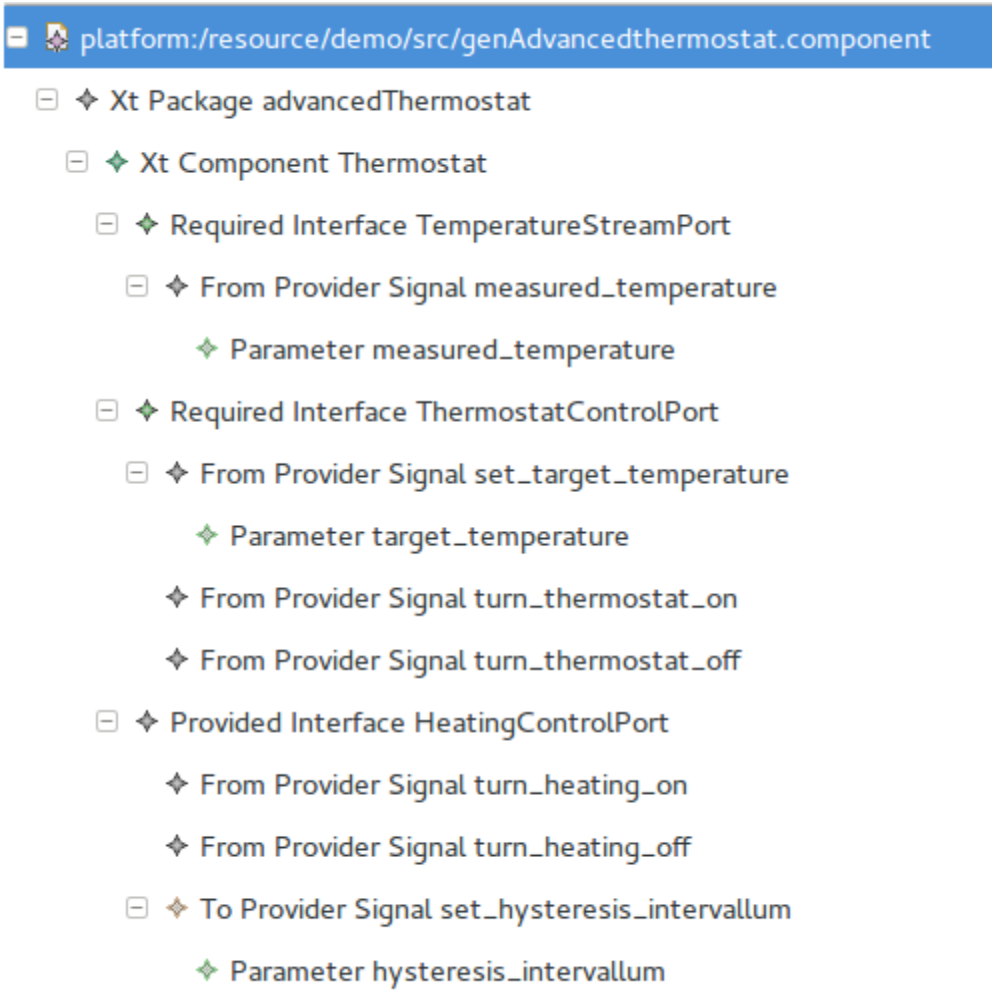


Figure 16. xtComponent Example

2.3.3 Generation of Mapping model and FMU model (Eclipse)

- Open the previously generated component model in runtime Eclipse.
- Right click on the `XtPackage` element and select `Generate Mapping Model`.
- Move xml file and Makefile where to the xtUML executable code is.

The outcome of this section is an fmi2modeldescription model, a mapping model and a Makefile. The mapping model maps each of the component model's signals and parameters to a different fmu model's scalarVariable.

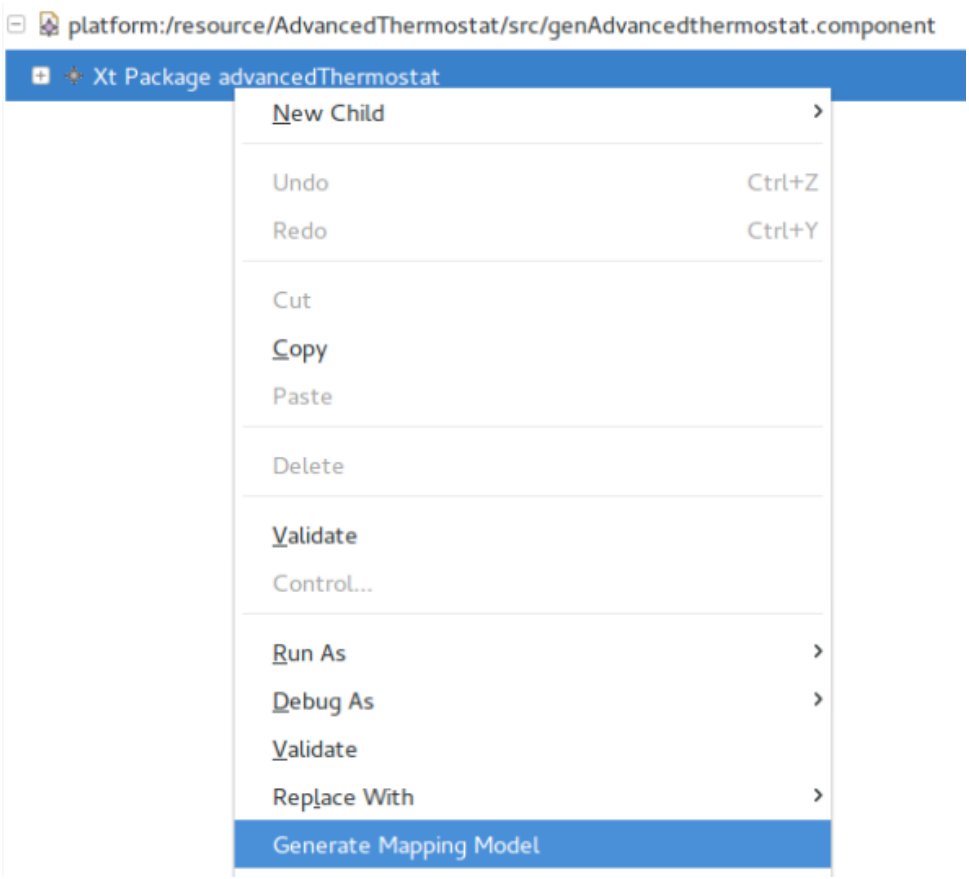


Figure 17. Generate Mapping model

2.3.4 Generation of Interface between xtUML and FMU (Eclipse)

- Open the previously generated mapping model in runtime Eclipse.
- Right click on the `Mapping` element and select `Generate Interface between XtUML and FMI`.
- Move these files to where the xtUML executable code have moved.

The outcome of this section are two C files, one with header, and a modelica model. Except for the modelica model, these files provide wrapping xtUML model that works like other FMUs from outside.

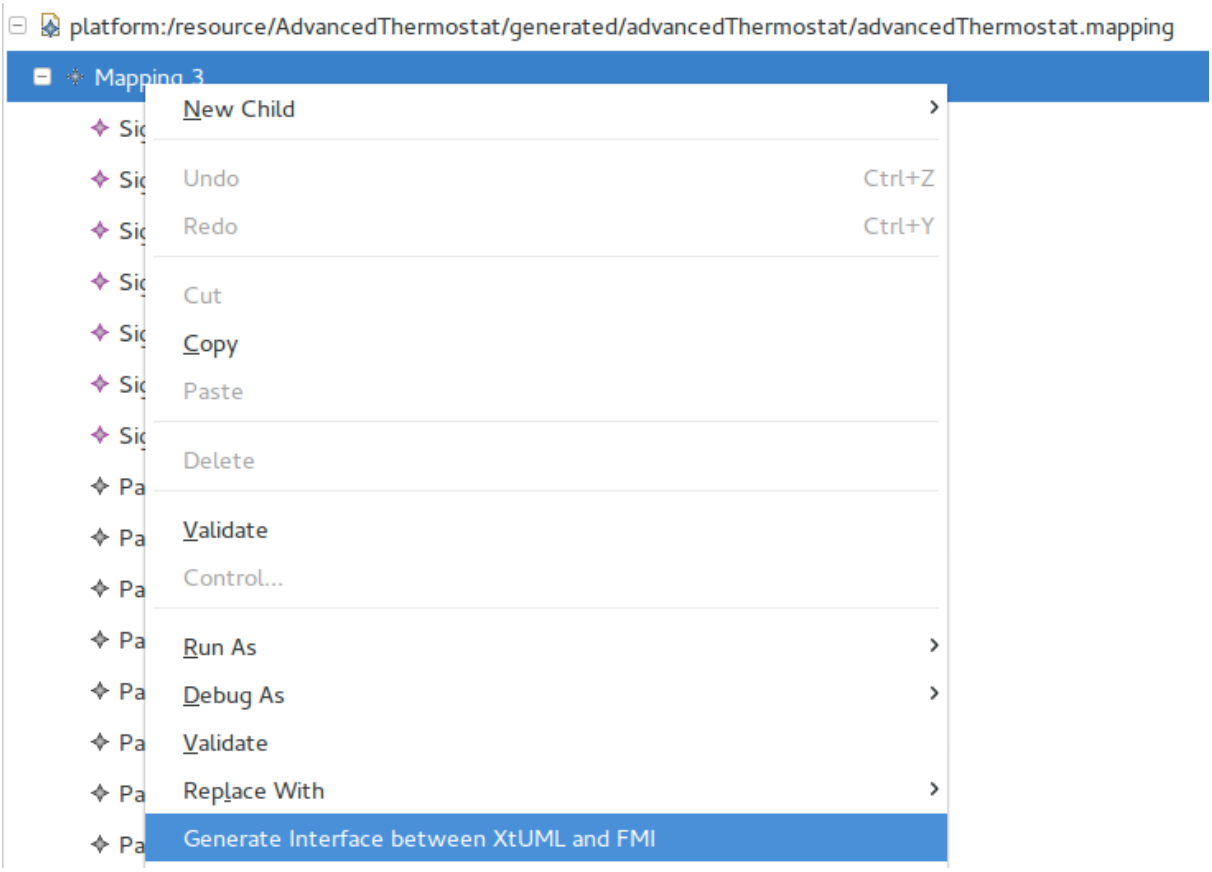


Figure 18. Generate interface between xtUML and FMU

2.3.5 Generation of FMU from xtUML (FMUSDK)

- Make sure the three types of resources are in the correct location (`opencps-hungary/simulator/fmusdk2/fmu20/src/models/[systemName]`) from the first, third and fourth section.
- Open a terminal and navigate to `opencps-hungary/simulator/fmuCreator`.
  - **In Windows it is not enough to open a simple terminal, but Cygwin terminal.**
- Run Makefile
  - Command syntax: `make -i -k model=[systemName]`
  - You have to change [systemName] to the name of the folder where resources are.

The outcome of this section is an FMU. The Makefile copies this FMU to the other FMUs (`opencps-hungary/simulator/fmus`)

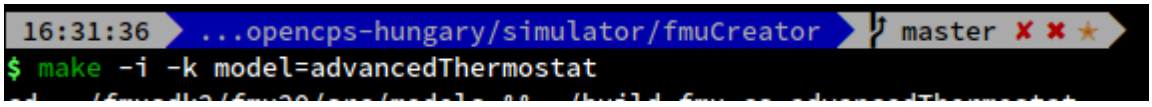


Figure 19. Execute fmuSDK

2.3.6 Usage of the modelica interface model

Open OMEdit. Open the generated modelica interface model with `File → Open Model/Library File(s)`, search the model and `Open`. This Modelica model is a package, which contents interfaces.

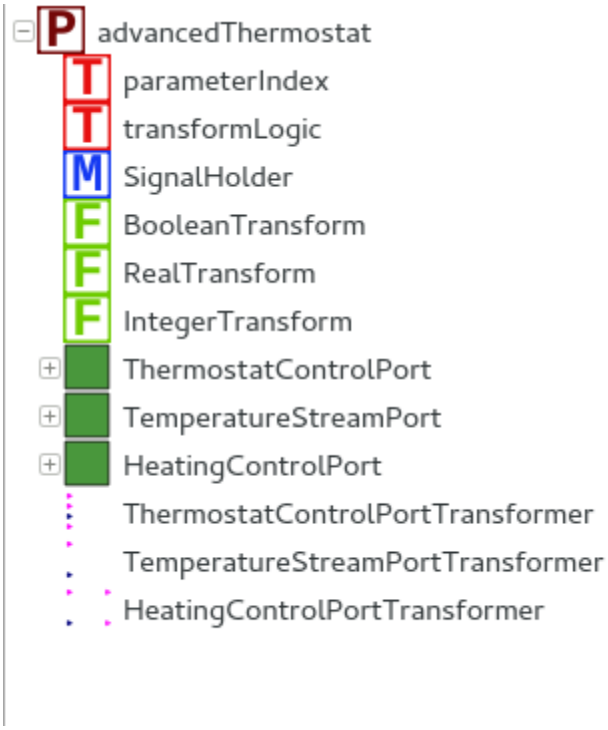


Figure 20. Loaded Modelica interface model

Drag and drop the Port and the PortTransformer pairs that you want to use. And make connections between the Port and the PortTransformer.

For example if you want use only HeatingControl, then Drag and drop ThermostatControlPort and ThermostatControlPortTransformer and add this line to the model at equation block to make connection (or just simply connect these in Diagram View). (Names of components can be different, because you can name the components at drag and drop)

```
connect(heatingControlPort1,
heatingControlPortTransformer1.HeatingControlPort1);
```

Now you are ready to use this interface, you just have to connect to its input and output ports that you want.

2.4 Description of the used xtUML model

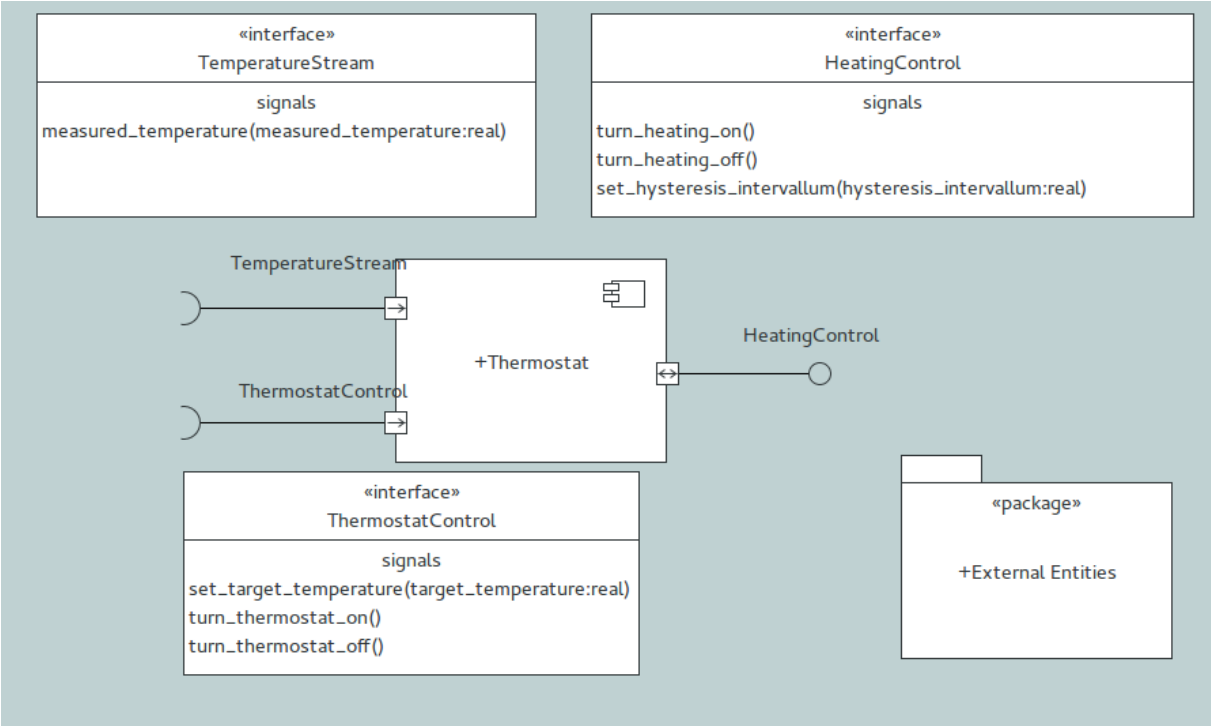


Figure 21. advancedThermostat xtUML model

The Thermostat is designed to control the heating of a room to keep its temperature at a desired value. It can communicate with 3 interfaces that it has: TemperatureStream, HeatingControl and ThermostatControl. The TemperatureStream is responsible for getting the current temperature of the room. The ThermostatControl is responsible for controlling the thermostat. It has 3 signals: *turn on* and *off* the thermostat and *set temperature* that heats the room to the desired temperature. HeatingControl has signals for both directions. *Turn heating on* and *off* are outputs of the model and *set hysteresis intervallum* is an input signal.

The inner logic of the model is represented in Figure 22.

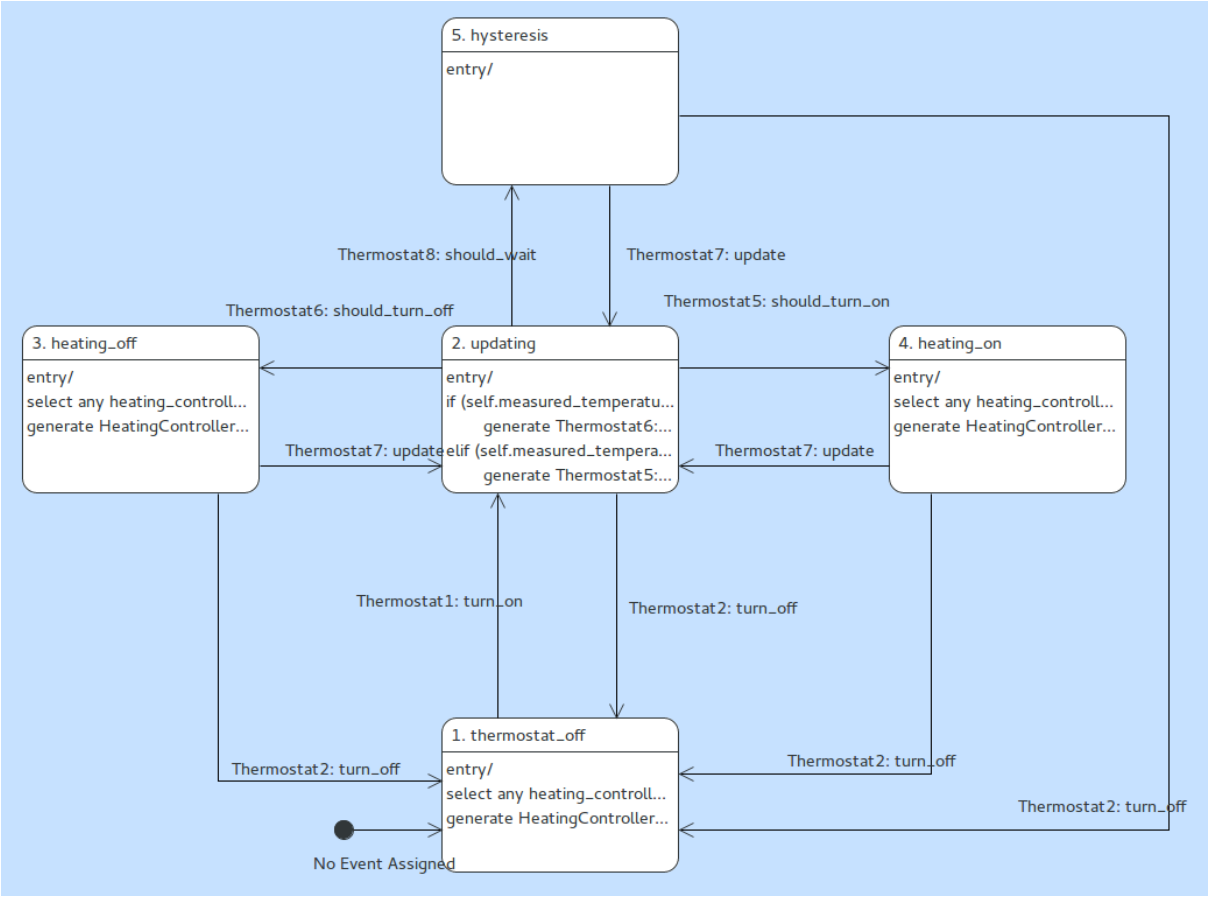


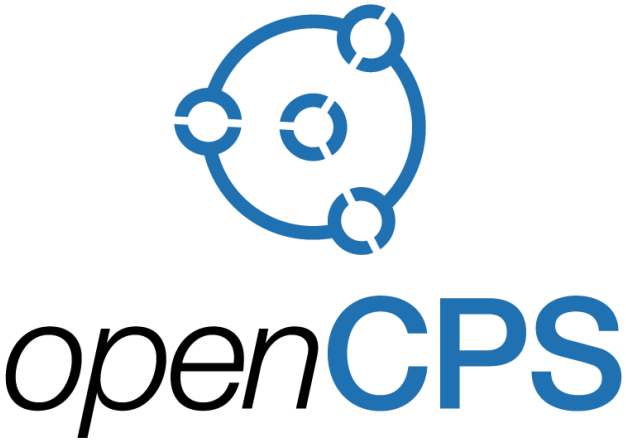
Figure 22. Thermostat class

The model starts in the first state and steps to *Updating* state when `turn_thermostat_on` signal is received. At the *updating* state it compares the measured temperature to desired temperature and if the measured temperature is bigger than desired temperature (plus hysteresis), then it steps to *heating_off* state but if the measured temperature is smaller than desired temperature minus hysteresis, then it steps to *heating_on* state. Otherwise it steps to *hysteresis* state. *Heating_on* and *heating_off* states send `turn_heating_on` and `turn_heating_off` signals, if a token enters them. From *heating_on*, *heating_off* and *hysteresis* states the model steps back to updating state if an update event is generated, that happens when the desired temperature, measured temperature or hysteresis intervallum changes. From all states the model will step to *thermostat_off* state, if it receives `turn_thermostat_off` signal.

3 SUMMARY

The current annex presents an FMU generator for tranlsatting xtUML to FMU. The main idea behind the transformation is to generate a wrapper for the xtUML executable code, which provides the interface between xtUML and FMU by performing translations between FMI variables and UML actions and also handling the control functions.

Annex 2 of D2.2	The txtUML modeling tool
Access[1]:	**PU**
Type[2]:	**Report**
Version:	**1.2**
Due Dates[3]:	M24



Open Cyber-Physical System Model-Driven Certified Development

Executive summary[4]:

This document presents txtUML latest version (0.7.0) which is a textual modeling tool for software development according to the executable UML paradigm. It provides two textual notations for defining models which can then be executed, debugged, integrated into Java programs, visualized, animated, exported to a standard UML representation and translated to C++ code. The generated C++ code is also packaged to a standard FMU.

1 Access classification as per definitions in PCA; PU = Public, CO = Confidential. Access classification per deliverable stated in FPP.
2 Deliverable type according to FPP, note that all non-report deliverables must be accompanied by a deliverable report.
3 Due month(s) according to FPP.
4 It is mandatory to provide an executive summary for each deliverable.

Deliverable Contributors:

	Name	Organisation	Primary role in project	Main Author(s)[5]
Deliverable Leader[6]	Jérémie TATIBOUET	CEA	Task Leader	
Contributing Author(s)[7]	Dávid János NEMETH	ELTE-Soft	Contributor	X
	András NAGY	ELTE-Soft	Contributor	X
	Zoltán Gera	ELTE-Soft	Contributor	
	Boldizsár NEMETH	ELTE-Soft	Contributor	
	Máté SZOKOLAI	ELTE-Soft	Contributor	
Internal Reviewer(s)[8]				

Document History:

Version	Date	Reason for Change	Status[9]
0.1	10/11/2017	Initial version of the deliverable	Draft
1.0	17/11/2017	Final version based on feedbacks	Final
1.1	17/11/2017	Apply remarks from Magnus	Final
1.2	20/11/2018	Updated version on FMI compati-bility	Final

5 Indicate Main Author(s) with an "X" in this column.
6 Deliverable leader according to FPP, role definition in PCA.
7 Person(s) from contributing partners for the deliverable, expected contributing partners stated in FPP.
8 Typically person(s) with appropriate expertise to assess deliverable structure and quality.
9 Status = "Draft", "In Review", "Released".

CONTENTS

CONTENTS3
ABBREVIATIONS3
1 OVERVIEW4
2 MOTIVATION4
3 ARCHITECTURE6
3.1 Standalone syntax7
3.2 Embedded language9
3.2.1 Static validation of the embedded language11
3.3 Exporting UML2 models11
3.4 Diagram generation12
3.5 Execution, debugging and animation13
3.6 Compilation to C++14
3.7 FMU export15
4 USER GUIDE16
4.1 Installation16
4.2 Sample models17
4.3 Creating own models17
4.4 Modeling language17
4.5 Generating diagrams18
4.6 Running and debugging models19
4.7 State machine animation20
4.8 Compilation to C++20
4.9 FMU export21

ABBREVIATIONS

List of abbreviations/acronyms used in document:

Abbreviation	**Definition**
API	Application Programming Interface
CSS	Cascading Style Sheets
DSL	Domain-Specific Language
EMF	Eclipse Modeling Framework
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
IDE	Integrated Development Environment
JDT	Java Development Tools
JVM	Java Virtual Machine
UML	Unified Modeling Language

1 OVERVIEW

The name *txtUML* stands for textual, executable, translatable UML [1]. It is an Eclipse-based tool built on top of JDT [2], Xtext [3] / Xbase [4] and Papyrus UML [5]. The tool is designed for textual model editing. This makes storage, version control, compare and merge processes, editing and searching easier and more efficient.

The tool supports two textual syntaxes for modeling: the standalone syntax which is designed to be clean and short and alternatively, the txtUML Java API which can be used to define models as standard Java programs. The tool supports the generation of graphical UML diagrams from the textual descriptions in the form of class and state machine diagrams. The layout of the diagrams can be controlled by a simple textual diagram layout language [6]. Models can be seamlessly integrated into Java programs, they can be executed and debugged. Generated state machine diagrams can be animated during model execution to further enhance comprehension of model dynamics.

Compatibility with other tools is ensured by generating standard UML models in EMF-UML2 [7] format. This representation is the input for our model compiler which generates C++ code, optionally packaged to produce an FMU.

2 MOTIVATION

Executable UML [8] models define both behavior and structure of software. These models can be executed, debugged and tested independently of the target platforms, providing early validation [9]. Model compilers translate them to efficient, platform-specific target code. Providing a practical toolchain for large scale executable UML modeling in industrial setup is challenging: version control, compare and merge functions, convenient editor, debugging support, high quality diagrams and model compilation need to be provided. On the other hand, the toolchain should be lightweight for scalability, stability and for low tool development costs.

Executable software modeling starts with a platform-independent model. Such a model is completely independent of the execution platform and implementation language and can be executed, debugged and tested on model-level. This enables early functional validation of the software being developed. In order to test and deploy the product on the target platforms, model compilers are used to generate code in selected implementation languages. These code generators take additional information about the specifics of the targeted platform (in the form of platform-specific model or platform description).

The key point here is model-level execution which enables the following two use cases:

- Interactive debugging: The execution of the model can be analyzed using the usual debugging features (breakpoints, stepping, variable view) and model specific features such as the animation of state machines. This use case requires the integration of the model execution engine with the user interface of the development environment.

- Automated mass testing: The model is exercised on a configured set of test cases as part of nightly testing or sanity checks before a commit. In this case command line compatible tooling is needed which can be easily integrated into testing frameworks. Runtime performance of the model execution engine is important in this use case.

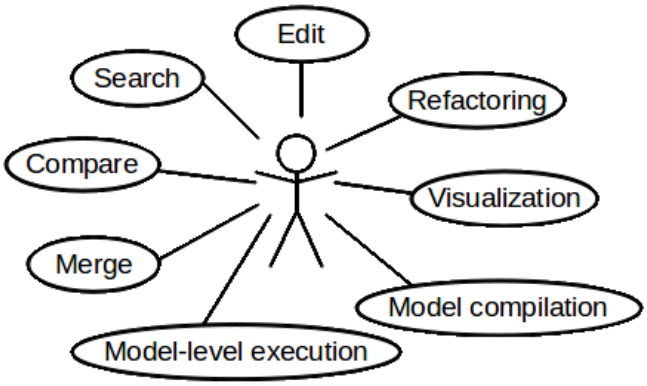


Figure 1: Use cases of executable UML modeling

An executable software modeling environment must support many functionalities: a model editor with the graphical visualization of the model, tools for model compare and merge, a debugger with graphical animations, a model compiler, etc. Figure 1 depicts the many different use cases such a toolset is responsible for. Our experience shows that the available open source tools still need to evolve a lot to provide convenient, robust, scalable and stable solution for all these requirements.

Textual modeling [10] solves many of these concerns: High-quality text editors with sophisticated editing and search-related features are available and users can select from numerous compare and merge tools. It is also faster for experienced developers to edit models

in text rather than to edit graphics which is partly the consequence of the maturity of text editors compared to graphical model editors. However, merely defining a textual notation for modeling does not solve all the issues. Text editors need plugins to do syntax highlighting and auto-completion correctly for the new language. Moreover, graphical visualization of certain kinds of models, like UML for example, is essential: Understanding a model is much easier by looking at an expressive diagram than reading text. Thus the visualization of the textual model must be established. Executable modeling makes even more heavy-weight demands: interpreter and debugger are also required.

3 ARCHITECTURE

txtUML proposes a novel architecture for text-based executable software modeling, taking into account the use cases and challenges discussed in Section 2. Figure 2 gives an overview of this architecture. Dashed lines on the diagram denote interaction between two modules, while continuous lines represent input and output. Modules with gray background are developed in the txtUML project while white ones are independent components we rely on.

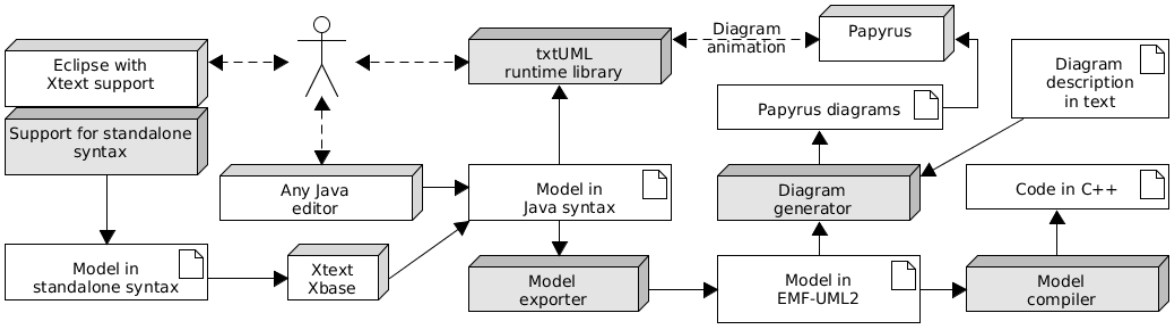


Figure 2: Overview of txtUML

Users define the UML models in text and have two options regarding the syntax: Standalone syntax is clean and short but users need to learn new syntactic elements. The other option is an embedded language in Java which is realized by a Java API providing the necessary constructs to define models. This option is useful for Java programmers not willing to learn new syntax and opens up possibilities to edit, run and debug txtUML models in non-Eclipse Java development environments.

Models in standalone syntax are translated on-the-fly to the embedded Java syntax using Xtext and Xbase as underlying Eclipse technologies. See Section 3.1 and Section 3.2 for details of the two syntaxes and the translation process. The resulting Java programs – on top of the txtUML runtime libraries – can be run and debugged in any Java environment. If Eclipse is used, Xtext and Xbase makes the standard debugging features (breakpoints, variable view, stepping in the code) available in the standalone syntax as well.

The Java programs defining models can be translated to EMF-UML2 representation which is the de facto standard format of UML models in Eclipse environment. In order to help

understanding and validating the models created in text, we generate UML diagrams compatible with the Papyrus open source UML framework, see Section 3.4. Currently class and state machine diagrams are supported. The txtUML runtime is able to communicate with the generated state machine diagrams and can animate them when the model is running or being debugged.

The toolchain is completed by a C++ code generator that uses the EMF-UML2 model as input, see Section 3.6. The toolchain can be extended by further project-specific code and document generators all working on the same, platform-independent EMF-UML2 representation.

The main novelty of this architecture is the multi-purpose Java syntax which is (1) a full-fledged language frontend, (2) the target of the translation from the standalone syntax and (3) the source of the UML model generation process at the same time. We summarize the most important advantages of this setup as follows:

- Running the models as Java programs provides higher performance than interpretation. This is important in automated testing scenarios.
- Learning new syntax and using its editor is not mandatory: The Java frontend is standard Java with a smart API and can be used in any Java development environment.
- The platform-independent, high abstraction level language allows the generation of standard UML models with diagrams and translation to platform-specific implementation languages.

3.1 Standalone syntax

From now on, *XtxtUML* will stand for the standalone syntax variant (as an abbreviation of Xtext-based txtUML) whereas we will refer to the Java-embedded alternative as *JtxtUML* (for Java-based txtUML) which will be discussed in detail throughout Section 3.2.

Essentially, the XtxtUML syntax can be considered syntactic sugar on top of JtxtUML as we map the elements of the former back the latter one. The base of this mapping is Xtext's built-in JVM types Ecore metamodel which is a sophisticated internal representation of the Java type system covering structural concepts such as class attributes and methods as well. As the compilation of its constructs to Java is predefined, in most of the cases merely specifying the connection between elements of our syntax and the JVM metamodel was sufficient to provide automatic code generation. The mapping itself is defined with the help of the framework's JVM model inferrer API.

One of the main advantages of using Xtext for implementing the standalone syntax variant is that in this way, highly customizable Eclipse IDE support such as syntax highlighting, hyperlinking and reference lookup is provided out of the box by the framework. Validation for language elements can also be defined in a declarative manner. The aforementioned mapping makes it possible to use XtxtUML entities and their generated JtxtUML equivalents interchangeably across other XtxtUML or even Java sources.

Based on Xtext, not only structural but also behavioral parts of the new language can be implemented. For the latter, significantly more challenging task we heavily modified Xtext's reusable expression language, Xbase – both in its grammar and semantics to suit our needs. Due to the overall customization-oriented nature of the framework it was even possible to extend Xbase with new expressions – e.g. signal sending and association navigation – by defining their syntax, type computation, compilation to Java and optional validation.

For a brief insight into XtxtUML, see the following example.

```
package examples.counter;

signal S;

class Sender {
  public void emit() {
    send new S() to this->(SR.r).one();
  }
}
class Receiver {
  private int count;

  initial Init;
  state Accepting;

  transition Initialize {
    from Init;
    to Accepting;
  }

  transition Accept {
    from Accepting;
    to Accepting;
    trigger S;
    effect { count++; }
  }
}

association SR {
  hidden 1 Sender s;
  * Receiver r;
}
```

This simple model consists of two classes, Sender and Receiver which are connected by the association SR. When the emit method of a Sender instance is called, it sends a new instance of signal S to one of the Receiver instances which are accessed by the aforementioned association.

The arrival of the signal triggers a reflexive transition in the receiver which – as its effect – increments the counter containing the number of received signals.

One of the main design concepts of XtxtUML was to provide a clean and intuitive, Java-like syntax both for structural entities and action code. We believe that using this approach not only is it easier for Java developers to become familiar with the language but the mapping to JtxtUML can be defined in a more straightforward way as well.

3.2 Embedded language

JtxtUML, our second syntax, is embedded in pure Java without any extensions or modifications to the host language, enabling users to write their models using only well-known language constructs and our API. The current implementation is based on Java SE 8, the newest version of Java as we aimed to provide a convenient, fast, easy-to-read syntax and for this reason we were ready to take advantage of any features that are provided by the Java SE.

As it was mentioned in previous sections, a JtxtUML model is also a runnable Java program in itself therefore speed is indeed an important aspect here. Although creating a user-friendly API sometimes requires slight compromises on runtime performance, our experience so far is that the achieved performance is more than good enough for testing and debugging purposes.

The following short example is the same that is shown in Section 3.1 but this time in JtxtUML.

```
package examples.counter;

import hu.elte.txtuml.api.model.*;

class S extends Signal {}

class Sender extends ModelClass {
   public void emit() {
      Action.send(new S(), this.assoc(SR.r.class).one());
   }
}

class Receiver extends ModelClass {
   private int count;

   class Init extends Initial {}
   class Accepting extends State {}

   @From(Init.class) @To(Accepting.class)
   class Initialize extends Transition {}

   @From(Accepting.class) @To(Accepting.class) @Trigger(S.class)
   class Accept extends Transition {
```

```
        @Override
        public void effect() {
            count++;
        }
    }
}

class SR extends Association {
    class s extends HiddenEnd<One<Sender>> {}
    class r extends End<Any<Receiver>> {}
}
```

As it can easily be noticed, JtxtUML is more verbose than its counterpart, the XtxtUML syntax but being an embedded language it has many advantages that make it a reasonable option to choose, like the aforementioned familiarity of Java developers or the off-the-shelf massive language support.

The example also shows the similarity of JtxtUML and XtxtUML which was an aim of our project as they are only syntactic variants of the same language with the ability to switch from one to the other, learning only minimal extra information. In case of becoming familiar with JtxtUML, this extra information is mainly about the Java language elements we use to represent those UML features that are not present in Java.

To describe the structure of a model, no mutable language constructs (like variables) are used to prevent accidental modification of the model structure at runtime. This approach resulted in the fact that almost all model elements are represented by a Java type – a Java class, in most cases – with a special super type to show the kind of the particular element and also to inherit behavior which becomes important when executing models. To keep JtxtUML code free from string literals referencing model elements by name – making refactoring really hard –, we take advantage of Java reflection which let us refer to a type at runtime through its associated java.lang.Class object.

Annotations and generics (type arguments) are widely used as well to write static information in JtxtUML models. Annotations are suitable for adding data that is not always required (e.g. the trigger of a transition), explicitly naming properties (e.g. the @From and @To annotations) or containing primitive values (e.g. the @Min and @Max annotations which are used to write custom association end multiplicities; this feature is not presented in the above example). Generics can help to reference types when this information is also required at compile time, like in the case of association ends, as the this.assoc call has to return a collection of the desired type. These type parameters are retrievable at runtime as well because they are set in the declaration of a type and that can be inspected with Java reflection.

Despite these powerful features of Java, some limitations of the language proved extremely hard to overcome. Type erasure, to begin with, deprived us of many possibilities to write things in a simpler way. The lack of value types forced us to use immutable classes which can be

inconvenient for the users too, as they also have to manually implement custom value types in an immutable and therefore verbose way. Garbage collection gives us no opportunity to force the deletion of objects from the heap or at least to check whether the user's code holds any references to them which would be helpful to effectively implement and dynamically validate model object deletion. The parameter passing rules of Java will make it challenging to implement UML's *out* and *in-out* parameter passing modes. However, the greatest limitation seemed to be the single inheritance of Java which made us unable to introduce multiple inheritance between model classes which is allowed in UML. The default Java solution for this problem, the usage of interfaces, could not be applied here because Java interfaces are too limited in features to be used instead of classes and it would be very inconvenient for a user to create both the interface and the implementing class for a single model class.

In case of the action code, both our opportunities and requirements proved to be much less than in the case of the model structure. It is simple Java action code with the extension of public and protected methods of API types, most importantly the class `Action`, whose static methods implement basic operations of JtxtUML, like sending signals, linking associations or deleting model objects.

3.2.1 Static validation of the embedded language

Enhancing Java with the required UML features is only part of the task when defining an embedded language like JtxtUML as Java provides many tools that cannot be translated to UML at all or only if they are used with certain restrictions. Examples include casting, threading and synchronization, local and anonymous classes; not to mention the various features of the standard library or any other libraries written in plain Java which may only be accessed from JtxtUML in a well-controlled way.

For this reason and to ensure the semantical correctness of the models as well, a validator is provided which uses the Java Development Tools Eclipse plugin to parse and check JtxtUML models. The use of JDT instead of standard Java reflection is an unfortunate necessity which is further explained in the next section as we first faced the decision between these two options during the implementation of the model exporter.

3.3 Exporting UML2 models

For visualizing and compiling the models we decided to export them into standard UML model format. The generated UML models are used as an intermediate representation for compilation to other programming languages and they can also be processed by external tools.

The export process is currently implemented as a batch operation converting the whole model at once. It parses all Java source files and outputs an EMF-UML2 model. We tried two approaches for extracting information from txtUML models:

- Java reflection and AspectJ: This solution uses standard Java reflection to analyze the structural elements (for example classes, method signatures) of the code. However, Java

reflection cannot provide information on method internals. Therefore, we experimented with AspectJ to export operations. AspectJ can inject aspects (additional method calls) to predefined points in the Java code and these aspects can collect the necessary information to complete the export.

- Parsing: In this case we parse the Java code using JDT and walk through the abstract syntax tree in order to translate the txtUML model to an EMF-UML2 model.

We have found out some drawbacks of the first solution: In that case the system has to run methods to analyze their body and each time a method call has parameters, dummy values need to be produced which complicates implementation and makes it fragile. Furthermore, AspectJ caused inconveniences while running the Java debugger on the model and interfered with debugging features provided by Xtext/Xbase for the XtxtUML syntax. As these problems became unmanageable, we switched to the second, JDT-based solution.

Another dilemma is about the representation of action code in the UML model. One possibility to encode behavior in UML is using opaque behaviors: These are just strings labeled with the name of the language they are written in. We decided not to use opaque behaviors for two reasons: Polluting the UML model with action code in XtxtUML or JtxtUML syntax would introduce non-standard elements, limiting the compatibility with third party tools. Also, a model compiler would have to parse and type check these opaque behaviors and do reference resolution, which introduces a lot of complexity. Therefore, we have chosen the other possibility, namely UML activities. This provides standard and language-independent action code format. On the other hand, it requires nontrivial translation logic both in the exporter module and in the model compiler. It is also a threat that UML activities are extremely verbose and this might lead to scalability problems in case of large models with much action code.

3.4 Diagram generation

While textual modeling is beneficial in several aspects, we consider graphical diagrams extremely important for understanding models. For this reason, we included a diagram generation module into the toolchain which produces Papyrus or JointJS diagrams on top of the exported EMF-UML2 models. As of now, class diagrams and state machine diagrams are supported.

The most important question of visualization is the layout. A popular solution is the application of autolayout algorithms. However, that is not ideal if users want to control the layout, possibly partially, and would like to store layout information under version control. To solve this problem, we have created a small DSL, embedded in Java, to define diagram layout concisely. The following example shows a layout definition for the model presented earlier in Section 3.1 and Section 3.2:

```
public class ExampleDiagram extends ClassDiagram {
  @Left(val = Sender.class, from = Receiver.class)
```

```
    public class MyLayout extends Layout {}
}
```

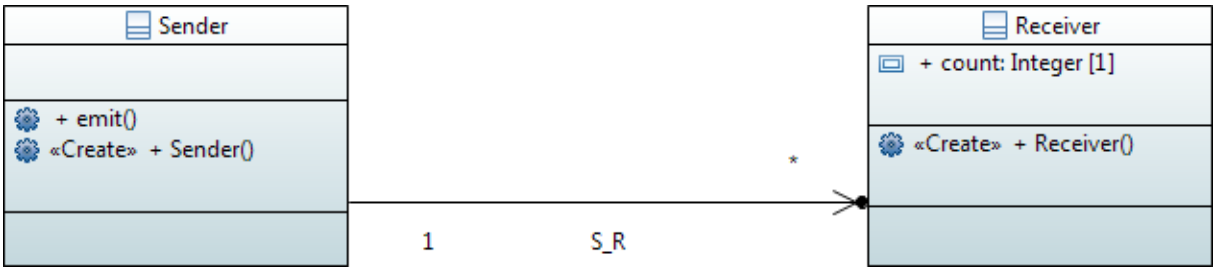


Figure 3: Generated Papyrus class diagram

This description requests the Sender class to be the left neighbor of the Receiver class. The resulting, generated Papyrus diagram is shown on Figure 3.

The language includes constructs to define the relative positioning of boxes on the diagram. The constraints are transformed to a linear inequality system of special form that can be solved by the Bellman–Ford graph algorithm. Once the boxes are placed on the diagram, the links are laid out on a grid using the A* algorithm with a cost function that minimizes length, number of crosses and turns.

3.5 Execution, debugging and animation

The architecture presented in this paper provides model-level execution for models written in any of the two syntactic variants. In case of the embedded language, the models are Java programs using the txtUML API and runtime library therefore these can be executed and debugged in any Java development environment. For models in the standalone syntax, execution and debugging controls are provided by Xtext, based on the transformation to the embedded language. This includes breakpoint support, variable view and session control functions like step over, step into, step out, resume and stop.

The model execution runtime library adds two useful features: runtime validation and state machine diagram animation. Runtime validation generates warnings, for example when multiplicity constraints are violated or signals are dropped. This feedback helps the modelers to find bugs early in the development process, even without generating code and deploying it on a target platform.

The runtime behind txtUML API does have sophisticated tracing capabilities. These are switched off by default when the program is run as a plain Java application. If the extra

functionality is switched on, the Eclipse-side plugin makes a connection towards the runtime in order to receive trace information. Data is provided even about individual object states and are fully kept track of.

The trace data is on one hand used to provide the user with sophisticated warnings, and on the other hand to animate the generated state machine diagrams. This is achieved by the CSS capabilities of Papyrus and modern web browsers. Because normal debugging features still work in this mode, breaking or reaching a breakpoint gives the possibility to examine various model states on the paused animation (see on Figure 4 and Figure 5).

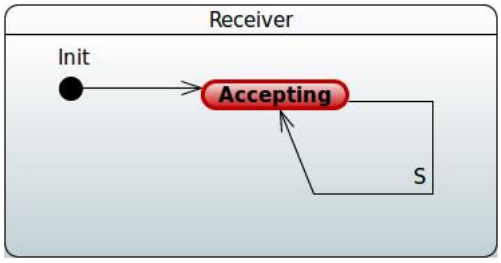


Figure 4: State machine is residing in a state

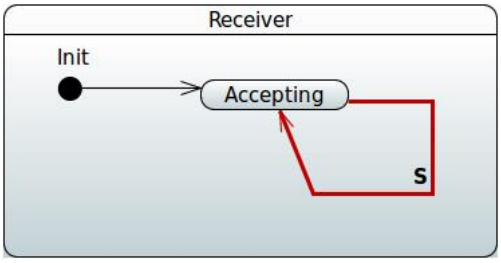


Figure 5: State machine is performing a transition

3.6 Compilation to C++

We made a significant design choice by compiling from EMF-UML2 instead of using the original XtxtUML/JtxtUML code. The EMF-UML2 representation created from XtxtUML/JtxtUML code is a de facto standard and gives enormous flexibility for our tool. By using EMF-UML2 directly, compilation fits into the general exporting framework and can use all the benefits and generality of other export methods. Support for a new language, a new tool or even a graphical representation can be easily added the same way because the exporting mechanism does not rely on any specifics of the Java code. This gives a true independence between model execution/testing and the compiled code which makes the development more robust. Currently we support compilation to standard C++14 (tested on *gcc*, *clang*, *msvc*).

We separate the compiled code into the following parts:

- Code generated from the EMF-UML2 representation of models.
- Prewritten runtime with the following components:
    - Model executors, currently two is provided:
        - The single threaded executor which processes model messages synchronously in a central event loop and terminates when all of them are handled.
        - The multithreaded executor which deploys model objects to separate threads – according to the deployment configuration – and runs multiple

event loops, not terminating even if there are currently no messages to process.

  - API based on JtxtUML action methods. It facilitates communication between models and the outside word to make integration with external applications easier.
  - State machine and event support with predefined base classes for model elements. This is to keep the amount of code needed to be generated to a minimum.

- Generated deployment configuration settings specifying thread usage – including object distribution and scaling – and the type of model executor to be used.

By keeping these isolated, the generated code is practically easier to integrate. Deployment settings reside in a few specific files and do not pollute the model code which stays clean this way. Settings can be easily changed in the configuration files without recompilation. The support runtime can be freely interchanged by another one. We plan to support the versatile usage of the compiled code by spending efforts on adding more runtimes and developing a rich deployment configuration for multiple target platforms. Support for composite structures is released which is also make possible to model interprocess communication between components via ports.

3.7 FMU export

Opposed to physical models usually described by differential equations, executable UML models are represented by classes and event-driven state machines. As the interface defined by the FMI standard is mostly suitable for models of the former type, transforming an UML model into an FMU is not a trivial task. An FMU wrapper has to be provided to manage exposed variables, life cycle and execution of UML models through FMI functions. An ideal solution should work for any existing UML models without requiring additional modifications but unfortunately, in our current implementation we had to make certain constraints on source models.

We have decided to explicitly represent the environment (or physical reality) with a predefined class even on model level. In each model a singleton object has to be specified which should connect itself with the instance of the aforementioned environment – passed through its constructor – via an association. Simulation steps are controlled with preselected signals. At the start of a step, a signal containing the values of input variables in its attributes (the input signal) is sent to the model object. The simulation step ends when the model object sends a signal containing the values of output variables in its attributes (the output signal) to the environment.

To keep the packaged FMU as lightweight as possible, we have decided not to use the XtxtUML/JtxtUML representation with our model executor written in Java as its base. Instead, we rely on the model compiler discussed in Section 3.6 to generate C++ code from the source model and then include the exported code in the FMU. This also makes implementing the

necessary C functions of the FMI standard easier. In fact, the class representing the environment in the source model is only a stub which is handled differently during the code generation process, eventually being transformed into the partially predefined implementation of our FMU wrapper.

Note that the wrapper is not entirely predefined: some parts are generated dynamically from the source model, e.g. which depend on input variables. In order to improve maintainability, we split the wrapper definition in half:

- An abstract base class containing attributes and functions which are required to ensure compatibility with the C++ model execution runtime but are independent from FMI concepts.
- A subclass of the aforementioned base which implements FMI-specific features only. This is where FMI functions are defined according to the semantics specified above. For example, the fmi2DoStep function sends an input signal to the model object parameterized with the input variables stored in the wrapper. Similarly, when the wrapper processes an instance of the output signal it updates the output variables based on the received signal and calls the stepFinished callback to mark the simulation step as finished.

While this solution certainly works, it is unfortunate that UML models have to satisfy specific constraints if we want to translate them into FMUs.

4 USER GUIDE

This short user guide is intended to provide practical details about the usage of the tool. A more complete documentation can be found at the official website[10] of the project. We also recommend watching our introductory video[11]. Alternatively, you can visit our GitHub repository[12] as well.

4.1 Installation

As txtUML is implemented as a set of Eclipse plugins, at first you have to install Java and Eclipse. Currently Java 8 and Eclipse 4.6.2 (Neon.2) is supported. For version 0.7.0, add the following update site in Eclipse (*Help > Install New Software... > Add...*):

Select txtUML to be installed and let Eclipse guide you through the rest of the installation process. Restart Eclipse at the end of the installation process. In case of a successful installation,

10
11
12

the txtUML menu appears in Eclipse's menu bar and there is a txtUML wizard category when selecting *File > New > Other...*.

4.2 Sample models

For a quick start we recommend experimenting with the sample models[13]. Download and unzip the sample models. Import them into your Eclipse workspace (*File > Import... > General > Existing Projects into Workspace*). Clean and build the projects (*Project > Clean...*). Sample models are implemented using either XtxtUML syntax (see the source packages *<name of example>.x.model*) or JtxtUML syntax (see the source packages *<name of example>.j.model*), or both. In addition, the sample models are accompanied with diagram descriptions (see the Java classes inheriting from the ClassDiagram or StateMachineDiagram type).

4.3 Creating own models

- New txtUML project: txtUML models should be placed in txtUML projects. A new txtUML project can be created by selecting *File > New > Project... > txtUML > txtUML Project* and setting the project name. By default, the project will be created in the current workspace. To override this, uncheck the *Use default location* checkbox and select a location for the new project.
- New txtUML model: Select *File > New > Other... > txtUML > txtUML Model*. Select a *Source folder* from an existing project for the new model. Select an existing *Package* from that folder or type a new package name. Type a *Name* for the new model. Select the *syntax* of the new model: XtxtUML for custom modeling syntax or JtxtUML for Java syntax. Both XtxtUML and JtxtUML models can be connected with Java code, can be run and debugged and used as a source for Papyrus UML model generation. A txtUML model is a package with either a *package-info.java* file (in case of JtxtUML) where the package has an annotation of the form @Model("ModelName") or a *model-info.xtxtuml* file (in case of XtxtUML) which has a model declaration of the form model-package example.x.model as "ModelName";. All files in this package (and its subpackages) are part of the model. The wizard described above creates one of these files depending on the XtxtUML/JtxtUML selection.
- New model elements: For XtxtUML syntax, select *File > New > Other... > txtUML > XtxtUML File*. Fill in the source folder and package to place the new source file in, then enter a file name. You can also choose between the two possible extensions: *.xtxtuml* or *.txtuml*. For JtxtUML syntax, select *File > New > Class* to create a new Java class.

4.4 Modeling language

The txtUML language covers a subset of UML. We summarize the supported elements below:

13

- Class modeling: classes with attributes and operations; simple binary associations; compositions; (single) inheritance.
- State modeling: simple and composite states; transitions triggered by signals; guards; choice states.
- Behavior modeling: action code can be written in operations of classes, entry and exit actions of states and effects of transitions. Supported base types are int, double, boolean and String with the usual arithmetic and logic expressions, variables and assignment. Control structures (loops, branches), attribute access and operation calls are supported. UML-specific actions: creation and deletion of objects; linkage and unlinkage via associations and connectors; reading links; sending signals; accessing signal data in entries, exits and effects.
- Component modeling: interfaces containing signals; ports; connectors.

The design of the two language variants follows a pattern: kinds of the model elements are shown by keywords (signal, class, transition) in XtxtUML while the Java version uses inheritance from Signal, ModelClass and Transition. These classes are provided by the txtUML Java API. Properties of the transitions are expressed by Java annotations (e.g. @From) in Java, while attribute-like syntax with keywords (e.g. from) is used in the standalone version. See the demo models or the Language Guide[14] to study the txtUML language both in XtxtUML/JtxtUML. In case of JtxtUML, the JavaDoc[15] of the API can also be used.

4.5 Generating diagrams

It is possible to generate EMF-UML2 models together with Papyrus or JointJS diagrams from txtUML models. Currently class and state machine diagrams can be generated. Content and layout of the class diagrams and flat state machine diagrams can be defined by textual diagram descriptions.

The following simple example assumes classes A, B, C and D in the model. We create a class diagram where classes A, B and C are in a row, and class D is below B. Diagram definitions can be written using a Java API. See the Diagram Language Guide[16] for a detailed description.

```
public class ExampleDiagram extends ClassDiagram {
  @Row({A.class, B.class, C.class})
  @Below(val = D.class, from = B.class)
  class ExampleLayout extends Layout {}
}
```

14
15
16

To generate diagrams, select *txtUML > Generate Papyrus diagrams from txtUML* or *txtUML > Generate JavaScript diagrams from txtUML* from the menu bar. Diagram descriptions are grouped by projects. You can select several descriptions – if the descriptions are related to different models, a separate Papyrus project will be generated for each individual model. Diagrams can be generated from a context menu as well, either in the *Project Explorer* or in the *Package Explorer*. Simply right click on a diagram description file (you can select several descriptions too) and choose *Visualize as Papyrus diagram* or *Visualize as JavaScript diagram*.

4.6 Running and debugging models

Models in txtUML can be run as Java applications with the help of a model executor which is part of the txtUML modeling API. While it is possible to write custom model executors, the default one will be sufficient in most cases. Model executors can be managed through the ModelExecutor interface. This interface has two static create methods to instantiate the default executor. The first is without parameters while the second one takes a single String argument as an optional name for the new executor instance. This name will appear in the automatic logs.

In the simplest case, a main Java class that solely executes a model would look like this:

```
public class Tester {
   public static void main(String[] args) {
      ModelExecutor.create().run(() -> {
         MyClass instance = Action.create(MyClass.class);
         Action.start(instance);
         Action.send(new MySignal(), instance);
         // ...
      });
   }
}
```

The run method takes a Runnable instance, the initialization of the model execution which should create, link, start and send signals to the model objects which are required at the beginning of the model execution. This initialization code will run as part of the model, so any action that is allowed in the model is also allowed here.

The model executor writes log messages to the console and to a log file. Runtime errors and warnings are always logged but there is an optional trace logging which reports all important events during the model execution, for example, when a state machine of a model object leaves or enters a state. This trace logging is switched off by default but can be switched on with the setTraceLogging method. It is important to call this method before starting the model execution with the run method.

Models in txtUML can be debugged as well. Switch to Java or Debug perspective and create a new run/debug configuration of type *Java Application*. Breakpoints can be created and managed the same way as for Java programs. The standard debug controls (stop, pause, resume,

step, step-into) work as usual. The variable view can show the current signal, current state, associations and the attribute values of the actual object.

4.7 State machine animation

State machine diagrams generated by txtUML can be animated. Create a new run/debug configuration of type *txtUML Application*. Open the generated Papyrus or JointJS diagram and start the model either in run or in debug mode. For a JointJS diagram, the port displayed by the running model must be copied into the appropriate field in the browser then the switch must be turned 'on'. The current state and currently executed transition gets highlighted. For each state machine diagram, the state changes of the first activated object of the corresponding type will be highlighted. An expected later improvement will make it possible to select the object to be animated during the debug session.

4.8 Compilation to C++

The C++ model compiler can be reached by selecting the *txtUML > Generate C++ code from txtUML menu*. To generate code, a txtUML deployment configuration must be specified. The runtime library contains only prewritten *.cpp* files so they can be used for other generated models too. A deployment configuration is a description of how the object instances will be distributed into different threads. This is a special class which is derived from the Configuration base class. The model classes can be grouped together and the events that arrive to classes belonging to the same group will be served by a configured thread pool.

For example, consider the following configuration:

```
@Group(contains = {A.class, B.class},
class DefaultConfiguration extends Configuration {}
```

This means that A and B will be served by the same thread pool and the remint classes will be grouped in the default group. A more complex example:

```
@Group(contains = {A.class, B.class}, max = 10, constant = 2, gradient = 0.5)
@Group(contains = {C.class})
class ExampleConfiguration extends Configuration {}
```

This means that instances of classes A and B are served by the same thread pool which contains two constant threads plus one for every 2 A or B instances created, but no more than 10. Instances of class C are served by another thread pool which contains only one thread (according to the default values).

The generated C++ code is saved in the *cpp-gen* folder of the selected project. Note that you might have to refresh the folder so that the newly generated files become visible in Eclipse. You can compile the generated files with any C++ compiler manually but we suggest using the compiler environment selector dialog to create native "makefiles" that can be used in the

compiler environment of your choice. The generated compiler environment is placed under the *<environment>_build* folder.

The selector contains only the commonly used environments so you can create any other environment by the generated *CmakeList.txt* file. It is recommended to create a new folder next to the generated files, where the build environment should be created. The compilation can be performed by the following command:

```
cmake -G <environment> -D CMAKE_BUILD_TYPE=<type> <path>
```

Where the parameters mean the following:

- <environment>: The chosen build environment. You can use the cmake --help command to list the possible build environments.
- <type>: The type of the build. Can be Debug or Release.
- <path>: The relative path to the generated *CMakeLists* file.

A concrete example:

```
cmake -G "MinGW Makefiles" -D CMAKE_BUILD_TYPE=Release ..
```

After translating a txtUML model to C++, you might want to create a *main.cpp* for testing purposes where you instantiate model objects and send signals to them. This can be achieved with our C++ runtime API. For more information, see the *main.cpp* files included with the demo models (look for the source packages *<name of example>.j/x.cpp* where they exist). A custom *main.cpp* file can be selected by the exporter dialog.

4.9 FMU export

The FMU export functionality is not yet released but it can still be tested using a custom txtUML build. From our GitHub repository download tag *fmu-export2*[17] and set up a development Eclipse according to our GitHub wiki page titled *Installation of the Development Environment*[18]. Using the *Launch Runtime Eclipse* run configuration you can start an Eclipse instance with FMU export capabilities. The complete feature will be launched in the next release.

In this Eclipse instance import the *MoonLander* project from the *examples/tests* folder of the downloaded branch into your workspace (*File > Import... > General > Existing Projects into Workspace*). Select *txtUML > Generate C++ code from txtUML*, specify the MoonLanderConfiguration class as deployment configuration, check the *Generate FMU* option and select the MoonLanderFMIConfiguration class as FMI configuration. After choosing *Finish*, the *MoonLander* model gets exported as an FMU. The generated files are placed under the *cpp-*

17 fmu-export-2
18

gen folder of the project and they are automatically complied under the *fmu_build* folder. A *<modelname>.fmu* file is also placed here which is possible to run by the *OMSimulator*.

It is also possible to run the compiled FMU with our debugger tool. This takes a text file as input where each line represents a simulation step and contains values of input variables. The debugger instantiates the FMU (fmi2Instantiate) and for each line of the input file it sets input model variables (fmi2Set), requests a simulation step (fmi2DoStep) and finally queries (fmi2Get) and prints output variables to the console. The debugger is automatically compiled along model files as the fmudebug executable to the same folder where other C++ files are built. Assuming that this directory is *<project root>/cpp-gen/<model package>* and an input file named *input.txt* is defined in *<project root>*, from the folder of the generated files the debugger can be started with the following command:

```
./fmudebug ../../input.txt
```

Such an input file is provided for the *MoonLander* example as well. Feel free to experiment with its contents and see how the FMU reacts.

Refeences

[1]
OMG, "Unified Modeling Language (UML)," 2015. [Online]. Available: http://www.omg.org/spec/UML/2.5/.
[2]
Eclipse Foundation, "Java Development Tools," [Online]. Available: https://www.eclipse.org/jdt/.
[3]
Eclipse Foundation, "Xtext," [Online]. Available: https://www.eclipse.org/Xtext/.
[4]
Eclipse Foundation, "Xbase," [Online]. Available: https://wiki.eclipse.org/Xbase.
[5]
Eclipse Foundation, "Papyrus," [Online]. Available: https://www.eclipse.org/papyrus/.
[6]
B. Gregorics, T. Gregorics, G. F. Kovács, A. Dobreff and G. Dévai, „Textual Diagram Layout Language and Visualization Algorithm," in *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems*, Ottawa, Canada, 2015.
[7]
Eclipse Foundation, "EMF-UML2," [Online]. Available: http://www.eclipse.org/modeling/mdt/?project=uml2.
[8]
S. J. Mellor and M. . J. Balcer, Executable UML: A Foundation for Model-Driven Architecture, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
[9]
G. Dévai, M. Karácsony, B. Németh, R. Kitlei and T. Kozsik, "Execution via Code Generation," in *1st International Workshop on Executable Modeling*, Ottawa, Canada, 2015.
[10]

H. Grönniger, H. Krahn, B. Rumpe, M. Schindler and S. Völkel, „Textbased modeling," in *4th International Workshop on Software Language Engineering*, Nashville, TN, USA, 2007.