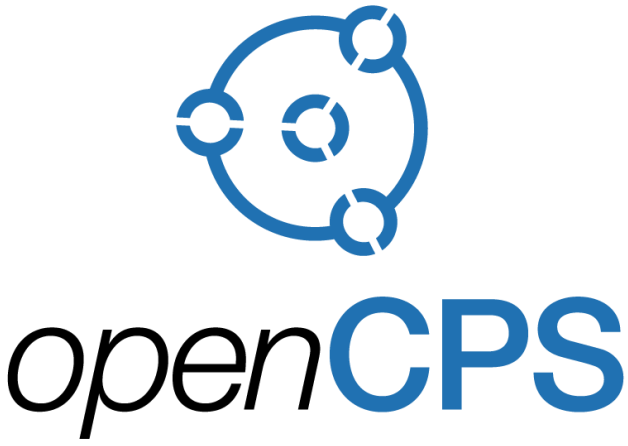
D2.2	Interoperability of the standards Modelica-UML-FMI
Access[1]:	**PU**
Type[2]:	**Report**
Version:	**0.8**
Due Dates[3]:	**M12, M24**



Open Cyber-Physical System Model-Driven Certified Development

Executive summary[4]:

At design time, a complex system is defined by a set of models which describe the different parts of the system. These models can be formalized using different modelling languages. Hence, the overall system is a set of heterogeneous artefacts that still need to be simulated altogether to assess the global system behaviour on a set of well identified scenarios. FMI standards defines how these artefacts can be combined in a simulation as a set of FMUs and also formalizes what the meaning of a simulation step is on this model. UML models can be used in such a simulation process, however the current set of language elements that have a formal semantics does not include state machines although these latter are heavily used across a large set of domains to model the dynamic of software applications. To make the usage of UML state machines possible in a simulation process, their semantics must be formally described. This deliverable reports the work done to define a Precise Semantics for UML state machines and normalize it at the OMG. In addition it also identifies how the UML specification could be extended to allow execution of xtUML state machines which are especially used at an industrial level by Ericsson and Saab and part of the xtUML (executable UML) language.

[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	T2.2 leader	X
Contributing Author(s)[7]	Jérémie TATIBOUET	CEA	T2.2 leader	X
	Gergely DEVAI	ELTE-Soft	T2.2 member	X
	Bernhard THIELE	LIU	T2.2 member	X
	Ákos Horvath	IQL	T2.2 member	X
	Gergely Seres	Ericsson	T2.2 member	X
Internal Reviewer(s)[8]	Ákos Horvath	IQL	T2.2 member	X

Document History:

Version	Date	Reason for Change	Status[9]
0.1	29/10/2016	Initial version of the deliverable	Draft
0.2	06/11/2016	Integration of the analysis of xtUML state machines semantics specifics	Draft
0.3	06/11/2016	Integration of Modelica language short description in clause 1.1	Draft
0.4	07/11/2016	Complete section 3 with description on PSSM differences compared to xtUML state machines.	Draft
0.5	10/11/2016	Add executive summary. Include minor adjustments to conclusions.	Draft

[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".

0.6	10/11/2016	Complete review is done and some minor adjustments are added to certain sections	In Review
0.7	11/11/2016	Apply initial review suggestions.	In Review
0.8	12/11/2016	Minor editorial improvements to finalize the document for release.	Released

CONTENTS

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
UML	Unified Modelling Language
RFP	Request for Proposal
FUML	Semantics of a Foundational Subset for Executable UML Models
PSCS	Precise Semantics of UML Composite Structures
PSSM	Precise Semantics of UML State Machines

ALF	Action Language for Foundational UML
RTC	Run to Completion
OMG	Object Management Group
EOO	Equation-based Object-Oriented
MLS	Modelica Language Specification

1 INTRODUCTION

1.1 Task Analysis

The task is entitled “Interoperability of the standards Modelica-UML-FMI”. Keywords of this title are *UML*, *Modelica* and *FMI*.

- UML [1] is a standard modelling language that can be used to describe the structure and the dynamic of a complex system. This modelling language is particularly well suited to precisely describe the software parts of a system and partially the execution platform (usually HW). The language has a precise semantics defined in its standard, however, some of this semantics are still described in natural language (i.e., in English) but a growing subset is now associated to a formal (i.e., operational) definition. The subset that currently has a formal semantics includes classes, composite structures and activities. These semantics are respectively described in fUML [2] and PSCS [3] documents. Any model conforming to the aforementioned subset can be precisely executed, thus also simulated.

- Modelica is language for describing the dynamic behavior of technical systems consisting of mechanical, electrical, thermal, hydraulic, pneumatical, control and other components. The behavior of models is described with ordinary differential equations (ODEs), algebraic equations (AEs), event handling and recurrence relations (sampled control). Object-oriented concepts are supported as a means of managing the complexity inherent to modern technical systems. Modelica can therefore be called an equation-based object-oriented (EOO) language. The most recent standard version is the Modelica Language Specification (MLS) 3.3 [4]. A brief discussion on Modelica semantics is further provided in the OPENCPS D3.2 report “Translation validation and traceability concept from acausal hybrid models to generated code”.

- FMI defines an open tool independent standard enabling the combination of a set of models (developed in different tools) describing the different parts of a complex system in a single simulation model. Each artefact contributing to this model is described as an FMU. An FMU is a box which exposes inputs that need to be provided for the underlying simulation model as well as outputs produced by this latter. It exists two types of FMU: *model exchange* and *co-simulation*. According to the FMI 2.0 specification [4], the kind *model exchange* means that the “*FMU includes the model or the communication to a tool that provides the model, and the environment provides the simulation engines*”. Conversely, the kind *co-simulation* means that “*FMU includes the model and the simulation engine, or a communication to a tool that provides the model and the simulation engine, and the environment provides the master algorithm to run coupled FMU co-simulation slaves together*”.

The word linking the aforementioned keyword is *Interoperability*. FMI standard provides a way to couple parts of a complex system that would have been specified using either with Modelica or UML. The assumption to make this coupling possible is that tools enabling the definition of models conforming to these languages are able to export FMUs for co-simulation. As a reminder, tools involved in that part of the project are Papyrus and Open Modelica. Papyrus provides the possibility to describe part of the system using UML while Open Modelica provides the possibility to describe part of the system using the Modelica language.

At simulation time, FMUs for co-simulation provided by these two tools interoperate thanks to the semantics defined by the FMI standard. This semantics is captured by the master algorithm implemented by the simulation environment in which FMUs are imported and connected. While it is clear that FMI enables models specified in both languages to be involved in a co-simulation process, it is not clear what does a master simulation step implies in terms of simulation progress in models specified in UML. This problem is explained in the next section.

1.2 Problem Statement

UML provides the possibility to describe the dynamic of a system using state machines. This formalism is very popular for modelling event-based reactive behaviours and is widely used in industry. It is often used jointly with activities which are used to describe low-level computations occurring when a state is entered, exited or when a transition is traversed. These activities can be specified using the Alf textual notation [5]. The purpose of this standard notation is to provide users with an easy way to implement and maintain complex activities in their models.

Unlike for activities, classes and composite structures whose execution semantics are formally defined in fUML and PSCS, state machines semantics remain specified in a natural language (i.e., English). Hence:

1. It is not possible to use in simulation process models whose dynamic is specified using UML state machines.
2. As the semantics is not formally defined it is not possible to clarify what are the implications in terms of simulation progress between a step asked by the master algorithm and run-to-completion steps performed in an executed state machine.

On way to resolve point 1 and to clarify point 2 is to provide a formal definition of UML state machines semantics. One can argue that in the past many tools provided a semantics for UML state machines. That is true, but it is not possible to say that semantics described by these tools fully complied with the one described by UML since no reference model was developed to assess this conformance. In addition, none of the defined semantics were standardized nor relied on the standard describing semantics of a foundational subset for executable UML models: fUML.

1.3 Deliverable Content

Two different releases are required for task 2.2 of work-package 2: M-12 and M24. The current document corresponds to the M-12 release. Two main contributions are described:

1. The definition of a precise semantics for UML state machines. The resulting specification document is very likely to be adopted as an official OMG standard. The current document describes contributions to construct the PSSM specification (as a response to the PSSM RFP [7]). It especially provides an overview of the core semantics and it explains how the defined semantics was tested (via the test suite and the prototype implementation of the semantics) to ensure it matches the one defined in the UML standard.

2. A preliminary analysis was conducted by Inc-Query, ELTE-Soft, Ericsson, Saab AB and CEA to determine what are the semantics differences existing between UML state machines and xtUML state machines. This analysis is based on the standard semantics defined for UML state machines in response to the PSSM RFP [6]. Its purpose is to evaluate the required effort to capture xtUML state machines semantics as an extension of UML state machine semantics. The final goal of this work is to use such extension to allow xTUML models to be executed and leverage the possibility to export such models into FMUs.

These two contributions are respectively described in sections 2 and 3 of this document.

2 PRECISE SEMANTICS OF UML STATE MACHINES

The purpose of this section is to provide of an overview of the work done in response to the PSSM RFP [6]. Clause 2.1 present the scope of the specification. Clause 2.2 describes the specification architecture as well the methodology followed to build the specification. Clause 2.3 focus on the PSSM specification content. It especially provides details on the PSSM subset and on the core part of the semantic model. In addition, this clause explains the test suite architecture and provides a detailed explanations about the execution of two test cases. Note that information about implementation of the semantic model and how the RFP requirements are addressed are also provided in sub clauses 2.3.4 and 2.3.5.

2.1 Scope of the specification

This sub clause establishes the relationship of PSSM to the syntactic and semantic models from fUML [2] and PSCS [3] specifications (see Figure 1). Thanks to this relationship it highlights that by construction the way PSSM is defined complies with the way fUML and PSCS were defined.

2.1.1 Overview

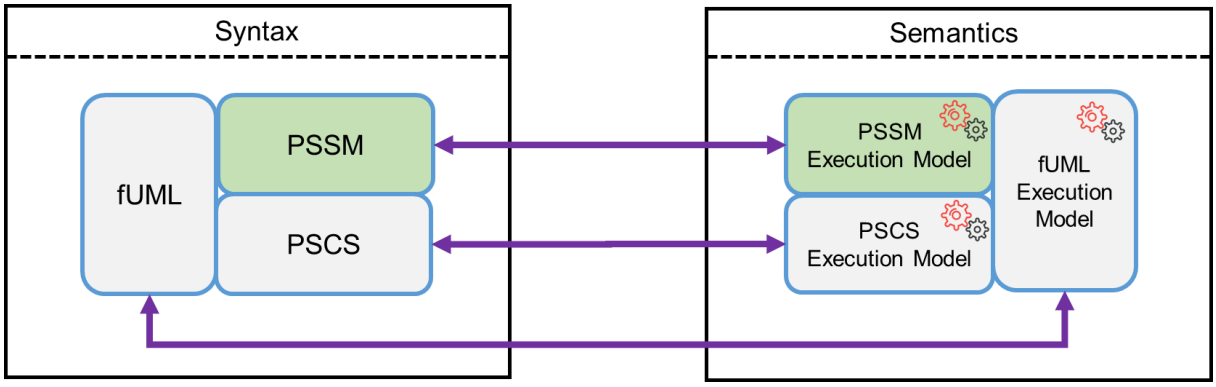


Figure 1 - Scope of this specification

The Precise Semantics of UML State Machines specification is an extension of the Semantics of a Foundational Subset for Executable UML (known as “foundational UML” or “fUML”) that defines the execution semantics for UML state machines.

Syntactically, this specification extends fUML with a large subset of the abstract syntax of state machines as given in UML (see chapter 14 for [1] and later versions). Semantically, this specification extends the fUML execution model in order to specify the operational execution semantics of the state machines abstract syntax subset.

In practice, the semantic model defined to capture UML state machines semantics is an extension of the semantic model described in Precise Semantics of UML Composite Structures (PSCS). The semantic model described in this standard is itself an extension of the one described in fUML. The definition of PSSM semantic model on top PSCS semantic model ensures that semantics given in this specification are compatible with the extensions defined in PSCS.

2.1.2 Conformance Levels

Even though PSSM is built on top of PSCS, a tool implementing this specification is not required to demonstrate a conformance to PSCS to also demonstrate a conformance to PSSM. In order to make this possible the specification defines two levels of conformance:

1. **PSSM-only**. To demonstrate conformance to this level, a tool must implement fUML and PSSM. In addition, it must be able to pass all tests described in the PSSM test suite but not those defined in the PSCS test suite.

2. **PSSM and PSCS**. To demonstrate conformance to this level, a tool must implement fUML, PSCS and PSSM. In addition, it must be able to pass all tests defined in both PSCS and PSSM test suite.

In other words, if the level of conformance is **PSSM-only** that means the tool implementing the specification is able to execute any model conforming to the abstract syntax subset covered by both fUML and PSSM. However it will not be capable of executing models relying on concepts provided by the PSCS subset. To achieve this a joint **PSSM and PSCS** conformance is required.

2.2 Specification Architecture

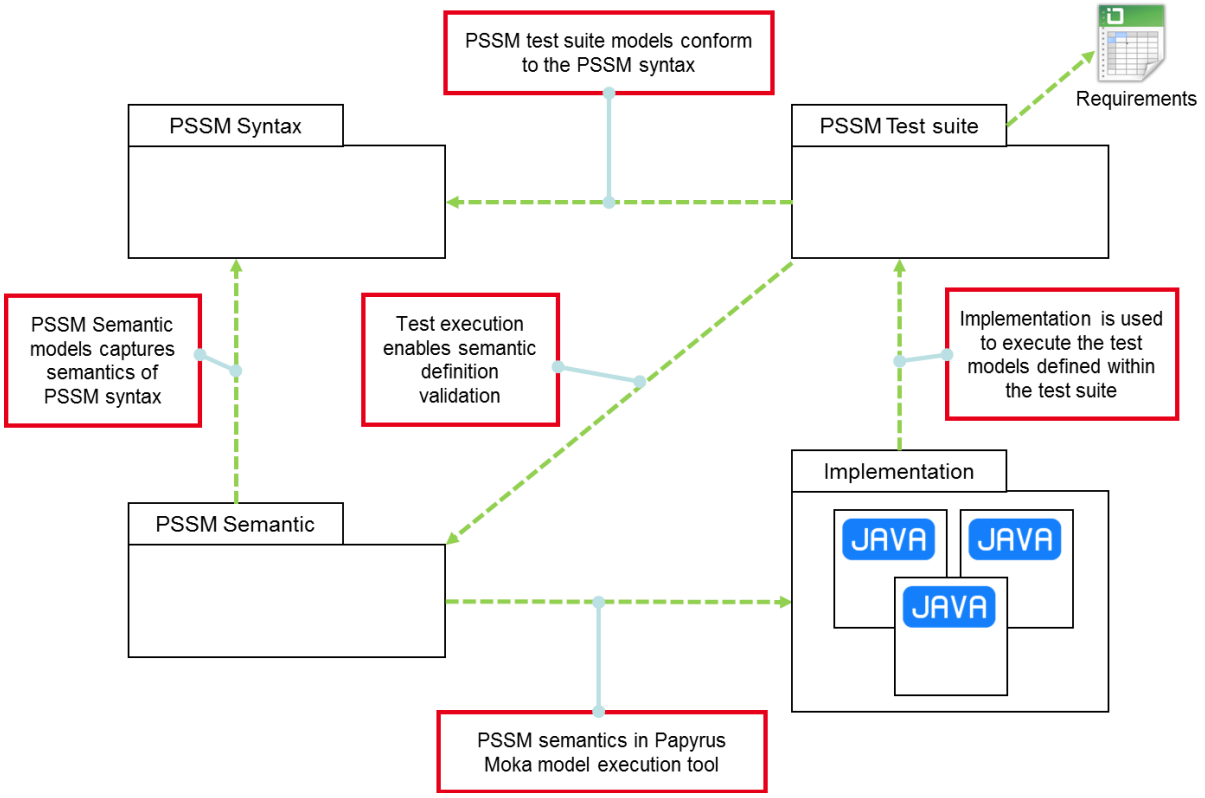


Figure 2 - PSSM specification architecture

Architecture of this specification is depicted in Figure 2. It relies on four pillars:

1. **PSSM Syntax**. The subset of UML state machines for which a precise semantics is described. This subset is a superset of fUML abstract syntax.

2. **PSSM Semantic**. The semantic model that captures the definition of the precise semantics for state machines. This model is a class model describing a set of semantic visitors and their associations, which are responsible for the definition of the precise semantics of each syntactic element included in the PSSM syntax.

3. **PSSM Test suite**. The test suite is a model describing a set of test cases. Each test case is designed to assess a particular part of the UML state machine semantics. These "parts" are requirements that have been extracted from section 14 of [1] and referenced in an excel file. PSSM test suite is a PSSM-only conformant model. Hence, it can by construction be executed using the semantics captured by the semantic model. A tool implementing the semantic model and passing all tests described in the test suite model can say it correctly capture the UML state machine semantics.

4. **Implementation**. The specification is delivered with a proof of concept implementation. This implementation is one possible implementation of the semantic model that is defined for PSSM. It is used for the purpose of executing the test suite and therefore validate that the expected semantics is correctly captured by the semantic model.

The complete development of this specification was driven by the tests and the identified requirements. The methodology consists in the following steps:

1. Select a new requirement.
2. Refine the semantic model to capture the requirement.
3. Reverberate the semantic model changes to the implementation.
4. Add a test in the test suite to demonstrate support of the requirement.
5. Execute that test using PSSM implementation.
6. Check if the trace generated by the test is included in the set of expected traces.

If the test fails then the semantic model and the implementation are refined. Conversely, if the test pass then a new requirement is selected and the same steps that those mentioned above are applied. Note that the semantics attached to each requirement was discussed in details by the PSSM submission team before proceeding to any change in the semantic model.

2.3 Specification Content

In the two previous sections, the scope of the specification was identified and the architecture of this specification was described.

This section describes the content that was included in main parts of the specification. Clause 2.3.1 gives an overview of the abstract syntax for which a semantics is provided by PSSM as well as the additional constraints that are added to the syntax. Clause 2.3.2 describes the definition of the core state machine semantic visitors and explain how these extensions are used at runtime to execute a model. Clause 2.3.3 describes the test suite architecture, the process to describe new tests and how the tests are related to the identified requirements. Finally, clause 2.3.4 explains the implementation design, the integration of this latte into Moka (Papyrus model execution platform) and the procedure to execute the PSSM test suite through this implementation.

Note: *The purpose of this section is not to fully describe the different parts of the specification. Instead, for the syntax, the semantics, the test suite and the implementation it provides a sufficient level of details to highlight the work done in the context of that task. In addition, it refers to the different part of the specification that readers have to look at if they need more detailed information.*

2.3.1 Syntax

The definition of the PSSM syntax corresponds to the selection of the UML meta-classes required to construct that subset and the addition of rules constraining usage of these meta-classes. Both aspects are described sub clauses 2.3.1.1 and 2.3.1.2.

2.3.1.1 Meta-classes

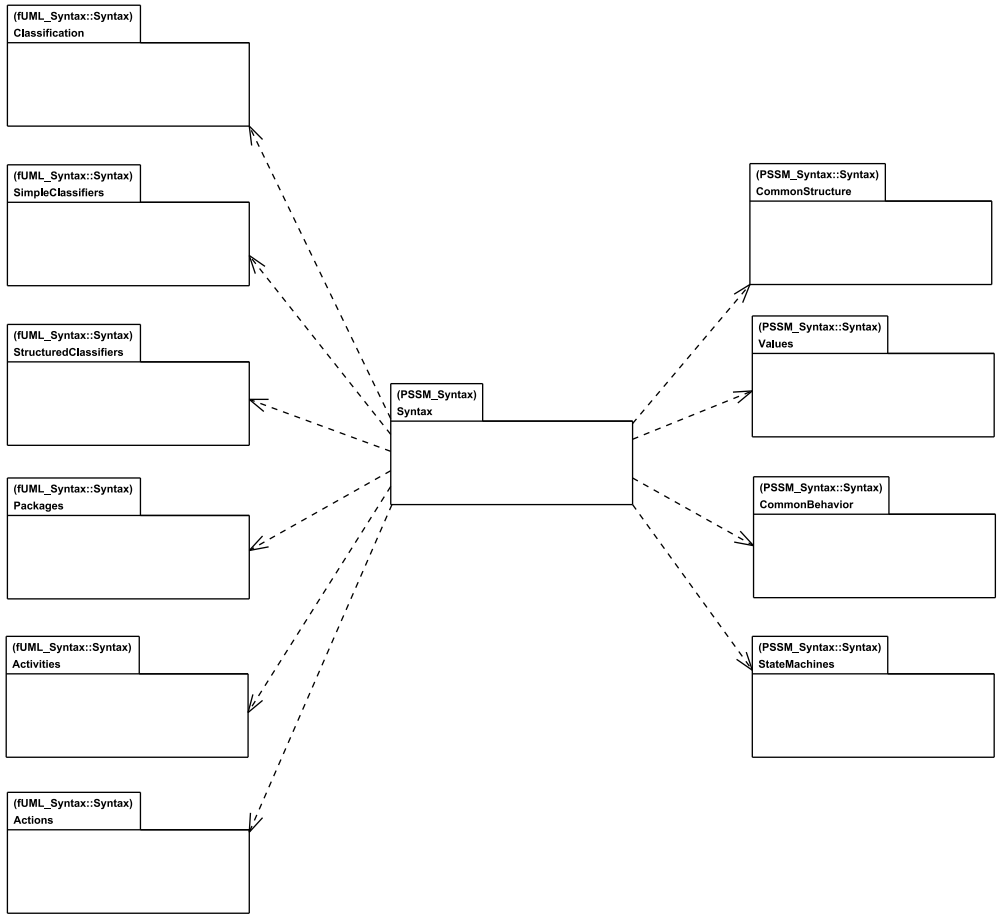


Figure 3 - PSSM Syntax Package

PSSM is based on UML 2.5. The subset of the metamodel that is covered by PSSM is captured in the package `PSSM_Syntax::Syntax` (see in Figure 3). This package imports into its namespace exactly the meta-classes included in the PSSM subset.

On the left hand side of Figure 3, all imported packages are those included meta-classes supported by the fUML subset. On the right hand side, all imported packages contain meta-classes that are specific to UML state machines. `CommonStructure`, `Values` and `CommonBehavior` are imported in addition to `StateMachines` package. One can notice that fUML already imports meta-classes available in these low level packages (`CommonStructure`, `Values` and `CommonBehavior`). However, PSSM requires some that are missing in fUML. This explains why such imports are required.

Examples justifying such imports are meta-classes `Expression` and `OpaqueExpression`. Neither meta-classes are included in the fUML subset however, PSSM requires them. Indeed, it must be possible:

1. To specify that a `Transition` is an *else* transition. This is materialized by the fact that the guard specification is an `Expression` which has no operands but its associated symbol is “else”.

2. To specify the guard of `Transition` has an `OpaqueExpression`. In such situation, the `OpaqueExpression` is always associated to a behaviour that defines its specification. This behaviour can therefore be executed if specified as an `Activity` and will provide the verdict corresponding to the guard evaluation.

`Constraint` from `CommonStructure` and `CallEvent` from `CommonBehavior` are also imported following the same approach. `Constraint` is required since a guard on `Transition` is specified as a constraint. `CallEvent` is required since synchronous operations call on active objects are allowed by PSSM. This addition makes possible for a state to declare a deferrable trigger for a `CallEvent`. Furthermore it also enables transition to be reactive to dispatched `CallEvents`.

Nevertheless, the biggest addition in terms meta-classes remains by the import of the `PSSM_Syntax::StateMachines` package.

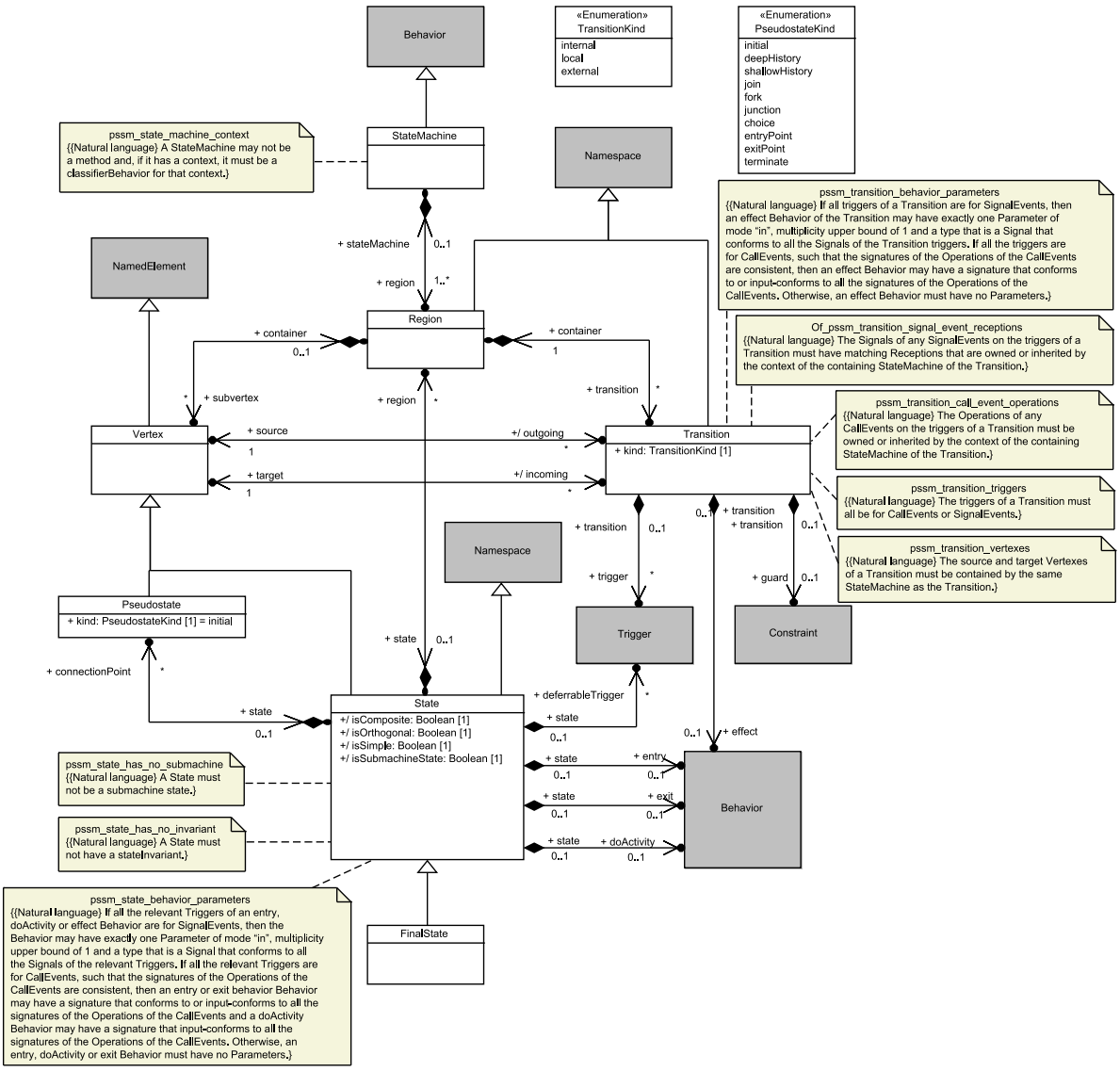


Figure 4 - Behavior State Machines

Note that the capability for StateMachine redefinition actually does not require any other meta-classes than those already included for behaviour state machines (see Figure 4).

2.3.1.2 Constraints

The package `PSSM_Syntax::Constraints` (see Figure 5) imports into its namespace all constraints applying on the PSSM subset. The approach is exactly the same than the one that was applied for the syntax. Constraints expressed for fUML subset and available in `StructuredClassifiers`, `Packages`, `Activities` and `Actions` are imported (which by construction also includes those defined in packages imported by these packages)

PSSM adds a significant number of constraints (see Figure 4). The added constraints have the role to ensure that if a particular model conforming to the PSSM subset also meets the constraints then this model can be executed using the semantics captured in the PSSM semantic model. These constraints are defined in Object Constraint Language (OCL) [8] in the specification document. These constraints can therefore directly be used by a tool implementing PSSM to validate before execution that the model is statically valid.

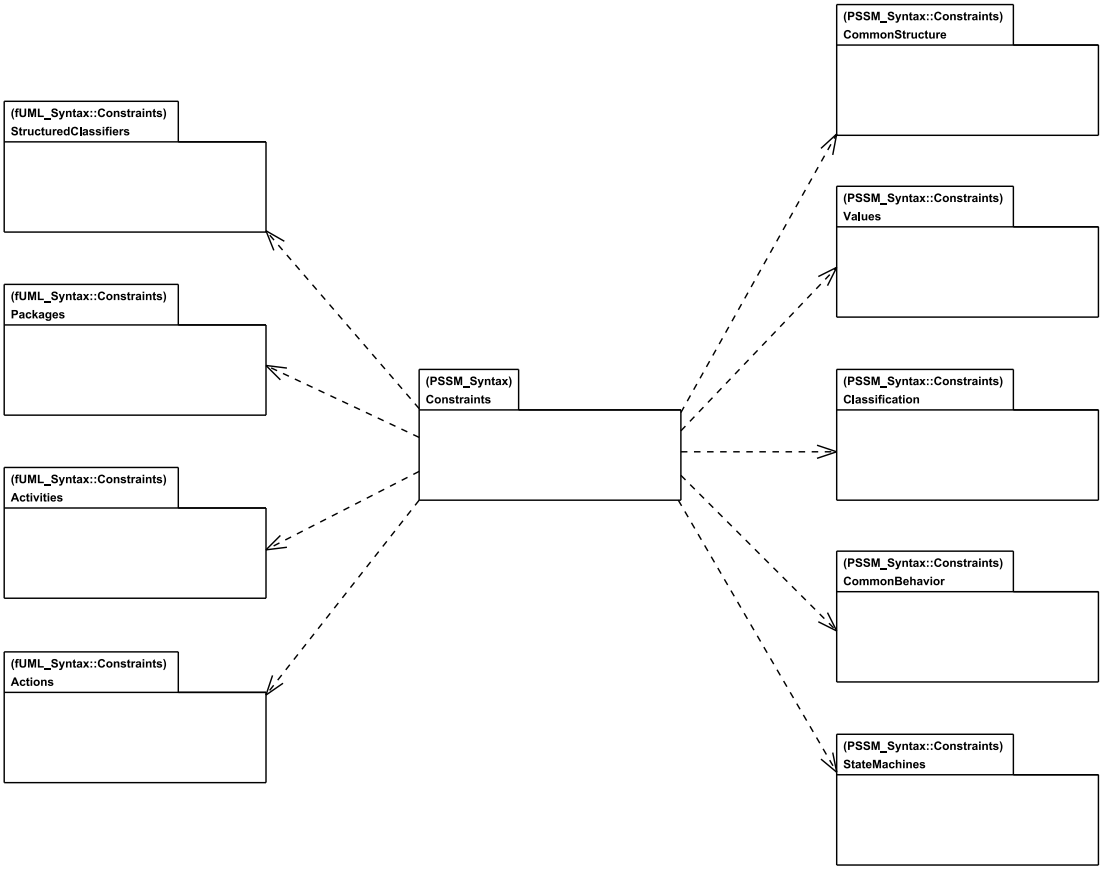


Figure 5 - PSSM Constraints

An example of constraint is:

- *A state machine may not be a method and if it has a context, it must be the classifier of that context.*

- ```
  context UML::StateMachines::StateMachine inv:
        self.specification = null and
        self.context <> null implies
        self.context.classifierBehavior = self
  ```

- The first part forbids the usage of a state machine as implementation of an `Operation`. This is due to the fact that a `StateMachine` is a `Behavior`, a behavior can have parameters but in the context of a state machine it is not clear how values associated to these parameters can be used at runtime.

- The second part of the constraint makes mandatory the fact that a state machine with a context must play the role of a *classifier behavior* for that context.

Most important constraints that are added by PSSM define the rules that signatures of `Behaviors` placed on `States` (entry/doActivity/exit), on `Transitions` and playing the role of a *specification* for an `OpaqueExpression` must conform to in order to let them have access to the data owned by the dispatched signal event occurrence and call event occurrence.

- If all the relevant `Triggers` of an entry, doActivity or effect `Behavior` are for `SignalEvents`, then the Behavior may have exactly one `Parameter` of mode "in", multiplicity upper bound of 1 and a type that is a `Signal` that conforms to all the `Signals` of the relevant `Triggers`. If all the relevant `Triggers` are for `CallEvents`, such that the signatures of the `Operations` of the `CallEvents` are consistent, then an entry or exit behavior `Behavior` may have a signature that conforms to or input-conforms to all the signatures of the `Operations` of the `CallEvents` and a doActivity `Behavior` may have a signature that input-conforms to all the signatures of the `Operations` of the `CallEvents`. Otherwise, an entry, doActivity or exit `Behavior` must have no `Parameters`.

- If all triggers of a `Transition` are for `SignalEvents`, then an effect `Behavior` of the `Transition` may have exactly one `Parameter` of mode "in", multiplicity upper bound of 1 and a type that is a `Signal` that conforms to all the `Signals` of the `Transition` triggers. If all the triggers are for `CallEvents`, such that the signatures of the `Operations` of the `CallEvents` are consistent, then an effect `Behavior` may have a signature that conforms to or input-conforms to all the signatures of the `Operations` of the `CallEvents`. Otherwise, an effect `Behavior` must have no `Parameters`.

2.3.2 Semantics

The previous section presented the definition of the PSSM subset and the addition of constraints for meta-classes included in this latter. The objective here is not to provide a detailed overview of all extensions defined in the specification. This section rather focus on the description of
  ```

core extensions defined by the PSSM semantic model. It provides the rationale for defining these extensions and explain their roles.

This section is organized as follows. Clause 2.3.2.1 reminds the design patterns and principle driving the construction of a semantic model. Clause 2.3.2.2 describes the root element for specifying the execution semantics of state machines. Clause 2.3.2.5 explains what a state machine configuration is and how this concept is used to determine the impact of the dispatching of an event on the state machine. Finally, Clause 2.3.2.6 describes main semantic visitors defined for capturing the general semantics of vertices, transitions and regions.

2.3.2.1 Semantic Model Definition Principles

A semantic model is a class model whose role is to capture through the structure and the defined operations the semantics of a well identified subset of the UML syntax. Such model conforms to the fUML subset. Hence by construction it is executable and its semantics is provided by the one defined in fUML.

Elements defined in the semantic model can be classified into three categories:

1. **Values**. A value is the representation of an instance of a type. As an example, fUML defines `Object_` which is a specific type of value that enables the representation at runtime the instance of a `Class`.
2. **Visitors**. A visitor captures the semantics of a particular meta-class. As an example, fUML defines `AcceptEventActionActivation`, which is a specific semantic visitor capturing semantics of an `AcceptEventAction`.
3. **Others**. All elements defined in the semantic model which are not values or visitors. These elements usually capture internal logic of some visitors or are responsible to instantiate the semantic visitors. As an example, fUML defines an `ExecutionFactory` which is in charge of instantiating semantic visitors defined for meta-classes included in the fUML subset.

Semantic models for fUML and PSCS have been designed using the aforementioned principles. PSSM semantic model has been designed in a similar manner. Next section describes the root element for specifying the execution semantics of state machines.

2.3.2.2 State Machine Execution

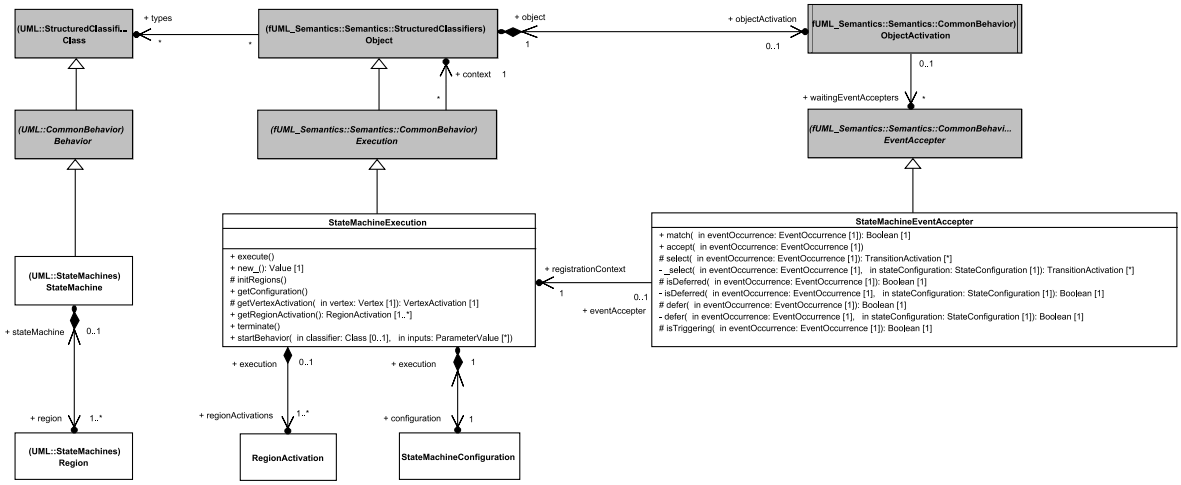


Figure 6 - State Machine Execution

`StateMachine` is a specialization of `Behavior` (see Figure 6). In order to capture the execution semantic of a `Behavior` the fUML semantic model provides the concept of `Execution`. This concept is by the way specialized by `ActivityExecution` whose role is to capture the execution semantics of an `Activity`.

The principal is similar to capture the execution semantic of a state machine. Hence a `StateMachineExecution` class is defined. This class is a specialization of `Execution`. The description of the dynamic corresponding to the execution of state machine is captured by overriding the abstract operation `execute` provided by the `Execution` class.

2.3.2.2.1 Execution start-up

The execution of a state machine starts when the `execute` operation is called on a `StateMachineExecution`. This call always occurs during the initial RTC step of the state machine. Indeed PSSM only defines semantics for state machine, which are active or state machine playing the *classifier behaviour* role.

- The first phase of the execution consists in instantiating visitors for all regions owned by the executed state machine. These visitors are `RegionActivation` and capture the execution semantics of `Regions`. Each activation instantiated for a region then create (in cascade) semantic visitors for all their contained elements. At the end of the instantiation phase, the execution for a state machine is the root element of tree like structure including all of the created semantic visitors.

- The second phase consists in concurrently proceed to the entering of each `Region`. Entered `Regions` are required to have an initial `Pseudostate`. If a `Region` has no such `Pseudostate` then it is ignored by the execution. The initial RTC step ends when the state machine has reached a stable configuration. This occurs when it exists no `Transition` available for firing and all entry `Behaviors` of entered states have completed their execution.

2.3.2.3 Execution and State Machine Event Accepter

Each evolution (i.e. move from the current configuration to the next one) of the state machine configuration is realized in a RTC step. A step is triggered by the fact that a state machine can accept the event that is dispatched (i.e., removed from the event pool). In order to allow a state machine to accept an event occurrence this latter must have registered an `EventAccepter`.

The `EventAccepter` concept is defined in fUML. It is specialized by activities to define an `ActivityEventAccepter`. The principle in activities is that the execution of an activity can suspends on `AcceptEventAction`. The semantics defined in fUML implies that an `ActivityEventAccepter` is registered when the action gets executed. Hence when an event will be dispatched, it may enable the trigger declared by the `AcceptEventAction`. In such situation the accepter is said to *match* the dispatched event. The execution then restarts from the action that registered the accepter.

PSSM defines a specialization of `EventAccepter`: `StateMachineEventAccepter`. There two fundamental differences between this type of accepter and the one defined for activities in fUML.

1. Conversely to activities where each `AcceptEventAction` register an accepter, a state machine always has a single state machine event accepter registered. The reason for this is that to determine how a state machine can respond to event a complete analysis of the current state machine configuration is required. Hence it is not possible to have separate event accepters for each individual transition.

2. The logic of matching and accepting a dispatched event is strongly different. Indeed, in activities to say a registered event accepter matches a dispatched event it is sufficient that the accept event action that registered the accepter declares a trigger for an event that matches the type of the dispatched event. In the case of a state machine, an event is said to match if in the current configuration the event can be deferred or it triggers one or more transitions outgoing states registered in the configuration. The verdict of the `match` operation is computed by analysing the overall state machine configuration. This analysis account for priority rules existing between `Transitions`, conflict resolutions, static analysis of paths leading to the next state machine configuration, etc.

If the dispatched event occurrence is deferred in the current state machine configuration then this latter is accepted and placed in the deferred event pool. The deferred event occurrence will only be released (i.e., returned to the regular event pool) when the state that provoked is deferral leaves the state machine configuration.

If the dispatched event is not deferred and triggers one or more transition in the state machine then this latter is also accepted. The acceptance of the event implies the functionality of the semantic visitors associated with various elements of the state machine to be executed. Execution of semantics related to these visitors lead the state machine to enter a new stable configuration.

It is important to note that this latter case only occur if all `Transitions` that are selected to fire lead the state machine to a valid state machine configuration. To ensure this, a static analysis

is always performed to check if the traversal of a `Transition` (possibly compound) leads to enter a valid state machine configuration.

2.3.2.4 Execution Completion vs Execution Termination

A state machine execution completes when all `Regions` owned by the executed state machine have completed. A `Region` is said to have completed its execution when the final state owned by that `Region` is reached.

A state machine execution can also be terminated. This situation is different from the completion. Indeed the termination of the execution of state machine is due to the execution of a terminate `Pseudostate`. When such pseudo state is exited the execution state machine stops immediately (i.e., no behaviours are executed in response to the termination), all visitors instantiated for the state machine are destroyed and the execution context of the state machine is destroyed.

Next section explains what the state machine configuration is and how this latter is used to determine the response of an executed state machine to a dispatched event.

2.3.2.5 State Machine Configuration

A state machine configuration is the representation of the hierarchy of active `States` of an executed state machine. The state machine configuration is stable before and after a RTC step but not during a RTC step. When an event is accepted by the state machine and it triggers or more transitions it always implies to move from the current state machine configuration to another state machine configuration. Obviously, the target state machine configuration can be the same than the source state machine configuration (e.g., case of self-`Transition`).

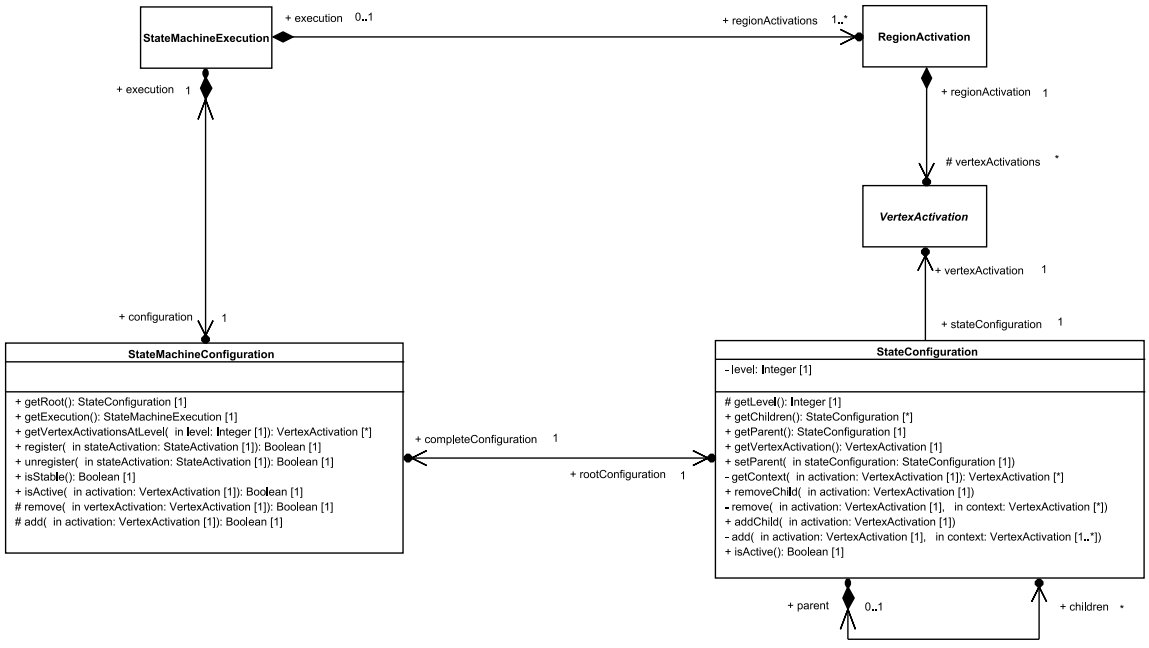


Figure 7 - State Machine Configuration

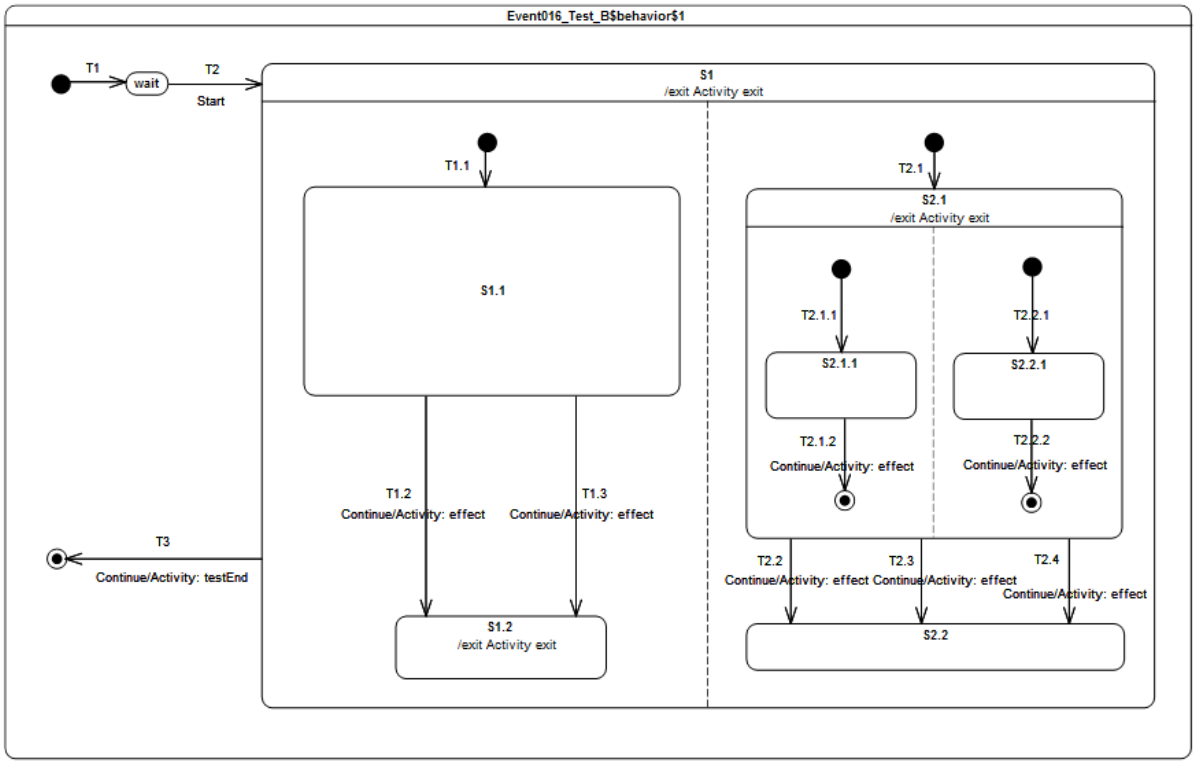


Figure 8 - Test Event 016 B

PSSM made the choice to explicitly represent during the execution of state machine. It provides for this `StateMachineConfiguration` and `StateConfiguration` concepts (see Figure 7). These two classes directly enter the "*Others*" category discussed in clause 2.3.2.1. Indeed there are neither classes defined to represent values or state machine semantic visitors. The `StateMachineConfiguration` class represents the overall state machine configuration. It references a `StateConfiguration`, which represents the root element of the state machine configuration. This `StateConfiguration` itself references a set of `StateConfiguration` that materialize active states located in different regions owned by the executed state machine.

Consider the state machine presented in Figure 8. Under the assumption that this state machine has already performed its initial RTC step, it is in configuration *wait*. *Wait* is the active state of the state machine. The configuration would be described as presented in Figure 9 by the PSSM semantic model. The root state configuration (i.e., abstraction of the state machine) has only a single child state configuration. This is perfectly fine since the executed state machine has only one region and in this region the simple state *wait* is active.

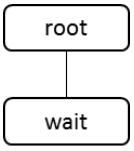


Figure 9 - State machine configuration after the initial RTC step

When the Start event is dispatched the state machine configuration is evaluated. The verdict of that evaluation is that the state machine can accept the event and this latter will trigger the compound transition T2(T1.1, T2.1(T2.1.1, T2.2.2.1)) leading to reach the state machine configuration described in Figure 10.

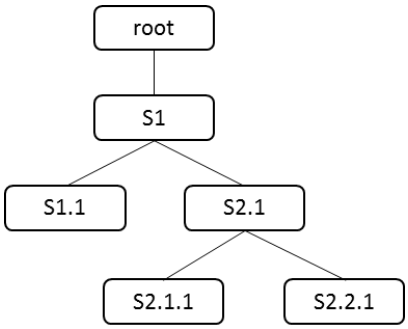


Figure 10 - State machine configuration after the step initiated by Start

When the Continue event is dispatched the state machine configuration is evaluated. The evaluation starts from the innermost active states (i.e., leaf state). After having evaluated S2.1.1 and S2.2.1 two transitions are included in the set of fireable transitions: T2.1.2 and T2.2.2. When S2.1 is evaluated one can notice that it has a transition that may fire using the Continue event. However transitions with a higher priority have already been included to the set of fireable transitions. Hence no transition outgoing S2.1 and reactive to a Continue event can be included to the set. At the end of the analysis the set of fireable transitions contains T1.2, T2.1.2 and T2.2.2. All of these transitions will be fired concurrently in the next RTC step. Note that conflicts between transitions are resolved during the evaluation of the state machine configuration thanks to a semantic strategy that is provided in fUML.

In short, a `StateMachineExecution` is always associated to a `StateMachineConfiguration`. The state machine configuration captures a sufficient abstraction of the executed state machine to enable the computation of a verdict regarding what the state machine shall do when an event is dispatched. Next section, describes the core state machine semantic visitor defined in PSSM.

2.3.2.6 State Machine Semantic Visitors

PSSM defines four core state machine semantic visitors. These visitors are listed below:

1. StateMachineSemanticVisitor – see clause 2.3.2.6.1
2. VertexActivation – see clause 2.3.2.6.3
3. TransitionActivation - see clause 2.3.2.6.4
4. RegionActivation – see clause 2.3.2.6.2

fUML provides the concept of `SemanticVisitor`. This concept is specialized by PSSM as a `StateMachineSemanticVisitor` (see Figure 11). A state machine semantic visitor is the common type of all visitors defined for state machine elements. It adds three elements to the `SemanticVisitor` concept provided by fUML:

Figure 11 - State Machine Semantic Visitors

2.3.2.6.1 StateMachineSemanticVisitor

1. A `StateMachineSemanticVisitor` is systematically associated with a `NamedElement`. The named element which is referenced by this type of visitor is always a state machine element.

2. A `StateMachineSemanticVisitor` is systematically associated with its parent `SemanticVisitor`. The parent can be a `StateMachineExecution` or another `StateMachineSemanticVisitor`. The overall set of state machine semantic visitors instantiated to execute a state machine are organized as a tree-like structure. This structure reflects the structure of the state machine.

3. The `StateMachineSemanticVisitor` class provides a set of utility operations that can be used or shall be implemented by all concrete specializations (e.g. concrete visitors `StatActivation` that captures the execution semantics of the `State` concept). As examples:

    a. `getExecutionContext` – returns the context object of the root state machine execution

b. `activate` and `activeTransition` – intended to be overridden by the specializations. Each specialization specifies the appropriate logic for instantiating visitors for state machine elements it contains.

This base `StateMachineSemanticVisitor` is specialized by `VertexActivation`, `TransitionActivation` and `RegionActivation` which are presented in next clauses.

2.3.2.6.2 Region Activation

A `RegionActivation` is a specialization of a `StateMachineSemanticVisitor`. It captures semantics of `Region`. Hence the node referenced by a `RegionActivation` is always a `Region`.

Semantics of a `Region` are mainly captured in `enter` and `exit` operations provided in the `RegionActivation` class. Note that these operations are only called if the `Region` is either entered or exited *implicitly*. An implicit entry occurs in the following situations:

1. The `Region` is owned by the `StateMachine`.

2. The `Region` is owned by a `State` and a `Transition` targets the edge of the `State` that contains that `Region`.

3. The `Region` is owned by a `State` and the `Transition` targets an entry point with no continuation `Transition`. The entry point must be owned by the `State` that contains the entered `Region`.

Execution of a `Region` that is entered implicitly always starts from its *initial* `Pseudostate`. The single outgoing `Transition` of the *initial* `Pseudostate` is traversed immediately and its target is entered. In the case where no *initial* `Pseudostate` is available the *Region* is ignored. The consequence might be that a composite `State` is treated a simple `State` or a `StateMachine` execution terminates immediately since it only has single `Region` and this latter has no initial `Pseudostate`.

An implicit exit of one or more `Regions` occurs when a `Transition` exits a composite `State`. In that case, all `Regions` concurrently executed in that `State` are exited. This implies active `State` of each `Region` are exited. The exiting sequence for each `Region` starts by exiting the most nested active `States`.

Explicit `entry` and `exit` of `Region` is also supported in PSSM. An *explicit* entry occurs when a `Region` is entered via a `Transition` with a source outside the `Region` and target inside the `Region`. Conversely, an *explicit* exit of `Region` occurs when a `Transition` exits a `Vertex` located in the `Region` and targets a `Vertex` located outside the `Region`.

When a `Region` is entered explicitly, the execution of this latter does not start from the *initial* `Pseudostate`. The `Region` is considered as being entered when the target `Vertex` located inside the `Region` is entered. Note that if the `Region` is located in a composite `State`

owning other Regions which are not entered explicitly then these latter are entered using the *implicit* entry semantics.

When a Region is exited explicitly, then the source Vertex is exited (if the source is a State that is composite then active States located within are exited). If the Region that is exited is owned by a composite State that has additional Regions then these latter are exited using the *implicit* exit semantics.

2.3.2.6.3 VertexActivation and the StateActivation Specialization

The state machine semantic visitor VertexActivation is abstract and captures the common semantics of each kind of Vertex (i.e., State, FinalState and all kind of Pseudostate). Hence the node referenced by a VertexActivation is always a Vertex.

VertexActivation defines the common way to enter any kind of Vertex. To do so it provides the operations enter and exit.

1. A Vertex can only be entered if its prerequisites (specific to each kind of Vertex, based on the redefinition of the VertexActivation isEnterable operation) have been fulfilled. In this case, and only in this case, the VertexActivation can be entered (using the enter operation). The entry semantics are specific to each kind of Vertex. Nevertheless, each specialized Vertex is entered using a given *entering Transition* and knows about the common ancestor it shares with the source VertexActivation. The entered VertexActivation always takes advantage of the common ancestor (a RegionActivation) information to identify if the parent VertexActivation must also be entered before.
2. A Vertex can only be exited if its prerequisites (specific to each kind of Vertex, based on its redefinition of the VertexActivation isExitable operation) have been fulfilled. In this case, and only this case, can the VertexActivation be exited (using the exit operation). The exit semantics are specific to each kind of Vertex. Nevertheless, each specialized Vertex is exited using a given *exiting Transition* and knows about the common ancestor it shares with the target VertexActivation. The exited VertexActivation always takes advantage of the common ancestor (a RegionActivation) information to identify if the parent VertexActivation must be exited after.

PSSM defines StateActivation which specializes VertexActivation in order to capture the semantics of the State meta-class. This specialization defines the specific entry and exit semantics of a State by completing the initial semantics defined in VertexActivation.

1. The only prerequisite a State must fulfill to be entered is that it must not be already active (i.e., it must not be part of the state machine configuration). If it can be entered, then the common ancestor rule described by the semantics captured in VertexActivation applies. As soon as this rule was applied, the remaining part of

the entry sequence is specific to a State. This sequence consists in referencing the State in the state machine configuration, execute its entry Behavior (if any), invoke (it will be executed asynchronously from the state machine) its doActivity (if any) and enter concurrently its owned Regions (if the State is composite).7

2. The only prerequisite a State must fulfill to be exited is that it must already be active (i.e., it must be part of the state machine configuration). If it can be exited, then the exit sequence that is specific to a State is performed. This sequence consists in exiting all regions owned by that State (if it is composite), aborting the doActivity (if any) that has been invoked from State, updating the history of the Region owning that State and removing the State from the state machine configuration. As soon as the exit sequence specific to State was performed then the common ancestor rule described by the semantics captured in VertexActivation applies.

2.3.2.6.4 TransitionActivation and the ExternalTransitionActivation specialization

The state machine semantic visitor TransitionActivation is abstract and captures the common semantics of all kind of Transition: *local*, *internal* and *external*. This common semantics is especially captured by the *fire* operation defined in this visitor. The firing sequence consists in exiting the source Vertex, executing the effect behaviour (if any) and entering the target Vertex. Although the firing sequence is common to all kind of Transition, this is not the case of the sequences that respectively define what the impact of exiting the source and entering the target is. These sequences are specific to the different kind of Transition. To capture this, the semantic model defines three specializations of TransitionActivation.

ExternalTransitionActivation is one of the defined specialization. It captures the semantics of an *external* Transition. To do so it redefines the operation exitSource and enterTarget declared by TransitionActivation.

1. exitSource. It is only possible to exit the source Vertex using an external Transition if the prerequisites to exit the Vertex are fulfilled. Note that these prerequisites are specific to each Vertex. In the situation where the prerequisites to exit the source are fulfilled as well as those to enter the target, then the common ancestor existing between the source and the target is taken into account to perform the exit sequence of the source. Conversely, if the target is not ready to be entered (i.e., the prerequisites are not satisfied) then the source is exited, but the exit sequence does not account for the common ancestor existing between the source and the target.

2. enterTarget. The general case occurs when the target Vertex can be entered. Its entering sequence is executed and this latter accounts for the common ancestor existing between the source and the target. It also exists a specific usage of external transition that implies a different semantics than the one presented for the general case. Consider the situation where an *external* Transition source is an internal (maybe deeply nested) Vertex of the target. This assumption implies that the target is a composite State and the Transition ends it *inside* edge. In this situation, the target cannot be entered since it is already part of the state machine configuration. Nevertheless the Region owned by the target State and which owns (directly or indirectly) the source

`Vertex` will be considered as being completed. Hence if this `Region` is the only one owned by the target `State` then this latter completes.

To summarize, the firing of a `Tansition` can be viewed as chain of calls. These calls represent the different sequence actions realized during the firing of the `Transition`. A RTC step can be represented by a single chain of calls or set of chain of calls in the case where the dispatched event implies the concurrent firing of multiple `Transitions`.

```
fire()
      exit(exitingTransition, eventOccurrence, commonAncestor)
            exit(exitingTransition, eventOccurrence, commonAncestor)
                  …
      executeEffect(eventOccurrence)
      enter(enteringTransition, eventOccurrence, commonAncestor)
            enter(enteringTransition, eventOccurrence, commonAncestor)
                  …
End
```

Clauses 2.3.2.2, 2.3.2.3 and 2.3.2.6 provided the reader with an overview of PSSM core semantic element. Next section describes the test suite structure, some tests and their relationships to identified semantic requirements.

2.3.3 Test Suite and Semantic Requirements Coverage

Design of the PSSM semantic model was driven by semantic requirements that were identified for UML state machines and testing of the validity of the implemented requirements. Clause 2.3.3.1 describes the role of the test suite model. Clause 2.3.3.2 explain how test cases are described in this test suite. Finally clause 2.3.3.3 proposes a review of few test cases provided in the test suite.

2.3.3.1 Test Suite Role

The test suite has three roles:

1. It defines a reference model that when executed provides a way to assess that requirements implementation in the semantic model captures exactly the semantics intended by the UML specification.

2. It defines a reference model that a tool trying to implement PSSM semantics can use to evaluate if its implementation is effectively PSSM conformant. To claim such conformance a tool must pass all tests defined in the test suite.

3. It provides a way to ensure that no regressions are introduced in new increments introduced in the semantic model. Indeed each time a new part of the semantics is described, all tests already defined in the test suite must still be passed.

2.3.3.2 Test Suite Architecture

The test suite model is separated in two distinct parts. The first part corresponds to the definition of the abstract architecture of a test case. The second part is a set of packages where each

package refers to a particular test category. For example, one test category in the test suite captures all tests related to transition semantics. Each test case in this category assert a specific part of the transition semantics. This part has been previously identified in semantic requirement.

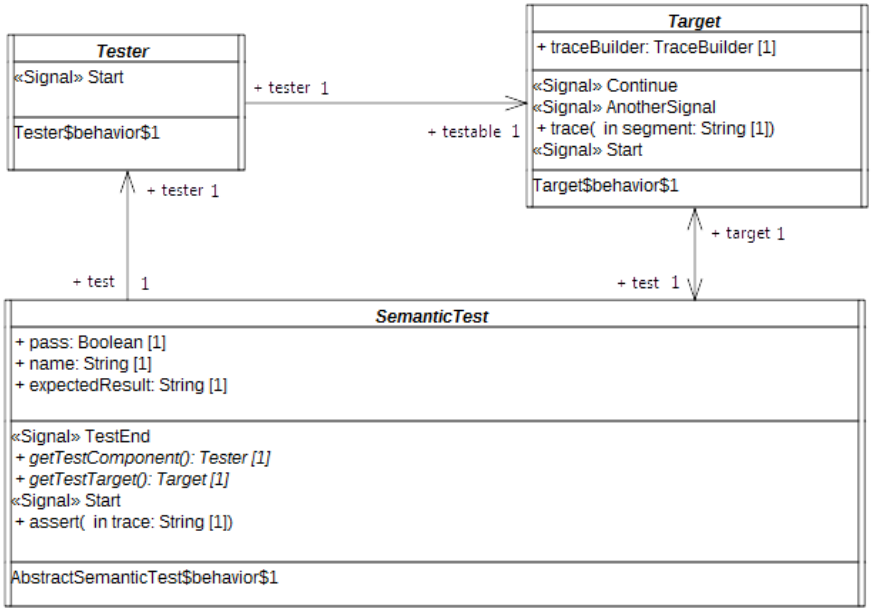


Figure 12 - Test Suite Architecture

Figure 12 shows the abstract architecture of a test cases. This architecture is composed of three elements:

1. **Tester**. The tester is an abstract active class which encodes in its classifier behaviour the stimulation sequence (i.e., a set of events) that will be sent to the test target. The classifier behaviour provided by the abstract tester is empty. Specializations of this class are intended to provide a new classifier behaviour describing the user defined stimulation sequence that must be sent.

2. **Target**. The target is an abstract active class. It receives the stimulation sequence sent by the Tester. The received events will enable the classifier behaviour to realize RTC steps. Throughout its execution the classifier behaviour generates an execution trace. This trace is stored by the target (see `traceBuilder` in class Target in Figure 12). The classifier behaviour of the abstract target is empty. Specialization of this class are intended to provide a new classifier behaviour. In the context of PSSM the classifier behaviour shall be a state machine.

3. **SemanticTest**. The semantic test acts as a controller for the tester and the target. It takes into account the trace generated by the target and use it to compute the test verdict. The verdict is either *pass* or *fail*. If the verdict is *pass* then it means the trace generated by the target is included in the set of expected traces for the test. Conversely, if the verdict is *fail* then the generated trace is not included in the set of expected traces. Note that specializations of this class are not intended to provide a new classifier behaviour.

Each tests declared within the PSSM test suite extends this abstract architecture. Some of the tests defined in the test suite model are described in the next section.

2.3.3.3 Tests

Clause 2.3.3.3.1 explains how test description is build / organized in PSSM specification. Finally clauses 2.3.3.3.2 and 2.3.3.3.3 explain in details two PSSM tests. These tests rely only on semantic functionalities described in section 2.3.2.

2.3.3.3.1 Tests Description

All test are descriptions follow the exact same pattern.

1. The state machine under test is presented.
2. The stimulation sequence received by the state machine is described.
3. A trace that can be generated by the test is described.
4. Test purpose is reminded and the execution corresponding to the trace is described.
5. A table describing all RTC steps realized during the execution is provided.
6. If alternative execution traces can be generated, then these latter are listed.

Test described in clauses 2.3.3.3.2 and 2.3.3.3.3 will be described according to this pattern.

2.3.3.3.2 Test 1: Transition 007

Tested state machine

The state machine that is executed for this test is presented in Figure 13.

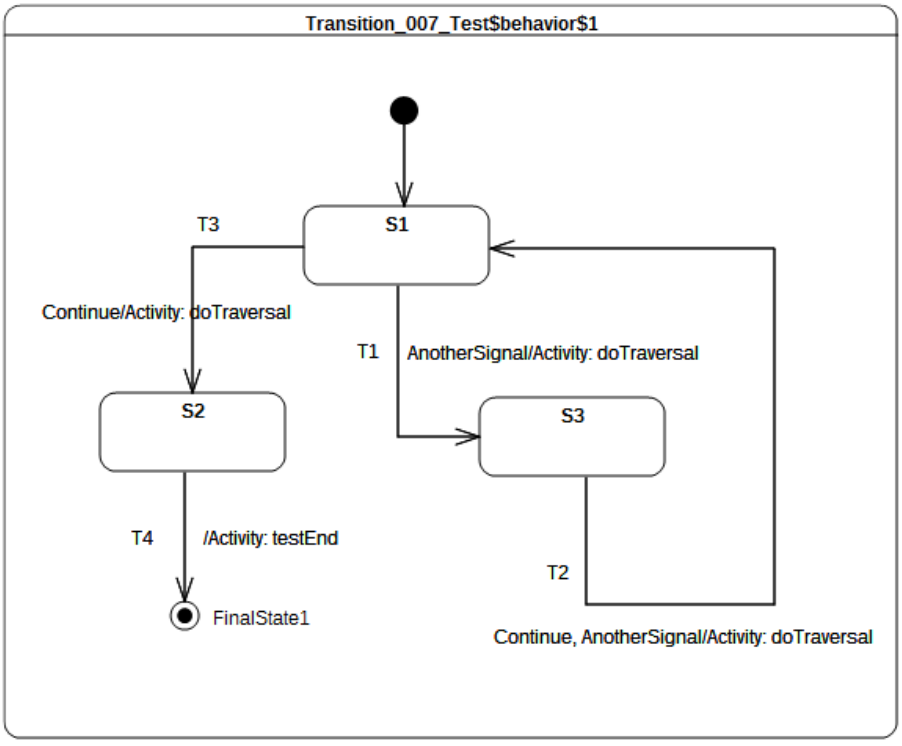


Figure 13 - Transition 007 Classifier Behavior

Test execution

Received event occurrence(s)

- AnotherSignal – received when in configuration *S1*.
- Continue – received when in configuration *S3*.
- Continue – received when in configuration *S1*.

Generated trace

- T1(effect)::T2(effect)::T3(effect)

  Note. The purpose of this test is to validate that "*A transition may own a set of triggers, each of which specifies an Event whose occurrence, when dispatched, may trigger traversal of the Transition*" (see section 14.2.3.8 of [1]). Consider the state machine presented in Figure 13 has already performed its initial RTC step. The current state machine configuration is *S1*. When the completion event generated for that state is dispatched it is lost since S1 does not have a completion transition. The next event to be dispatched is AnotherSignal. It triggers T1 which has declared a trigger for this event type and S3 is entered. As S3 does not have a completion transition, its completion event is lost. When Continue is dispatched T2 is triggered. This is fine since T2 declares triggers both for Continue and AnotherSignal. The state machine at the end of the steps is back in configuration *S1*. The completion event generated for that state is lost. The next event to be dispatched is a Continue evet occurrence. It triggers T3 and S2 is entered. The completion event of S2 is used to trigger in the next RTC the transition T4. When the final state is reached the region completes which implies that the state machine execution also completes.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[InitialTransition]
2	[AnotherSignal, **CE(S1)**]	[S1]	[]
3	[**AnotherSignal**]	[S1]	[T1]
4	[Continue, **CE(S3)**]	[S3]	[]
5	[**Continue**]	[S3]	[T2]
6	[Continue, **CE(S1)**]	[S1]	[]
7	[**Continue**]	[S1]	[T3]
8	[**CE(S2)**]	[S2]	[T4]

2.3.3.3.3 Test 2: Exiting 003

Tested state machine

The state machine that is executed for this test is presented in Figure 14.

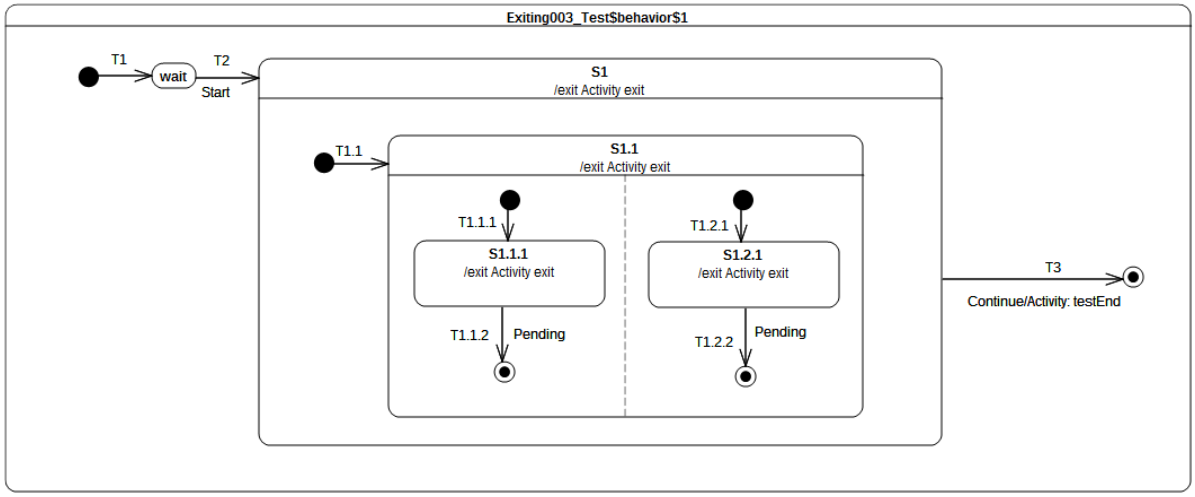


Figure 14 - Exit 003 Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1[S1.1[S1.1.1, S1.2.1]]*.

Generated trace

- S1.1.1(exit)::S1.2.1(exit)::S1.1(exit)::S1(exit)

  Note. The purpose of this test is to demonstrate that "*When exiting from a composite state, exit commences with the innermost state in the active state configuration. This means that exit behaviors are executed in sequence starting with the innermost active state*" (see section 14.2.3.4.6 of [1]). Consider the state machine presented in Figure 14 has already realized its initial RTC step. The current state machine configuration is *wait*. The completion event generated for the wait state is lost since it has no completion transition. The next event to be dispatched is Start. It triggers the compound transition T2(T1.1(T1.1.1, T1.2.1)). At the end of the of the RTC step the new state machine configuration is S1[S1.1[S1.1.1, S1.2.1]]. Completion events generated by S1.1.1 and S1.2.1 are both lost. The RTC initiated by the dispatching of the Continue event has a great impact on the state machine configuration. Indeed, although the S1 is left by T3, the semantics requires that before S1 is actually exited, its complete hierarchy of active state must be exited. The exit sequence starts with the innermost active states so S1.1.1 and S1.2.1. Here consider that S1.1.1 exit behaviour is executed before the S1.2.1 exit behaviour. A soon as both have been exited, S1.1 exit behaviour can be executed followed by S1 exit behaviour. When the final state is reached the state machine execution completes.

RTC steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T1.1(T1.1.1, T1.2.1))]
4	[Continue, CE(S1.2.1), **CE(S1.1.1)**]	[S1[S1.1[S1.1.1, S1.2.1]]]	[]
5	[Continue, **CE(S1.2.1)**]	[S1[S1.1[S1.1.1, S1.2.1]]]	[]
6	[**Continue**]	[S1[S1.1[S1.1.1, S1.2.1]]]	[T3]

Alternative execution steps

The presence of orthogonal regions in S1.1 enables the possibility to observe an alternative execution trace for that test. This trace captures the situation in which the S1.2.1 exit behaviour gets executed before S1.1.1 exit behaviour.

- S1.2.1(exit)::S1.1.1(exit)::S1.1(exit)::S1(exit)

2.3.4 Implementation

An implementation of the PSSM semantic model was developed. This implementation is integrated as an execution engine in Moka (the Papyrus model execution platform)[10]. It can be retrieved from the branch *bugs/465888-SMExecPrototype* available at the Papyrus repository[11]. The implementation is capable of executing all tests that are described in the PSSM test suite model. The process of setting up the environment to make possible the test suite execution is described below.

1. Download the Mars version of Eclipse
2. Install Papyrus Mars
   - http://download.eclipse.org/modeling/mdt/papyrus/updates/releases/mars
3. Check out the Papyrus repository
   - git clone https://git.eclipse.org/r/papyrus/org.eclipse.papyrus
4. Switch to the branch *bugs/465888-SMExecPrototype*
5. In your working directory move to *extraplugins/moka* and import the plugin
   - org.eclipse.papyrus.moka.fuml.statemachines

[10] https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution
[11] https://git.eclipse.org/c/papyrus/org.eclipse.papyrus.git/

6. Launch an Eclipse runtime

7. In the runtime instance import the PSSM test suite. This test suite is available under *resources/tests* folder of the *oep.moka.fuml.statemachines* plugin.

8. Go to *Window* menu and select the *Preferences* item. A popup opens. In that popup you must go to *Papyrus* category and select *Moka* item. In the displayed preference page make sure that "StateMachines semantics [prototype]" execution engine is selected.

9. Create a new Moka launch configuration. This configuration must be similar to the one shown in Figure 15.

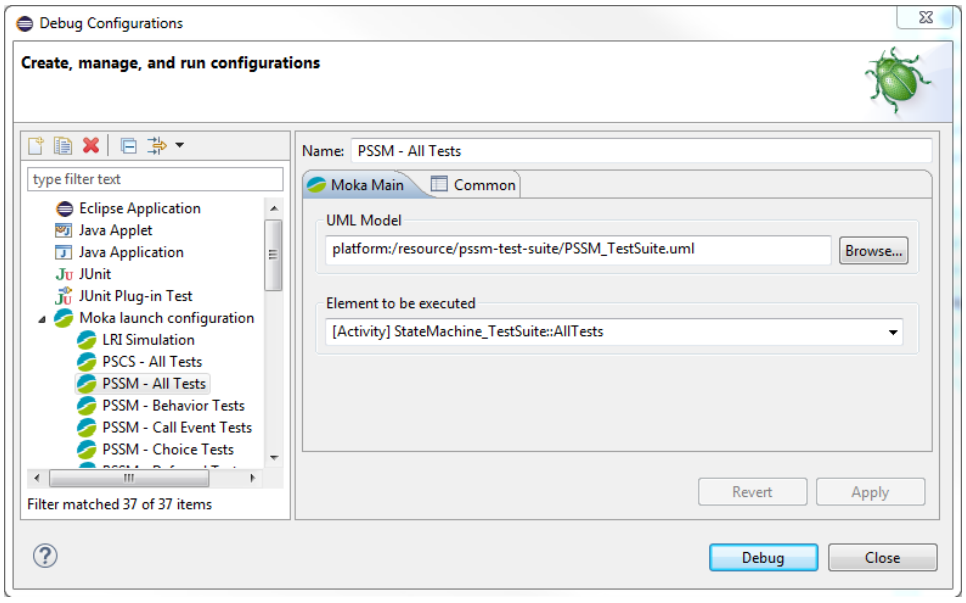


Figure 15 - Run all tests

10. Click on the *Debug* button to launch the execution of all tests defined in the PSSM test suite. By looking at the Moka console during the execution you shall observe that all tests pass.

Note: The current version of this execution engine does not provide a connection with Moka animation and debug framework. Hence the execution cannot be observed on the state machine diagrams nor suspended for debug. State machines execution engine will only be integrated in the official release of Moka when PSSM will have completed its finalization process.

2.3.5 Specification Status

The current version of the PSSM specification (i.e., Abstract syntax, semantic model, test suite and implementation) provides support for all mandatory requirements (see clause 2.3.5.1) defined in the PSSM RFP and some of the non-mandatory requirements (see clause 2.3.5.2).

2.3.5.1 Mandatory requirements

Requirement	Response
6.5.1a Behavior state machine semantics (excluding redefinition and submachines)	Support for all required elements and Call events for synchronous operation call.
6.5.1b Event data passing	Achieved using a parameter-passing approach for guard expressions and event behaviours (see clause 2.3.1.2).
6.5.1c Standalone and classifier behaviour execution.	Strict support for these two cases.
6.5.1d Consistency with PSCS	Achieved by defining PSSM execution model as an extension of the PSCS model.
6.5.1e Relationship to fUML	Achieved by defining PSSM execution model as an extension of the fUML model.
6.5.1f Extension of fUML base semantics (if necessary)	This was not found to be necessary.
6.5.2a Semantic variabilities	Proposal does not define any additional semantic variabilities.
6.5.2b Semantic variants	Proposal does not define any additional semantic variants.
6.5.3a UML 2 conformance	Proposed PSSM syntax subsets UML 2.5 abstract syntax.
6.5.3b fUML conformance	Proposed PSSM semantics are based on fUML 1.2.1.
6.5.3c PSCS conformance	Proposed PSSM semantics are based on PSCS 1.0

6.5.3d Common Logic conformance	Not applicable, proposal does not extend the fUML base semantics.
6.5.4a Test suite	Proposal includes suite of 105 tests.
6.5.4b Test suite coverage	Current coverage is 100% of identified functional requirements.

2.3.5.2 Non-mandatory requirements

Requirement	Response
6.6.1 Submachine states	Not included in the proposal.
6.6.2 Protocol state machine	Proposal includes a non-normative annex on protocol state machines
6.6.3 State machine redefinition	Proposal discusses the semantics of state machine redefinition and includes its formal specification in the semantic model.
6.6.4 Asynchronous operation call	Not included in the proposal.
6.6.5 Triggers with ChangeEvents	Not included in the proposal
6.6.6 Alf for action language concrete syntax	Proposal uses Java as the action language (Alf may be used in the future).

2.3.5.3 Specification milestones

PSSM RFP was issued by March 2015. The initial response to that RFP (aka. initial submission) was provided by the PSSM team in March 2016. For that initial submission 80% of the semantic requirements were covered (i.e., supported and tested). The revised submission will be provided by the PSSM team on November 8th, 2016. This submission covers 100 % of the semantic requirements and support for these latter is shown by the 103 test cases described in the PSSM test suite.

For both submissions (i.e., initial and revised), a strong participation to the development of the semantic model, the test suite and the prototype implementation was realized in the context of the OpenCPS project. This assertion is also true for the editing of the specification.

After the revised submission, PSSM 1.0 will enter in its finalization process. The objective will be to fix all issues that may be discovered in the specification document as well as other normative artefacts: PSSM semantic model, PSSM syntax and PSSM test suite.

3 XTUML STATE MACHINES SEMANTICS AND PSSM

xtUML is a modelling language that emerged from the Shlaer-Mellor method, realized by the BridgePoint tool [7]. Up to our knowledge, it is widely used by Ericsson and Saab to design large software applications.

xtUML is designed as a UML profile that defines the constraints and extensions required to make UML executable and translatable. In xtUML the dynamic of a system can be described using state machines. The semantics defined for these state machines is in some areas different from the one defined by original UML. Nevertheless, it seems clear that both semantics for state machines rely on a common base.

Recently the Executable UML working group has worked on the definition of a Precise Semantics for UML State Machines (a.k.a, PSSM). The way the semantics is defined makes it possible to be tailored to account for xtUML state machine semantics specific requirements. The interest is that xtUML semantics will be based on the standard semantics defined for UML state machines. That means all state machine concepts that do not have a specific semantics defined in the context of xtUML will rely on the PSSM semantics. In addition, based on that semantic definition, it will be possible to precisely define what the impact of a FMI master simulation step is on a state machine defined using xtUML.

In order to define what the tailoring of the PSSM semantics should be, a preliminary analysis of semantic differences existing between UML state machines and xtUML state machines was realized within OpenCPS. Clause 3.2 provides the reader with basics on xtUML state machine abstract syntax. Clauses 3.2, 3.3, 3.4 and 3.5 discuss specific aspects of xtUML state machines for which a different semantics that the one defined in PSSM is intended.

3.1 Basics of xtUML State Machines

All xtUML state machines are flat. Hierarchy and multiple regions are not allowed. Therefore, each object instance is in exactly one state in any point in time (if we think of state transitions as instantaneous actions).

State transitions are triggered by signals arriving to the object that owns the state machine. States can have entry actions. These are executed whenever the state machine enters the state. Transitions can have effects, executed when the transition is used. A transition, including the execution of the effect, updating the actual state, execution of the entry action, and all synchronous operation calls from these, is a single run-to-completion step. Operation calls are always synchronous, while signal sending is always asynchronous.

A typical xtUML state machine is shown on Figure 16.

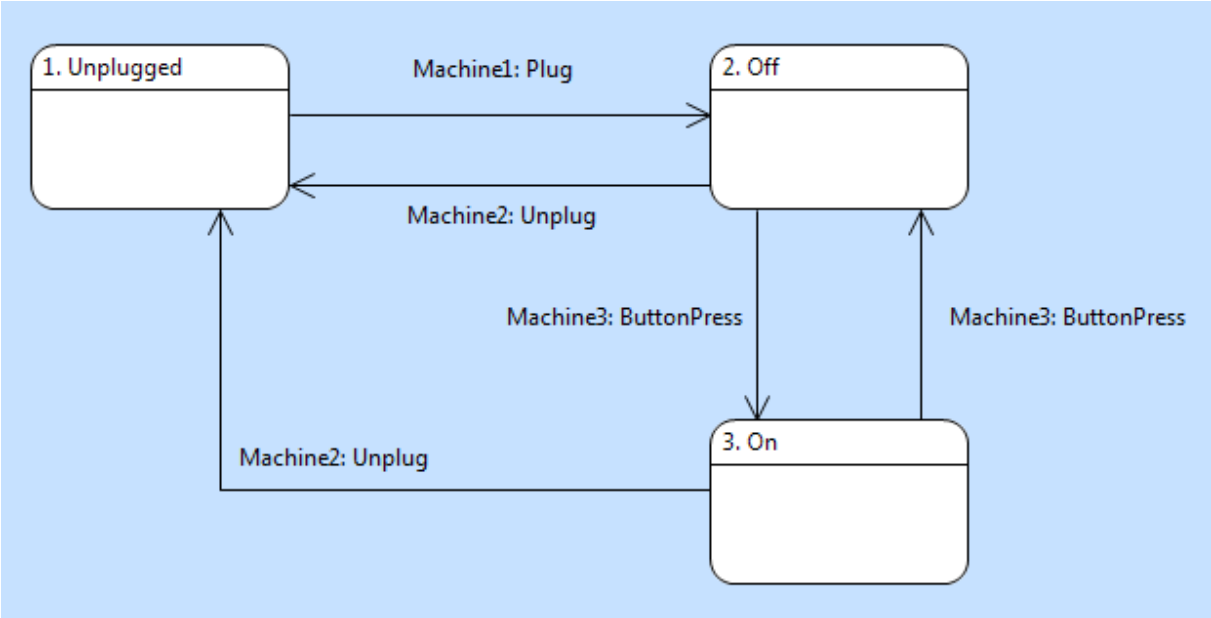


Figure 16 - An xtUML State Machine

PSSM

PSSM syntax enables definition of significantly more complex state machines than xtUML. Indeed PSSM allows, for instance, state machines to be hierarchical and to have multiple regions. Nevertheless it is incorrect to say that PSSM syntax is a superset of xtUML syntax for state machines. Indeed, it exists information that can be added in an xtUML conformant state machine model that cannot be recorded directly by a PSSM conformant model unless a profile is used. For instance in xtUML the ID of a state is an information that is used to define what the initial state of a state machine is. Another example is the specification that an event *can't happen* in particular state. UML does not provide concepts to record such information.

3.2 State Machine Initialization

States of xtUML state machines are numbered. The state with the lowest number is the initial state. When an object is created, the initial state is the active one, but its entry action is not executed.

The graphical elements that, in standard UML, denote an initial pseudostate and initial transition, in xtUML denote a "creation transition". There is a special signal sending operation in the xtUML action language that creates a new instance of the given class and executes the named creation transition in its state machine, additionally, it also executes the entry action of its target state. In this case, the target state of the creation transition will be the active state of the newly created object. There is no limit on the number of creation transitions in an xtUML state machine.

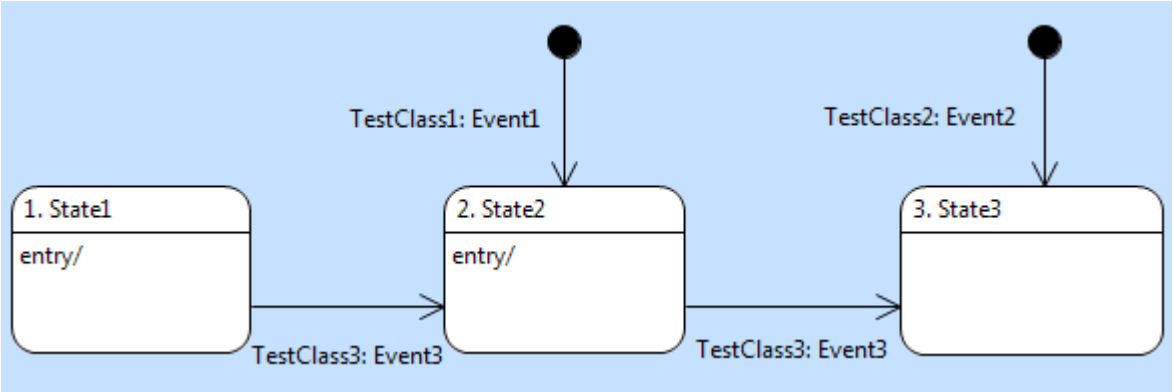


Figure 17 - State Machine with Creation Transitions

```
// instance1:
create object instance instance1 of TestClass;

// instance2:
generate TestClass1 to TestClass creator;

// instance3:
generate TestClass2 to TestClass creator;
```

Figure 18 - Object Creation Transitions

Figure 17 shows a state machine with three states and two creation actions. Figure 18 shows xtUML action code to create three object instances of the class owning the state machine in Figure 17. The first line of action code creates *instance1*, which will be in state *State1*, because that is the lowest numbered state. The next two lines of action code create *instance2* and *instance3* respectively, by using the two creation transitions. For this reason, these two instances will be in *State2* and *State3*, respectively.

PSSM

xtUML provides two ways to define what the initial state is in a particular state machine. The first approach consists in designating that state by its ID. The second approach consists in designating the state using a creation transition. Both approaches can be combined in the same state machine. At runtime, the way the container of the state machine is instantiated determines which initialization approach is used (either implicit or explicit).

1. Implicit initialization.

   - **Syntax**. UML does not provide a way to define a particular state as being the initial state based on its ID. Hence PSSM syntax does not provide such capability.

   - **Semantics**. In PSSM, if a region (which can either be owned by a state machine or a composite state) does not have an initial pseudostate, then it is ignored by the execution (see clause 14.2.3.2 of [1]). This means for instance that if a composite state has a single region and this latter has no initial pseudostate then the composite state is handled as a simple state. If we consider here the state machine shown in Figure 16 where the initial state is defined as being *unplugged*, it will not be possible to execute this latter without an extension to the PSSM semantic model. Indeed, as

it is, the main state machine region would be ignored in the execution and the execution would have completed.

As an additional difference, in PSSM if the state that is entered by an initial transition has an entry behavior then this behavior is always executed.

2. Explicit initialization.

   - **Syntax**. UML does not allow a region to have more than one initial pseudostate (see clause 14.5.8.6 of [1]). In addition, a transition that leaves a pseudostate is not allowed to declare a trigger (see clause 14.5.11.8 of [1]). These two constraints must be relaxed in order to allow definition of xtUML models based on UML. They can be defined using the approach defined in clause 2.3.1.2.

   - **Semantics**. The instantiation of a class having a state machine as its classifier behavior is not enough to make this state machine to be initialized (i.e., perform its initial RTC step). To do so, the classifier behavior of the class must explicitly be started using a `StartClassifierBehaviorAction`. The semantics to capture the starting of a classifier behavior is defined in UML. It consists in placing an `InvocationEventOcurrence` (see clause 8.4.3.2.6 of [2]) to the event pool and register an accepter for this event occurrence. When accepted the event occurrence, triggers the execution of the `Execution` (see clause 8.4.2.2.1 of [2]) corresponding to the classifier behavior. If the classifier behavior is a state machine with a single region, then the transition (maybe compound) outgoing the initial pseudostate is traversed and the target state is entered. Although the initial RTC step is initiated by the acceptance of an event occurrence, this latter does not *trigger* the initial transition. In other words, the initial transition does not have a trigger matching the dispatched event. This particular point of the semantics is different from what xtUML specifies. Indeed, in xtUML, the event occurrence that initiates the first RTC step might be used to trigger one particular initial transition among the set of initial transitions available in the state machine model (see Figure 17). The transition elected to be fired is the one with a trigger matching the creation event. In its current status, if PSSM was used to execute the model specified in Figure 17, one of the two initial transition would have been traversed during the initial RTC step and triggers placed on this transition would have been ignored.

3.3 Unexpected events

There are two different reasons for not having a transition with a given signal from a given state:

- "*Can't happen*" means that the modeled domain does not allow that particular signal to arrive in that state. If it indeed happens, that *signals an error*. This allows the developer of the state machine to specify ill-behavior of the system that requires a runtime error to be signaled.
- "*Event Ignored*" means that the given signal may arrive in that state as part of the normal operation of the system, but there is no action or state change to take, therefore the event is silently ignored.

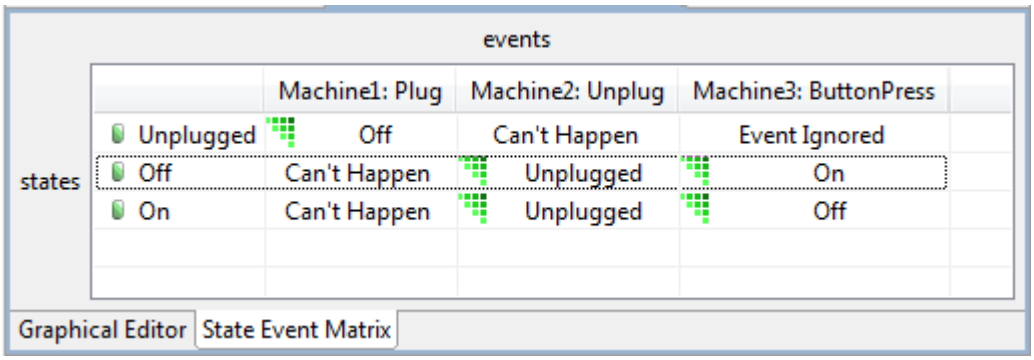


Figure 19 - State Event Matrix

Figure 19 shows the "State Event Matrix" of the state machine on Figure 16. For example, a "Plug" event in the state when the machine is "On" cannot happen (because the machine is already plugged in). On the other hand, a "ButtonPress" event can arrive even if the machine is "Unplugged", but it cannot start the machine, therefore it is ignored.

PSSM

In fUML, a classifier behavior can register event accepters for a well identified set of event types. If such accepter is registered and the dispatched event occurrence matches an accepter then the event occurrence is accepted (i.e. it initiates a RTC step). Otherwise, if no accepter match then the event occurrence is lost. Hence no RTC is initiated

1. *Can't happen*

   - **Syntax**. In UML, there is no way to specify that an event is not allowed to occur in particular state. Note that UML offers the possibility for an event to be deferred by a state but the purpose is different.

   - **Semantics**. In xtUML, when an event occurrence is lost because it is specified as *can't happen* an exception must be raised. Unfortunately, fUML, PSCS and PSSM do not provide semantics for exceptions. Support for exception raising capability is not intended to be part of PSSM but rather part of further fUML versions.

2. *Event ignored*

   - **Syntax**. In UML, there is no way to specify that an event will be explicitly ignored in a particular state machine configuration. However if no reaction is intended for an event in particular state then not any transition declaring a trigger for that event should be specified.

   - **Semantics**. In PSSM, if in the current state machine configuration the event occurrence cannot be deferred and it triggers no transition then it is lost.

3.4 Event Priorities

When dispatching an event from the event queue of an object, signals sent by that object to itself have higher priority than signals from other sources. This enables splitting complex actions into multiple states: If the entry action sends a signal to *self*, that signal will be processed before other signals from different sources, so that the entry action can determine the next state. Control structures, like sequences, branches and loops can be explicitly visualized in the state machine this way. When the object stops sending events to itself, other signals in the event queue will be dispatched.

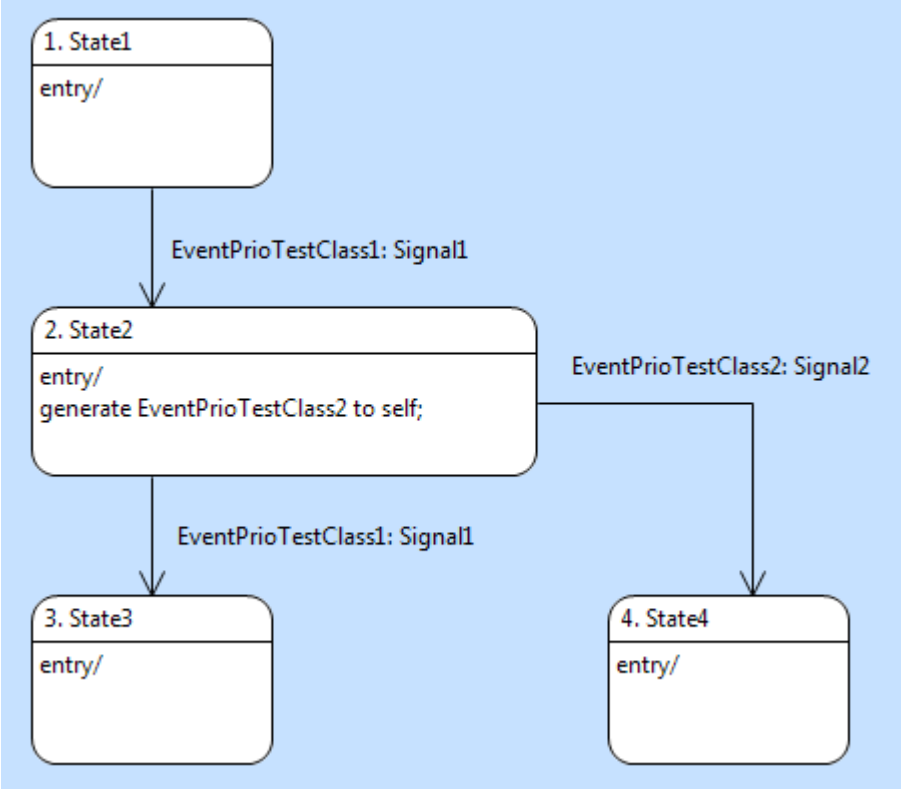


Figure 20 - State Machine with Self Event

```
create object instance inst of EventPrioTestClass;
generate EventPrioTestClass1 to inst;
generate EventPrioTestClass1 to inst;
```

Figure 21 - Action Code for Event Priority Testing

For example, in *State2* of the state machine of Figure 20 a signal is sent to *self*. The actions in Figure 21 first create an instance, which is in *State1*. Then *Signal1* is sent to the object two times. The first signal makes the object move from *State1* to *State2*, where the entry action sends a *Signal2* to the object. Even if this message arrives later, it will take priority over the second *Signal1* still in the queue. For this reason, *State4* will be active instead of *State3*.

PSSM

In PSSM, events sent by a state machine to itself do not have the priority over other events already existing in the pool. Assuming, it exits a transition between an initial pseudostate and *State1*, the execution of the state machine presented in Figure 20, through the PSSM semantic

model would be different than the one performed using xtUML semantics. Indeed, when the first *EventPrioTestClass1* is dispatched the transition between *State1* and *State2* is fired. The current configuration is now *State2* and new event occurrence of type *EventPrioTestClass2* was added to the pool. The next event to be dispatched is of type *EventPrioTestClass1* and when accepted makes the state machine to enter *State3*. It only now remains one event in the pool. This event occurrence of type *EventPrioTestClass2*. When dispatched it is lost since it cannot be used to trigger any transition.

It is important to note that in PSSM some events are generated implicitly on state completions. These events are called completion event. They are typically generated for simple state when the entry behavior (if any) and the doActivity behavior (if any) have completed their executions. Note that if no behavior is defined the state completes when entered. Completion events have the priority over other event available at the pool. When a completion event is dispatched it can trigger a completion transition outgoing the state from which it was generated. Completion events are not considered in xtUML semantics.

3.5 Polymorphic Event

The semantics of xtUML generalization relation is different from standard UML. The notation, however, is the same, see Figure 22.

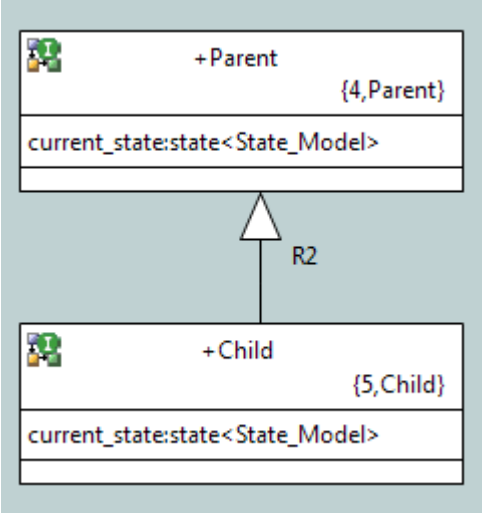


Figure 22 - Generalization Relation

The *Parent* and *Child* classes have to be instantiated separately, and explicitly linked across the generalization relation *R2*. See the first three lines of the action code in Figure 23.

```
create object instance parent of Parent;
create object instance child of Child;
relate child to parent across R2;
generate Parent1 to parent;
generate Parent2 to parent;
```

Figure 23 - Instantiation of Classes in Generalization Relation

The two instances exist and work separately, they can have separate state machines. In this respect, generalization relations are similar to simple associations, with an extra restriction: An

instance of the super type can be related to at most one subtype instance (even if there more than one subtypes).

There is no inheritance: *Child* instances do not have the attributes and operations of *Parent*. However, events are polymorphic. *Signals* sent to a *Parent* instance but not used in its state machine are automatically propagated to the related *Child* instance.

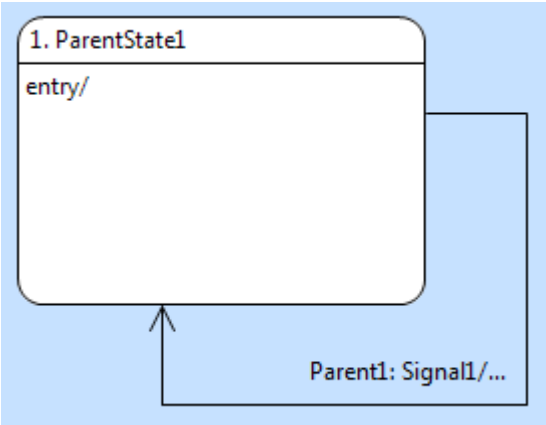


Figure 24 - State Machine of the Parent Class

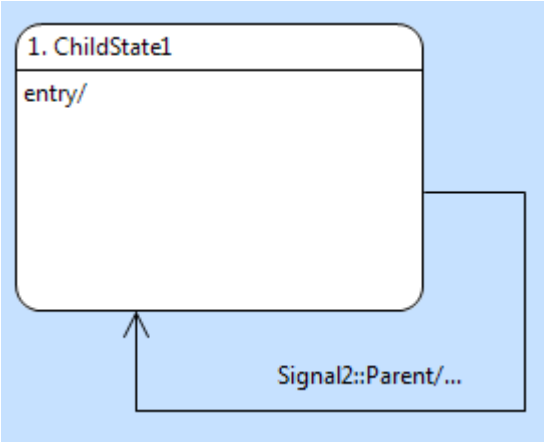


Figure 25 - State Machine of the Child Class

Figure 24 and Figure 25 shows the state machines of the *Parent* and *Child* classes respectively. Signal1 is used in Parent's state machine, while Child's state machine uses Signal2. There is mutual exclusion in using the signals: signals used in the state machine of a supertype cannot be used in the state machines of subtypes and vice versa.

The last two lines of action code in Figure 23 send Signal1 and Signal2 to the Parent instance. The first one will be processed by parent's state machine, while the second signal will be automatically propagated to and then processed by child's state machine.

PSSM

- **Syntax**.
  - UML and PSSM place no constraints on the events that can be received by the classifier behaviors. Hence if *Parent* and *Child* classes define classifier

behaviors that can both receive the same signal, the model is still syntactically and semantically correct.

- In UML the existence of a generalization relationship between two classes denotes that all features (i.e., behavioral and structural) which can be inherited will be available at the specializing class. Hence if the *Parent* class as an attribute then this latter can be used in the classifier of the *Child* class.

- **Semantics**.

  - When an instance of *Child* class is created no classifier behavior is started. It must be started explicitly using a `StartClassifierBehaviorAction`. The classifier behavior that is started is the one associated to the type of *object* pin of the action. Hence if the type is *Child* then the classifier behavior started is the one presented in Figure 25. To also have the classifier behavior of the *Parent* class running at the object activation associated to the instance of the *Child* class, it must also be started explicitly using the aforementioned approach. At runtime, the two classifier behaviors share the same event pool.

  - Consider that two event occurrences of type *Parent1* and *Parent2* are placed in the pool. When the first event occurrence is dispatched it triggers a RTC step in the classifier behavior of the *Parent* class. The other event occurrence triggers a RTC in the *Child* class classifier behavior.

  - Although the execution of the model specified in Figure 22 and receiving the stimulation sequence specified in Figure 23 is the same when the PSSM semantics or the xtUML semantics are used, the runtime structure is fundamentally different. Indeed, when executed through PSSM a single instance handles the two classifier behaviors. Conversely, when the model is executed through xtUML semantics two instances are created. Each instance handle the execution of a state machine. The semantics of the generalization relationship defines that both instance are related in the sense that classifier behaviors they execute share the same event pool. It also defines that executed classifier behaviors never compete for an event. Indeed, first the parent instance tries to accept the event and then if it is not accepted the child instance tries to accept it. In fUML this works differently since all classifier behaviors compete to accept an event occurrence.

3.6 Summary

Within this first evaluation of the differences between the original UML (as captured in PSSM) and the xtUML state machines we have identified 4 main categories of differences: state machine initialization, handling of unexpected events, event priorities and polymorphic events that are described in sections 3.2 3.3 3.4 and 3.5, respectively.

4 CONCLUSIONS

This document reports progress made on task T2.2 "*Interoperability of the standards Modelica-UML-FMI*" over the last twelve months. Two contributions are presented.

1. First contribution, is the participation to the definition of a Precise Semantics for UML State Machines. This definition has been proposed at the OMG by the PSSM team as a response to the PSSM RFP issued in March 2015. This document provides an overview of the specification architecture, highlights key points of the defined semantics and explains how this semantics was tested to ensure its was consistent with the one described in UML 2.5 [1]. This first contribution was a required step to ensure the possibility to use UML models with behaviours defined as state machines in a simulation process.

2. Second contribution is the analysis of the semantic differences existing between the semantics captured by PSSM and semantics of xtUML state machines. With the identification of these differences it is now possible to estimate which extensions would be required to the PSSM semantic model to be able to capture the semantics specific requirements of xtUML state machines. If such extensions to PSSM are defined, then xtUML models built using UML and a profile will be also usable in a simulation process.

Now that we have a precise semantics for UML state machines, the next step of the test is to clarify what an FMI simulation step implies in terms of execution in a state machine executed as the classifier behaviour of an FMU. This work will be reported in the next deliverable.

REFERENCES

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

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

[3] OMG, "Precise Semantics of UML Composite Structures (PSCS)," 2015.

[4] M. Association, "Modelica - A Unified Object-Oriented Language for Systems Modeling," 2014.

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

[6] OMG, "Action Language for Foundational UML (Alf)," 2013.

[7] OMG, "Precise Semantics of UML State Machines Request For Proposal (PSSM RFP)," 2015.

[8] OMG, "Object Constraint Language (OCL)," 2014.

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