Date: November 2016

Precise Semantics of UML State Machines (PSSM)

Revised Submission

In response to the Precise Semantics of UML State Machines RFP (ad/2015-03-02)

OMG Document Number: ad/2016-11-01

Machine Readable File(s):

PSSM Syntax Metamodel XMI, ad/2016-11-02

PSSM Semantics Model XMI, ad/2016-11-03

PSSM Test Suite Model XMI, ad/2016-11-04

fUML Revised Syntax Metamodel XMI, ad/2016-11-05

fUML Revised Semantics Model XMI, ad/2016-11-06

PSCS Revised Syntax Metamodel XMI, ad/2016-11-07

PSCS Revised Semantics Model XMI, ad/2016-11-08

Submitted by:
BAE Systems
Model Driven Solutions
No Magic, Inc.

Supported by:
Airbus Group
Commissariat á l'Energie Atomique et aux Energies Alternatives (CEA)
LieberLieber Software
Simula Research Laboratory

Table of Contents

0 Submission Introduction

0.1 Overview

This is OMG document ad/2016-92-xx, entitled *Precise Semantics of UML State Machines*, which may be abbreviated "PSSM". It is submitted in response to the RFP for *Precise Semantics of UML State Machines (PSSM)* (ad/2015-03-02). Similarly to previous Executable UML specifications on the *Semantics of a Foundational Subset for Executable UML Models (fUML)* and the *Precise Semantics of UML Composite Structures (PSCS),* the proposed PSSM specification defines the subset of the UML abstract syntax relevant to UML state machines, as an extension of the fUML abstract syntax subset (see Clause 7), and an operational semantics for that subset, as an extension to the fUML/PSCS execution model (see Clause 8). In addition, as required in the RFP, the proposed specification includes a suite of tests that may be used to validate the conformance of an execution tool to the specification (see Clause 9).

0.2 Submitters

The following OMG member organizations (with the given contact individuals) are jointly submitting this proposed specification.

- BAE Systems (Gregory Eakman, gregory.eakman@baesystems.com)
- Model Driven Solutions (Ed Seidewitz, ed-s@modeldriven.com)
- No Magic, Inc. (Nerijus Jankevicius, nerijus@nomagic.com)

The following OMG member organizations are supporters of and contributors to this proposed specification, but not formal submitters. All have licensed any contributed material as necessary to one of the formally submitting organizations.

- Airbus
- Simula Research Laboratory
- LieberLieber
- CEA

0.3 Mandatory Requirements

The following table describes how this proposed specification responds to each of the mandatory requirements of the PSSM RFP.

Requirement	Response
6.5.1a Proposals shall provide precise semantics for UML behavior state machines, including the following underlying metaclasses: i. FinalState ii. Pseudostate (all kinds) iii. Region (except for redefinition) iv. State (except for redefinition and submachine states) v. StateMachine (except state machine extension) vi. Transition (except for redefinition), including completion transitions (with no triggers) and transitions with triggers for the following kinds of events: 1. Call Event 2. Signal Event vii. Vertex	This proposal provides precise semantics for all required elements, as captured in the execution model described in Clause 8.
6.5.1b Proposals shall define how data associated with event occurrences shall be accessed by transition guards and passed to transition effect behaviors and state behaviors during the process of event dispatching and transition triggering.	The syntactic approach for the proposed approach for event data passing is described in 7.6.2.1. The semantics for it are defined in 8.5.10.
6.5.1c The precise semantics shall cover at least the cases of the standalone execution of state machines (i.e., with no other behaviored classifier as context) and state machines used as classifier behaviors of active classes.	The abstract syntax for state machines is restricted in this proposal to just the two cases of standalone state machines and state machines used as classifier behaviors (see the pssm_state_machine_context constraint in 7.6.2.2). Both of these cases are covered by the precise semantics described in 8.5.
6.5.1d The precise semantics for state machines shall include the meaning of specifying a port on a trigger in a state machine. The proposed semantics for this (and any other potential touch points with UML composite structure as identified by submitters) shall be consistent with the semantics of composite structures as defined in the Precise Semantics of UML Composite Structures (PSCS) specification. This consistency shall be such that there would be no conflict in a tool conforming to both the proposed state machine semantics and PSCS. However, proposals shall not require that a tool necessarily formally conform to the entire PSCS specification in order to conform to the precise semantics for state machines.	The semantics of a trigger with "port" references is covered as part of the execution model for state machines in 8.5. The proposed PSSM execution model is defined as an extension to the PSCS execution model, in order to ensure compatibility with PSCS. However, the proposal also provides a "PSSM-only" conformance level, which does not require a tool to conform to PSCS, as well as a "Joint PSSM and PSCS" conformance level, which requires conformance to both PSSM and PSCS (see the discussion in Clause 2).

Requirement	Response
6.5.1e The semantic description shall establish explicit relationships with fUML, for example by specifying a precise formal model transformation from the metaclasses listed above to metaclasses which are part of the fUML subset and/or by extending the fUML execution model, for example with appropriate visitor classes. Whatever the way the execution semantics are actually specified, proposals shall be readable as if they are additions to fUML semantics, rather than separate specifications.	This proposal defines the precise semantics for state machines using an execution model that extends the PSCS execution model, which is itself an extension of the fUML execution model.
6.5.1f Proposals shall extend the base semantics of fUML with specific axioms for UML state machines only if necessary. These new axioms shall have explicit relationships with existing axioms of fUML base semantics. These axioms shall be expressed in Common Logic Interchange Format (as was done for fUML). Submitters shall demonstrate, through manual or automated means, that the new axioms are consistent with fUML axioms.	This proposal does not extend the fUML base semantics.
6.5.2a Proposals shall precisely identify any allowed semantic variabilities. These semantic variabilities shall be in the scope of semantic variabilities allowed by UML state machines (potentially including only a subset of allowed UML semantic variabilities, as was the case for fUML).	This proposal does not define any semantic variabilities beyond those already allowed in fUML (see also the discussion 2.3).
6.5.2b Proposals shall define rules for defining semantic variants, where a semantic variant is an internally consistent set of values for the different semantic variabilities allowed from requirement 6.5.2.a.	This proposal does not define any semantic variants beyond those already allowed in fUML.
6.5.3a Proposals shall conform to the current version of the UML 2 metamodel and notation.	The proposed PSSM syntactic subset is a subset of the UML 2.5 abstract syntax metamodel (see Clause 7).
6.5.3b Proposals shall use the current version of the fUML specification.	The proposed PSSM semantics is based on the fUML 1.2.1 execution model. However, fUML 1.2.1 is based on UML 2.4.1, which has a different syntactic package structure than UML 2.5. In order to be consistent with the UML 2.5 package structure used in the PSSM syntactic subset, this proposal proposes that the fUML abstract syntax and execution models be reorganized into a structure consistent with UML 2.5 (see 6.2).

Requirement	Response
6.5.3c Proposals shall use the current version of the PSCS specification.	The proposed PSSM semantics are based on the PSCS 1.0 execution model. However, PSCS 1.0 is based on UML 2.4.1, which has a different syntactic package structure than UML 2.5. In order to be consistent with the UML 2.5 package structure used in the PSSM syntactic subset, this proposal proposes that the PSCS abstract syntax and execution models be reorganized into a structure consistent with UML 2.5, (see 6.2).
6.5.3d For any extension to the fUML base semantics using CLIF, proposals shall conform to the current version of the ISO Common Logic standard.	This proposal does not extend the fUML base semantics.
6.5.4a Proposals shall provide a suite of test cases that can demonstrate conformance to this specification.	The proposed test suite is described in Clause 9.
6.5.4b Proposals shall demonstrate the coverage by the test suite of all proposed state machine semantic functionality and the traceability of each test case to specific required functionality.	The test suite traces to a set of 108 semantic requirements identified from the semantic specification for state machines in Clause 14 of the UML specification [UML]. The proof-of-concept implementation developed by the submission team (see 0.5, issue 5) passes all tests in the test suite, verifying that the execution model defined in Clause 8 satisfies all 108 requirements.

0.4 Non-Mandatory Features

The following table describes whether and how this proposed specification provides the non-mandatory features listed in the PSSM RFP.

Feature	Response
6.6.1 Proposals may provide precise semantics of submachine states, as represented by the A_submachineState_submachine meta-association and including the ConnectionPointReference metaclass.	This proposal does not cover submachine states.
6.6.2 Proposals may provide precise semantics of UML protocol state machines, including the following underlying metaclasses: a. ProtocolConformance b. ProtocolStateMachine c. ProtocolTransition	The operational semantics of protocol state machines require the raising of exceptions in certain cases. However, fUML does not currently define the semantics of exceptions. Consequently, the submission team considered it inappropriate to define exception handling for all of fUML, based on just the PSSM-specific requirement to handle protocol state machine. Therefore, the proposal only includes a non-normative discussion of protocol state machine behavior, without a formal execution model (see Annex A).

Feature	Response
6.6.3 Proposals may provide precise semantics for state machine redefinition, as represented by the following meta-associations: a. A_extendedRegion_region b. A_extendedStateMachine_stateMachine c. A_redefinedState_state d. A_redefinedTransition_transition e. A_redefinitionContext_region f. A_redefinitionContext_state g. A_redefinitionContext_transition	This proposal includes semantics for state machine redefinition (see RegionActivation in 8.5.3).
6.6.4 Proposals may provide precise semantics for asynchronous operation calls (which are not currently allowed in fUML). If provided, such semantics should include the handling of asynchronous calls both by call event triggers in state machines and by operation methods. Proposals may additionally provide semantics for accepting call events in activities, as covered by the AcceptCallAction and ReplyAction metaclasses.	This proposal does not provide semantics for asynchronous operation calls.
6.6.5 Proposals may provide precise semantics for triggers with ChangeEvents.	This proposal does not provide semantics for triggers with change events.
6.6.6 Proposals may use the Action language for Foundational UML (Alf) as a concrete syntax for specifying the execution semantics of state machines.	This proposal currently uses Java as the concrete syntax for specifying detailed operation behaviors (as was also done in fUML and PSCS).

0.5 Issues To Be Discussed

1. Proposals shall discuss how state machines may be used to specify the behavior of passive classes.

The use of state machines for describing the behavior of instances of passive classes can be found in several UML tools, notably in Rhapsody and RSA RTE (both from IBM). In fact, the original 1.x series of UML specifications explicitly discusses the case of passive state machines (e.g., section 2.12.4.7 in the UML 1.3 specification). In particular, it mentions a possible means for ensuring preservation of run-to-completion semantics for passive classes whose behavior is specified via state machines.

However, the semantics of state machines used with passive classes is not entirely consistent with the normative semantics for behavior state machines for active classes, as specified in this proposal. Nevertheless, the submitters feel that the use of state machines with passive classes will still be of interest to those applying the PSSM specification, so we have included a discussion of this usage as informative Annex B.

2. **Proposals shall address issues with the UML abstract syntax involved in the specification of the accessing and passing of data from event occurrences, as required in 6.5.1b.**

This proposal addresses the issue of passing event data to and from effect, entry, exit and doActivity Behaviors, without changing the UML abstract syntax, by allowing Parameters on such Behaviors by which such data can be passed. This is covered syntactically in 7.6.2, particularly by the constraint Error: Reference source not found, and is captured semantically in the state machine execution model in 8.5.10.

The passing of data to Transition guard Behaviors is proposed to be handled in a similar way. Transition guards are actually Constraints (see [UML], subclause 14.3.2, and Figure 7.7 in this proposal), and the executable Behavior for such a guard must be given as the behavior of an OpaqueExpression acting as the specification of the constraint. This proposal is based on UML 2.5.1, in which the behavior of an OpaqueExpression may have in Parameters, in addition to a single required out Parameter (see 6.2). The in Parameters on such a Behavior may then be used to pass event data to be used in evaluating the guard expression.

3. **Proposals shall discuss the relationship of the proposed precise semantics for UML state machines to the causality model defined for the UML Profile for MARTE.**

The causality model defined in the UML Profile for MARTE[1] integrates the various mechanisms by which a behavior can be triggered upon the reception of an event occurrence. This model takes its semantics from UML 2.1 and explicitly declares conformance to Clause 13 of the draft UML 2.1 specification in course of preparation at the time.[2]

The two key aspects of the MARTE causality model are: the fact that a behavior occurs due to the existence of an event occurrence to which a time instant (or partial order) may be associated, and the need to indicate the active object (thread, concurrent unit, etc.) that will process the event occurrence in order to trigger the execution of a behavior. The causality model in MARTE treats behaviors in general, it makes no explicit distinction for state machines, though it states that the dispatching may be precisely described in the semantics of the high level or concurrency mechanisms used. The basic elements in this model are events, triggers and behaviors, plus the request to describe the communication of events among active elements.

The level of detail at which the relevant semantic aspects are described in MARTE, allows this proposal to fix the semantics of state machines freely as far as it is possible to distinguish the start and termination event occurrences associated to each behavior execution. Additionally, when a communication is needed among active elements, it must be possible to indicate the invocation occurrence and the receive occurrence in the concrete instances involved.

In general terms it can be seen that the relationship of the precise semantics for UML state machines described in this proposal to the causality model in MARTE is conditioned by the evolution of UML since its version 2.1. However, there still seems to be general consistency with the UML 2.5 Common Behavior semantics (see [UML], Clause 13).

4. **Proposals shall discuss the relationship of the proposed precise semantics for UML state machines to the specification of a state machine ontology and, particularly, to the integration approach of OntoIOP.**

This proposal uses an operational approach for defining the semantics of state machines, extending the similar approach used for fUML and PSCS (see [fUML] and [PSCS]). As was originally done for fUML, the PSSM operational semantics are given by an executable model written in the *base UML* (bUML) subset of fUML. The fUML specification provides a *base semantics* for bUML (see [fUML], Clause 10), using first-order predicate logic to specify axioms on the allowed execution traces for an executable bUML model. In this sense, the PSSM semantic specification is formally reducible to a set of assertions about the

1 *UML Profile for MARTE: Modeling and Analysis of Real-time Embedded Systems,* OMG document formal/11-06-02, subclauses 7.2.2 to 7.2.5.

2 *OMG Unified Modeling Language, Version 2.1,* OMG document ptc/2006-04-02.

execution traces generated by the PSSM execution model when interpreting a state machine model, and this could be used as a basis for integration into the OntoIOP Distributed Ontology, Modeling, and Specification Language (DOL)[3]

Indeed, the DOL specification already includes an informative annex showing how the basic class-modeling capabilities of bUML can be integrated into the DOL framework. However, this does not currently cover the behavioral-modeling aspects of bUML, which would have to be included to be able to reason about a behavioral model like the PSSM execution model. But, even if the DOL work was expanded to include bUML behavioral-modeling constructs, the indirection of reasoning about a state machine through a formal model of an operational model of state-machine execution could prove unwieldy. Therefore, it might be useful to deduce a formal logical semantics directly for at least some part of the PSSM abstract syntax subset, with equivalent semantics, in its scope, to the standard PSSM operational semantics.

However, the PSSM submission team has not had the time or resources to properly pursue this further, and it is thus not addressed at all in the main body of this proposal.

5. **Proposals shall discuss the relationship of the proposed precise semantics for UML state machines to the semantics defined for state machines in the W3C State Chart XML (SCXML) specification.**

It is fairly clear that SCXML was inspired in great part by UML. There is a significant conceptual, semantic, and even terminological overlap. However, there are also some important differences.

- *No structural context.* The general environment of SCXML state machines is completely different from that of UML state machines. In SCXML, a state machine is an independent self-contained entity that can interact with other, or *external*, entities located in a Web-based environment (e.g., external entities are accessed via URIs). External entities can be other SCXML state machines or any other kind of Web-based application. This interaction can be either synchronous or asynchronous. In other words, unlike UML, in SCXML there is no explicit structural context defined in which a state machine is defined; i.e., the (single) state machine always *is* the top-level concept.
- *No support for certain kinds of pseudostates.* SCXML only supports three kinds of pseudostates: initial, deep history, and shallow history. This means that it does not support exit and entry points, fork and join points, or terminate pseudostates. Also, there is no support for submachine states or state machine redefinition.
- *No support for "doActivity" behaviors.* The UML concept of "doActivity" behaviors associated with states is not directly supported in SCXML. The SCXML "invoke" is a somewhat similar capability, however, in the case of SCXML, the invoked service is "external" rather being limited to the context of the state machine or its owning classifier. Further, the semantics of when the invocation actually occurs and how the invoked service may communicate with its invoking state machine are not the same as for a UML "doActivity" behavior.
- *No support for protocol state machines.* (But protocol state machines are only covered informatively in this PSSM proposal anyway.)
- *No support for submachines.* (But submachines are also not included in this PSSM proposal).
- *States can own local extended data variables.* An SCXML document defines a single state machine as a set of states and a set of associated data (i.e., its "extended" state). States of any kind can optionally own local sets of data, something that has no equivalent in UML.
- *Different data and event models.* The data and event models are specific to SCXML and not equivalent to those in UML.
- *More refined model of completion events.* Like UML, SCXML supports completion event, but, in contrast to UML, these events are named and used to explicitly trigger completion transitions. However, their effect is the same, although it appears that SCXML gives the modeler more control over the triggering because individual completion events can be

3 *Distributed Ontology, Modeling, and Specification Language (DOL), v1.0 Beta 1,* OMG document ptc/2016-02-37.

differentiated. For example, it is possible to define a trigger that refers explicitly to the completion of a particular state. This cannot be done in UML since completions event triggers are implicit.

- *SCXML-specific action language.* The action language of SCXML is specific to it, although most action language (executable content) elements can be mapped to standard programming language equivalents.

On the other hand, it seems that the semantics of those SCXML concepts that have UML equivalents are compatible with UML semantics. Because of this semantic consistency, it is possible to define a mapping from a subset of the UML state machine abstract syntax into SCXML, such that a mapped state machine model will execute according to the SCXML specification with semantics that conform to those given in this proposal for UML state machines.

6. Proposals shall describe a proof of concept implementation that can successfully execute tests from the conformance test suite, without violating any tests from the PSCS conformance test suite.

CEA has developed a prototype proof-of-concept implementation of the execution model defined in Clause 8 of this specification.[4] It is integrated into the Papyrus model execution framework Moka[5] as a specific execution engine. Papyrus and Moka tooling are based on the Eclipse implementation of the UML 2.5 metamodel. Therefore, models that can be interpreted by the prototype must conform to this metamodel implementation.

The execution model implemented by the prototype is built on top of the one defined to capture the PSCS semantics. The prototype is able to execute any model conforming to the subset that is covered both by fUML and PSCS. In addition, it is also able to execute models including syntax elements that are covered by the subset of behavior state machines identified in this specification (see 7.6.2).

The PSSM execution model does not conflict with PSCS execution model. Both execution models can be used jointly. To demonstrate the absence of conflicts, the PSCS test suite was executed using the PSSM execution model. No regressions were detected. Execution traces generated by PSCS and PSSM during test suite execution are the same.

As required, this proposal also includes a conformance test suite, which is intended to validate conformance to the execution model (see Clause 9). The proof of concept implementation passes all tests in the test suite.

4 The code for this prototype is publicly available in the Eclipse Git repository, at http://git.eclipse.org/c/papyrus/org.eclipse.papyrus.git/tree/extraplugins/moka/org.eclipse.papyrus.moka.fuml.statemachines?h=bugs/465888-SMExecPrototype.

5 See https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution.

1 Scope

The *Precise Semantics of UML State Machines (PSSM)* specification is an extension of the *Semantics of a Foundational Subset for Executable UML Models* standard (known as "Foundational UML" or "fUML") [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 the *OMG Unified Modeling Language* specification [UML] (Clause 14, for UML 2.5 and later). Semantically, this specification extends the fUML execution model in order to specify the operational execution semantics of the state machine abstract syntax subset.

The semantic model defined in this specification is actually an extension of the model from the *Precise Semantics of UML Composite Structures (PSCS)* standard [PSCS], which is itself an extension of fUML. This is done in order to ensure that the semantics given in this specification are compatible with the extensions defined in PSCS and to allow for the definition of the semantics of triggers reference specific ports of an enclosing composite structure. However, this latter feature is the only point for which the semantics of state machines presented here depends in any way on the PSCS semantic extensions, and it is possible for an execution tool to conform to this specification without also conforming to the PSCS specification (see Clause 2).

Figure 1.1 shows schematically the relationship of PSSM to the syntactic and semantic models from the fUML and PSCS specifications.

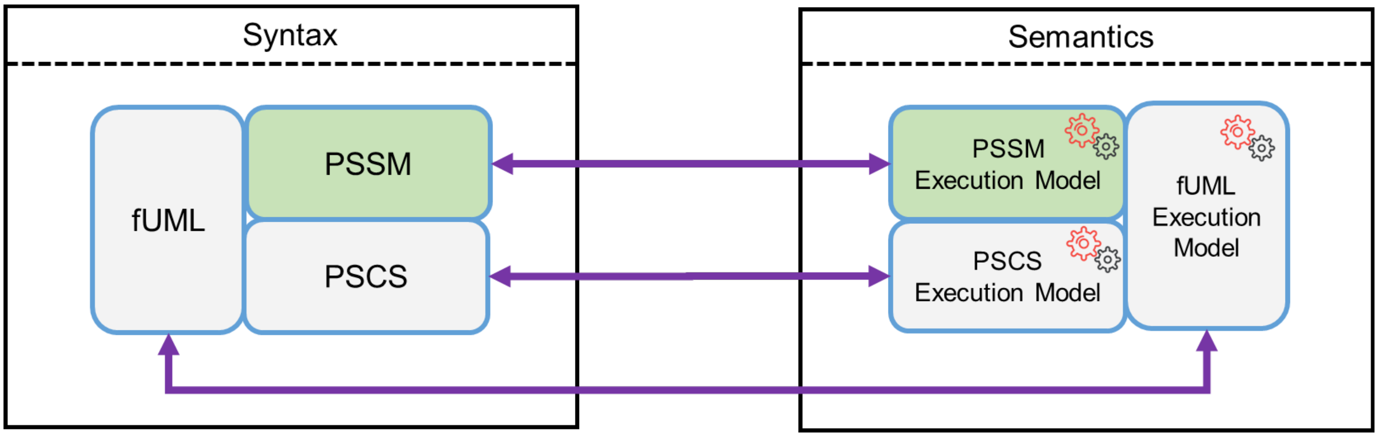


Figure 1.1 - Scope of this specification

2 Conformance

2.1 General

The PSSM specification is based on fUML. Hence, except where explicitly noted in this clause, the definitions, interpretations (meaning), and types of conformance and related terms in this specification fully match their corresponding definitions, interpretations, and types in fUML (see [fUML], Clause 2). Thus, as in fUML, conformance to this specification has two aspects:

1. *Syntactic Conformance* – A conforming model must be restricted to the abstract syntax subset defined in Clause 7 of this specification.

2. *Semantic Conformance* – A conforming execution tool must provide execution semantics for a conforming model consistent with the semantics specified in Clause 8 of this document.

2.2 Conformance Levels

The semantic model in Clause 8 are specified as an extension to the semantic model given in [PSCS]. However, the only point at which the semantics given in this specification actually depend on the PSCS semantics is for triggers with "port" references (see [UML], 13.3). There are two levels of conformance defined for this specification, depending on whether an execution tool conforms only to PSSM or conforms to PSSM *and* PSCS, including the semantics for triggers with "port" references. Both of these have syntactic and semantic aspects, as specified in the following subclauses.

2.2.1 PSSM-only Conformance

1. *Syntactic Conformance* – A conforming model must be restricted to the abstract syntax subset defined in Clause 7 of this specification, including the satisfaction of all additional constraints.

   Note. The abstract syntax subset defined in Clause 7 is a superset of the subset defined in [fUML]. Thus, every syntactically conforming fUML model is also a syntactically conforming PSSM model. The PSSM subset does *not* itself include Ports, so a model syntactically conforming to only the PSSM subset cannot have "port" references on any triggers.

2. *Semantic Conformance* – A conforming execution tool must provide execution semantics for a conforming model consistent with the semantics specified in Clause 8 of this specification. Demonstrating semantic conformance to fUML (as defined in Clause 2 of [fUML] and passing all the tests of the test suite in Clause 9 of this specification, *except* for those related to triggers with "port" references, is sufficient to demonstrate semantic conformance at this level.

2.2.2 Joint PSSM and PSCS Conformance

1. *Syntactic Conformance* – A conforming model must be restricted to the abstract syntax subset defined by the union of the subset defined Clause 7 of this specification *and* the subset defined in Clause 7 of [PSCS]. The model shall satisfy all constraints as specified in this specification and in [PSCS].

2. *Semantic Conformance* – A conforming execution tool must provide execution semantics for a conforming model consistent with the semantics specified in Clause 8 of this specification *and* the semantics specified in Clause 8 of [PSCS]. Demonstrating semantic conformance to fUML (as defined in Clause 2 of [fUML]) and passing all the tests of the test suite in Clause 9 of this specification *and* all the tests in Clause 9 of [PSCS] is sufficient to demonstrate semantic conformance at this level.

2.3 Genericity of the Execution Model

To support a variety of different execution paradigms and environments, the specification of the execution model incorporates a degree of genericity. This is achieved in two ways: (1) by leaving some key semantic elements unconstrained, and (2) by defining explicit semantic variation points. A particular execution tool can then realize specific semantics by suitably constraining the unconstrained semantic aspects and providing specifications for any desired variation at semantic variation points.

The semantic areas that are not explicitly constrained by the execution model in this specification are the same as the ones defined in subclause 2.3 of [fUML]. Different execution tools may semantically vary in these areas in executing the same model, while still being conformant to the semantics specified by the execution model in this specification. Additional semantic specifications or constraints may be provided for a specific execution tool in these areas, so long as it remains, overall, conformant to the execution model. For instance, a particular tool may be limited to a single centralized time source such that all time measurements can be fully ordered.

In contrast to the above areas, subclause 2.3 of [fUML] defines a set of explicit semantic variation points. The execution model as given in this specification by default fully specifies the semantics of these items. However, it is allowable for a conforming execution tool to define alternate semantics for them, so long as this alternative is fully specified as part of the conformance statement for the tool. This specification does not define any further semantic variation points in addition to those defined in fUML. Note, however, that the default event dispatching strategy defined for fUML is replaced by the default strategy given in subclause 8.4.1.2.1 of [PSCS], but this is only relevant for Joint PSSM and PSCS Conformance (see 2.2.2).

If a conforming execution tool wishes to implement a semantic variation in one of the above areas, then a specification must be provided for this variation via a specialization of the appropriate execution model class as identified above. This specification must be provided as a fUML model in the "base UML" subset interpretable by the base semantics of Clause 10 of [fUML]. Further, it must be defined in what cases the variation is used and, if different variants may be used in different cases, when each variant applies, and/or how what variant to use, is to be specified in a conforming model accepted by the execution tool.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For versioned references, subsequent amendments to, or revisions of, any of these publications do not apply.

[fUML]	*Semantics of a Foundational Subset for Executable UML Models (fUML),* Version 1.2.1, http://www.omg.org/spec/FUML
[PSCS]	*Precise Semantics of UML Composite Structures (PSCS),* Version 1.0, http://www.omg.org/spec/PSCS
[UML]	*OMG Unified Modeling Language (OMG UML),* Version 2.5.1, http://www.omg.org/spec/UML
[UTP]	*UML Testing Profile (UTP),* Version 2, Revised Submission, OMG document ad/2016-05-10

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Base Semantics

A definition of the execution semantics of those UML constructs used in the execution model, using some formalism other than the execution model itself. Since the execution model is a UML model, the base semantics are necessary in order to provide non-circular grounding for the execution semantics defined by the execution model. The base semantics provide the "meaning" for the execution of just those UML constructs used in the execution model. The execution model then defines the "meaning" of executing any UML model based on the full foundational subset. Any execution tool that executes the execution model should reproduce the execution behavior specified for it by the base semantics. (The base semantics for this specification are as specified in [fUML].)

Behavioral Semantics

The denotational mapping of appropriate language elements to a specification of a dynamic behavior resulting in changes over time to instances in the semantic domain about which the language is making statements.

Execution Model

A model that provides a complete, abstract specification to which a valid execution tool must conform. Such a model defines the required behavior of a valid execution tool in carrying out its function of executing a UML model and therefore provides a definition of the semantics of such execution.

Execution Semantics

The behavioral semantics of UML constructs that specify operational action over time, describing or constraining allowable behavior in the domain being modeled.

Execution Tool

Any tool that is capable of executing any valid UML model that is based on the abstract syntax subset defined in this specification and expressed as an instantiation of the UML 2 abstract syntax metamodel. This may involve direct interpretation of UML models and/or generation of equivalent computer programs from the models through some kind of automated transformations. Such a tool may also itself be concurrent and distributed.

Static Semantics

Possible context sensitive constraints that statements of a language must satisfy, beyond their base syntax, in order to be well-formed.

Structural Semantics

The denotational mapping of appropriate language elements to instances in the semantic domain about which the language makes statements.

Syntax

The rules for how to construct well-formed statements in a language or, equivalently, for validating that a proposed statement is actually well-formed.

5 Symbols

There are no symbols or abbreviated terms necessary for the understanding of this specification.

6 Additional Information

6.1 Relationship to UML

The PSSM specification is based on version 2.5.1 of UML, which addresses certain issues with UML 2.5 whose resolution was critical for PSSM. The relevant issues, are:

- *UMLR-92 UML/OCL spec mismatch* – The OCL 2.4.1 specification [OCL] states that, when OCL is used to specify a UML Constraint used as a class invariant, the OCL context classifier (i.e., the type of the OCL keyword `self`) is the `constrainedElement` of the Constraint. However, the UML 2.5 specification stated (as had previous versions before it) that the `context` Namespace that owns a Constraint acts as `self` if the Constraint is specified using OCL (presumably in the case that the Namespace is a Classifier), which was inconsistent with the OCL specification. The offending statement is removed in UML 2.5.1, making it consistent with the OCL specification. This consistency is important for the approach used to add constraints to the PSSM syntax subset (see 7.1).
- *UMLR-685 StateMachine Vertex needs to be made a kind of RedefinableElement* – UML allows a StateMachine to be extended through redefinition, providing also for the redefinition of Regions, States and Transitions within the extension StateMachine. However, in UML 2.5, if new States where added to an extension StateMachine, then it was possible to define Transitions that crossed from the extended StateMachine to the extension StateMachine. UML 2.5.1 has an additional constraint that disallows such Transitions, requiring the `source` and `target` of a Transition be contained in the same StateMachine as the Transition. This new constraint means that, when a Transition is added to an extension StateMachine with the intention of having an existing Vertex in the extended StateMachine as its `source`, then that `source` Vertex actually needs to be redefined in the extension StateMachine. This, in turn, implies that any kind of Vertex, not just States, must be redefinable, which is possible in UML 2.5.1. The constraint to disallow Transitions from crossing from one StateMachine to another is important in PSSM for the specification of StateMachine execution semantics (see 8.5.3). The PSSM semantics also take into account that Pseudostates are redefinable, in addition to States. (There is a third kind of Vertex, ConnectionPointReference, that also becomes redefinable in UML 2.5.1, but the PSSM syntax subset does not include ConnectionPointReference.)
- *UMLR-696 The behavior of an OpaqueExpression should be allowed to have input parameters* – In UML 2.5, if an OpaqueExpression specified a `behavior`, then that `behavior` was required to have a return Parameter and no other Parameters. In UML 2.5.1, such a `behavior` is also allow to have in Parameters. In PSSM, this is used to pass data from an event occurrence into a Behavior on an OpaqueExpression used to specify the `guard` Constraint on a Transition (see 7.3 and 8.5.10).

6.2 Changes to Adopted OMG Specifications

The PSSM syntax is a subset of the UML 2.5.1 abstract syntax metamodel, and the required functionality formalized in PSSM is taken from that specified in UML 2.5.1. However, PSSM is also semantically based on fUML and PSCS. But the current versions of these standards, fUML 1.2.1 [fUML] and PSCS 1.0 [PSCS] are based on UML 2.4. In order to avoid inconsistency, particularly given the sweeping reorganization of the UML abstract syntax metamodel adopted in UML 2.5, the fUML and PSCS syntax and semantics models have been migrated to UML 2.5.1 for use with PSSM, but with no change to their functionality. In addition, the fUML and PSCS models have been updated to use an approach for identifying and constraining their syntax subsets that is consistent with that used in PSSM (see 7.1)

6.3 Acknowledgments

The following people directly contributed to the development of this specification.

- Yves Bernard, Airbus
- Nerijus Jankevicius, No Magic
- Julio Medina, Universidad de Cantabria
- Ed Seidewitz, Model Driven Solutions
- Bran Selic, Simula Research Laboratory
- Daniel Siegl, LieberLieber
- Jeffery Smith, Sierra Nevada Corporation (formerly BEA Systems)
- Jérémie Tatibouet, CEA

7 Abstract Syntax

7.1 Overview

This clause defines the subset of the UML abstract syntax [UML] for which precise semantics are provided in this specification. Models that syntactically conform to this subset (see 2.1) are simply UML models constructed from the limited set of UML abstract syntax metaclasses included in the PSSM subset defined here. The definition of the subset thus consists of identifying exactly which metaclasses are included.

The subset definition is captured in the package PSSM_Syntax::Syntax, which imports into its namespace exactly those UML metaclasses included in the PSSM subset (see Figure 7.1). A UML model that syntactically conforms to this subset shall have an abstract syntax representation that consists solely of instances of metaclasses that are (imported) members of the PSSM_Syntax::Syntax package. For simplicity, meta-associations from the UML abstract syntax metamodel are *not* explicitly imported into the PSSM_Syntax::Syntax package, but it is, nevertheless, permissible for the model elements of a conforming model, within the PSSM subset, to be involved in any meta-associations consistent with both the UML metamodel and any further constraints as defined in this specification.

Note. This approach for defining a subset of the UML abstract syntax is similar to the approach used for defining the metamodel subset covered by a UML profile, in which specially identified PackageImports (`metamodelReferences`) and ElementImports (`metaclassReferences`) are used to import the metaclasses from the subset into the namespace of the Profile (see Clause 12 of [UML]).

The PSSM subset is an extension of the fUML subset (as specified in Clause 7 of [fUML]), and PSSM_Syntax::Syntax directly includes (via package import) all metaclasses from the Classification, SimpleClassifiers, StructuredPackages, Activities and Actions packages from the fUML subset model. It also includes all metaclasses in the CommonStructure, Values and CommonBehavior packages from the fUML subset, but, in these cases the PSSM subset also includes additional metaclasses from the corresponding UML abstract syntax packages that are not included in the fUML subset. Therefore, the PSSM_Syntax::Syntax package contains CommonStructure, Values and CommonBehavior subpackages that import all the metaclasses from the corresponding fUML packages, plus the additional required metaclasses from the UML abstract syntax metamodel (as further described in 7.2, 7.3, and 7.5, respectively), and the content of these subpackages is then further imported into the top-level Syntax package. Finally, the major extension provided by the PSSM subset is the inclusion of metaclasses from the UML StateMachines abstract syntax metamodel package (as described in 7.6), which are first grouped into the PSSMSyntax::Syntax::StateMachines subpackage and then imported into the top-level Syntax package.

The PSSM subset specified here is *not* an extension of the PSCS subset. To satisfy the requirements of the “Joint PSSM and PSCS” conformance level (see 2.2.2), the relevant abstract syntax subset is the *union* of the PSSM subset specified here and the PSCS subset specified in Clause 7 of [PSCS].

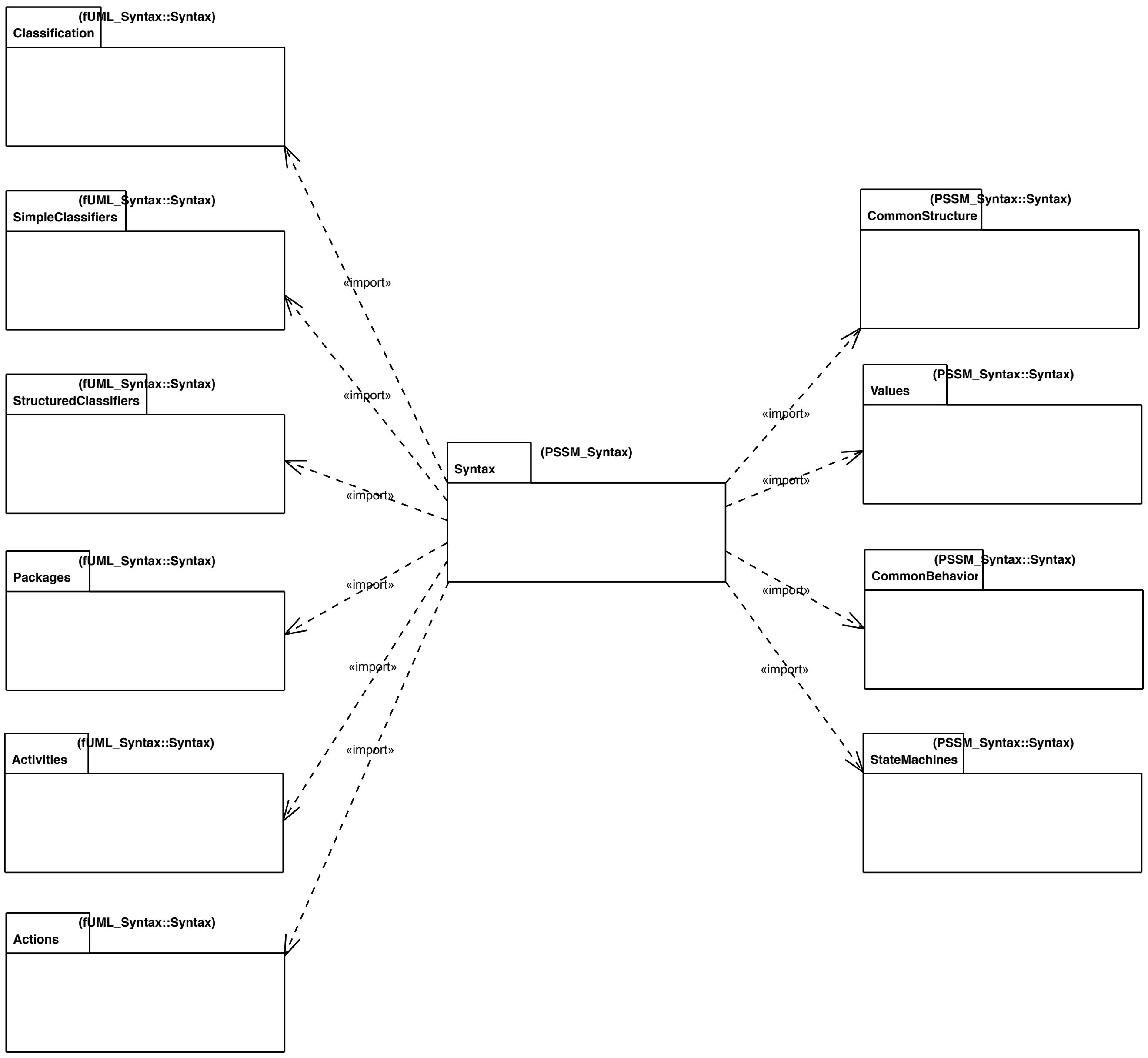


Figure 7.1 - PSSM Syntax Package

In addition to being representable within the PSSM abstract syntax subset, as described above, a UML model that syntactically conforms to PSSM shall also satisfy all relevant constraints defined in the UML abstract syntax metamodel [UML] *and* the additional syntactic constraints specified here for PSSM. The PSSM semantics specified in Clause 8 are only defined for well-formed PSSM models that meet all the necessary constraints. In the case of “Joint PSSM and PSCS” conformance (see 2.2.2), a well-formed model must further meet all the syntactic constraints required for PSCS (see Clause 7 of [PSCS]).

The constraints specified for PSSM are all those that are imported members of the PSSM_Syntax::Constraints package (see Figure 7.2). Each of these constraints has as its single constrained element the UML abstract syntax metaclass to which the constraint applies.

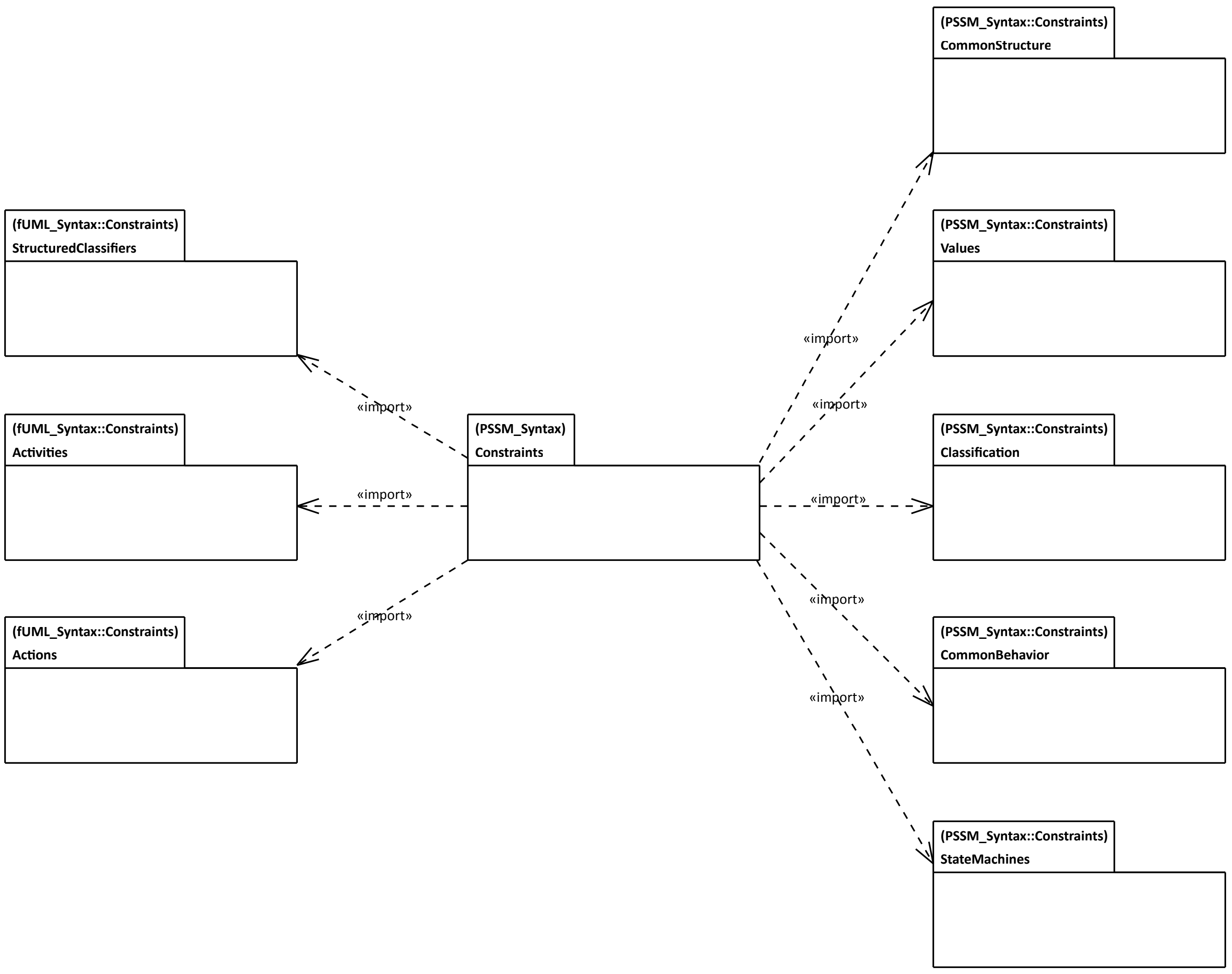

(PSSM_Syntax::Constraints)
CommonStructure
(fUML_Syntax::Constraints)
StructuredClassifiers
(PSSM_Syntax::Constraints)
Values
«import»
«import»
«import»
(fUML_Syntax::Constraints)
Activities
(PSSM_Syntax)
Constraints
(PSSM_Syntax::Constraints)
Classification
«import»
«import»
«import»
«import»
(fUML_Syntax::Constraints)
Actions
(PSSM_Syntax::Constraints)
CommonBehavior
«import»
(PSSM_Syntax::Constraints)
StateMachines

Figure 7.2 - PSSM Constraints Package

The PSSM_Syntax::Constraints package includes (via package import) all the constraints from the fUML constraints packages for StructuredClassifiers, Activities and Actions (see Clause 7 of [fUML]). It also includes all the fUML constraints for CommonStructure, Values and CommonBehavior, but with additional constraints for the additional metaclasses in the PSSM subset in those areas (see 7.2, 7.3, and 7.5, respectively). In addition, in one case (for Classification::Operation), a constraint from fUML is replaced in PSSM with a less restrictive constraint (see 7.4). Finally, additional constraints are included for the StateMachine abstract syntax specific to PSSM.

7.2 Common Structure

7.2.1 Overview

In addition to all the metaclasses included in the fUML subset CommonStructure package, PSSM includes the Constraint metaclass (see Figure 7.3). This metaclass is included in PSSM because the `guard` of a StateMachine Transition is given as a

Constraint (as shown in Figure 7.7). There is also an additional syntactic constraint specified for Constraint, as shown in Figure 7.3 and formally defined in 7.2.2.

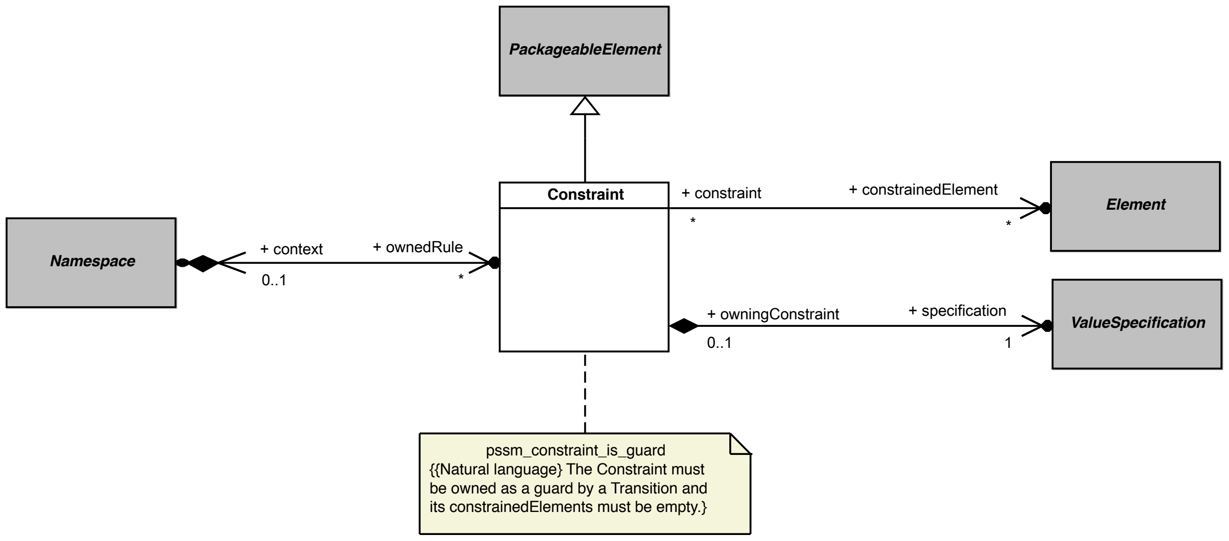


Figure 7.3 - Constraints

7.2.2 Constraints

pssm_constraint_is_guard

A Constraint must be owned as a `guard` by a Transition and its `constrainedElements` must be empty.

```
context UML::CommonStructure::Constraint inv:
    self.owner.oclIsKindOf(UML::StateMachines::Transition) and
    self.constrainedElement->isEmpty()
```

7.3 Values

7.3.1 Overview

In addition to all the metaclasses included in the fUML subset Values package, PSSM includes the Expression and OpaqueExpression metaclasses (see Figure 7.4).

The OpaqueExpression metaclass is included in PSSM in order to provide a way to specify the `specification` of a Constraint used as the `guard` of a StateMachine Transition (as shown in Figure 7.3). However, in order for such a specification to be precise, an OpaqueExpression is constrained to have a behavior that may be executed to provide the result value of the expression (as shown in Figure 7.4 and formally defined in 7.3.2).

The Expression metaclass is also used to specify the `specification` of a `guard` Constraint, but only in the specific case of an “else” `guard` on a Transition outgoing from a junction or choice Pseudostate. Such a `guard` is specified using an Expression whose `symbol` is “else”, with no `operands` (as shown in Figure 7.4 and formally defined in 7.3.2). No other forms of Expression are included in PSSM.

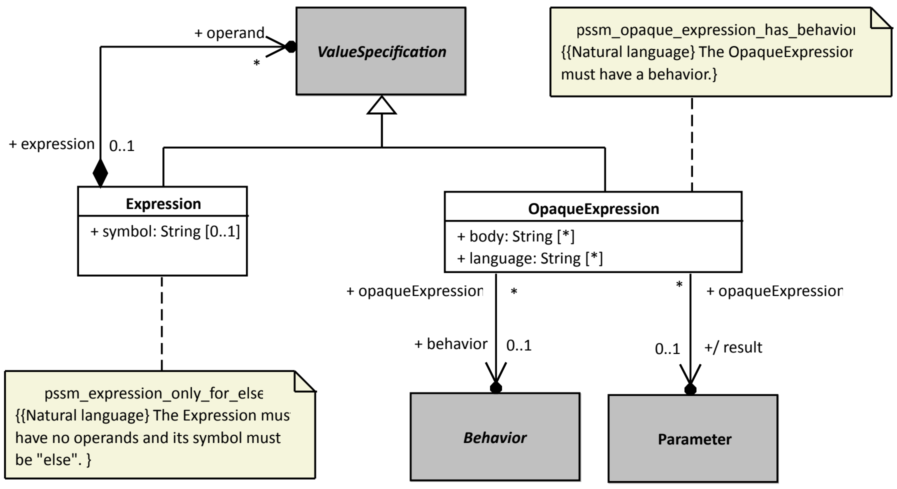


Figure 7.4 - Expressions and OpaqueExpressions

7.3.2 Constraints

pssm_opaque_expression_has_behavior

The OpaqueExpression must have a `behavior`.

```
context UML::Values::OpaqueExpression inv:
    self.behavior <> null
```

pssm_expression_only_for_else

The Expression must have no `operands` and its `symbol` must be “else”.

```
context UML::Values::Expression inv:
    self.symbol = 'else' and self.operand->isEmpty()
```

7.4 Classification

7.4.1 Overview

The PSSM subset includes all the metaclasses in the fUML subset Classification package and does not include any additional ones in this area. However, the fUML constraint fuml_operation_zero_or_one_method requires that a concrete Operation have a single associated `method`. This constraint is too restrictive for PSSM, because PSSM allows an Operation to be handled via a CallEvent `trigger` on a StateMachine Transition, in which case the Operation *cannot* have a method (see 7.5). Therefore, the PSSM_Syntax::Constraints::Classification package imports all the constraints from the corresponding fUML package *except* for the zero_or_one_method constraint, which is replaced with the pssm_operation_has_at_most_one_method constraint (as shown in Figure 7.5 and formally defined in 7.4.2).

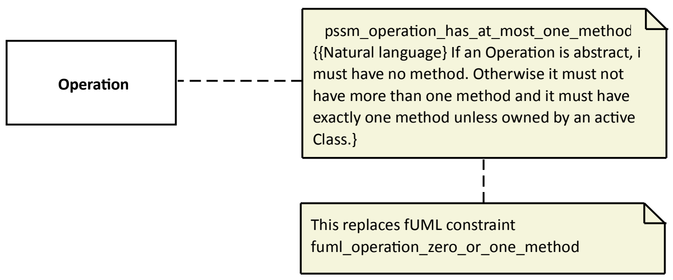

Operation
pssm_operation_has_at_most_one_method
{{Natural language} If an Operation is abstract, i must have no method. Otherwise it must not have more than one method and it must have exactly one method unless owned by an active Class.}
This replaces fUML constraint fuml_operation_zero_or_one_method

Figure 7.5 - Operations

7.4.2 Constraints

pssm_operation_has_at_most_one_method

If an Operation is abstract, it must have no `method`. Otherwise it must not have more than one `method` and it must have exactly one `method` unless owned by an active Class.

```
context UML::Classification::Operation inv:
     if self.isAbstract then self.method->isEmpty()
     else
          self.method->size() <= 1 and
          ((self.class = null or not self.class.isActive) implies
               self.method->size() = 1)
     endif
```

7.5 Common Behavior

7.5.1 Overview

In addition to all the metaclasses included in the fUML subset CommonBehavior package, PSSM includes the CallEvent metaclass (see Figure 7.6). Including CallEvent in the PSSM subset provides the ability for an Operation of a Class in a PSSM conformant model to be handled by a CallEvent `trigger` on a Transition of a StateMachine acting as the `classifierBehavior` of that class (see 7.6.2), rather than be implemented by a `method`. Since the UML specification specifies that having a `method` on an Operation means all calls on the Operation are handled by executing that `method` (see [UML], 13.3.3.2), even if there may also be executing Behaviors with CallEvent Triggers for the Operation, Operations on CallEvents in PSSM are constrained *not* to have a method, in order to avoid confusion. This constraint is shown in Figure 7.6 and formally defined in 7.5.2.

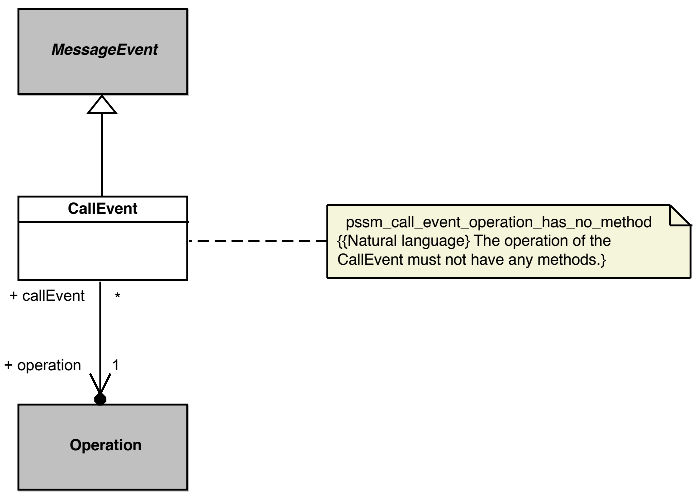

MessageEvent
CallEvent
pssm_call_event_operation_has_no_method
{{Natural language} The operation of the CallEvent must not have any methods.}
+ callEvent
*
+ operation
1
Operation

Figure 7.6 - CallEvents

7.5.2 Constraints

pssm_call_event_operation_has_no_method

The `operation` of the CallEvent must not have any `methods`.

```
context UML::CommonBehavior::CallEvent inv:
     self.operation.method->isEmpty()
```

7.6 State Machines

7.6.1 Overview

Not surprisingly, the largest extension to fUML provided by PSSM is in the area of StateMachines. Within this area, the PSSM subset includes abstract syntax for behavior StateMachines (see 7.6.2) and StateMachine redefinition (see 7.6.3). The formal PSSM subset does *not* include ProtocolStateMachines, they are discussed non-normatively in Annex A.

7.6.2 Behavior State Machines

7.6.2.1 Overview

A behavior StateMachine may be used in a PSSM-conformant model either stand-alone or as the `classifierBehavior` of an active Class. As shown in Figure 7.7, the PSSM subset includes the full UML abstract syntax for behavior StateMachines, *except* for the ConnectionPointReference metaclass. ConnectionPointReferences are used only in relation to submachine States, and such states are not allowed in a PSSM-conformant model. Figure 7.7 shows various additional constraints on StateMachines required for PSSM, including the constraints on the usage of StateMachines and the prohibition on submachine States, all of which are formally defined in 7.6.2.2.

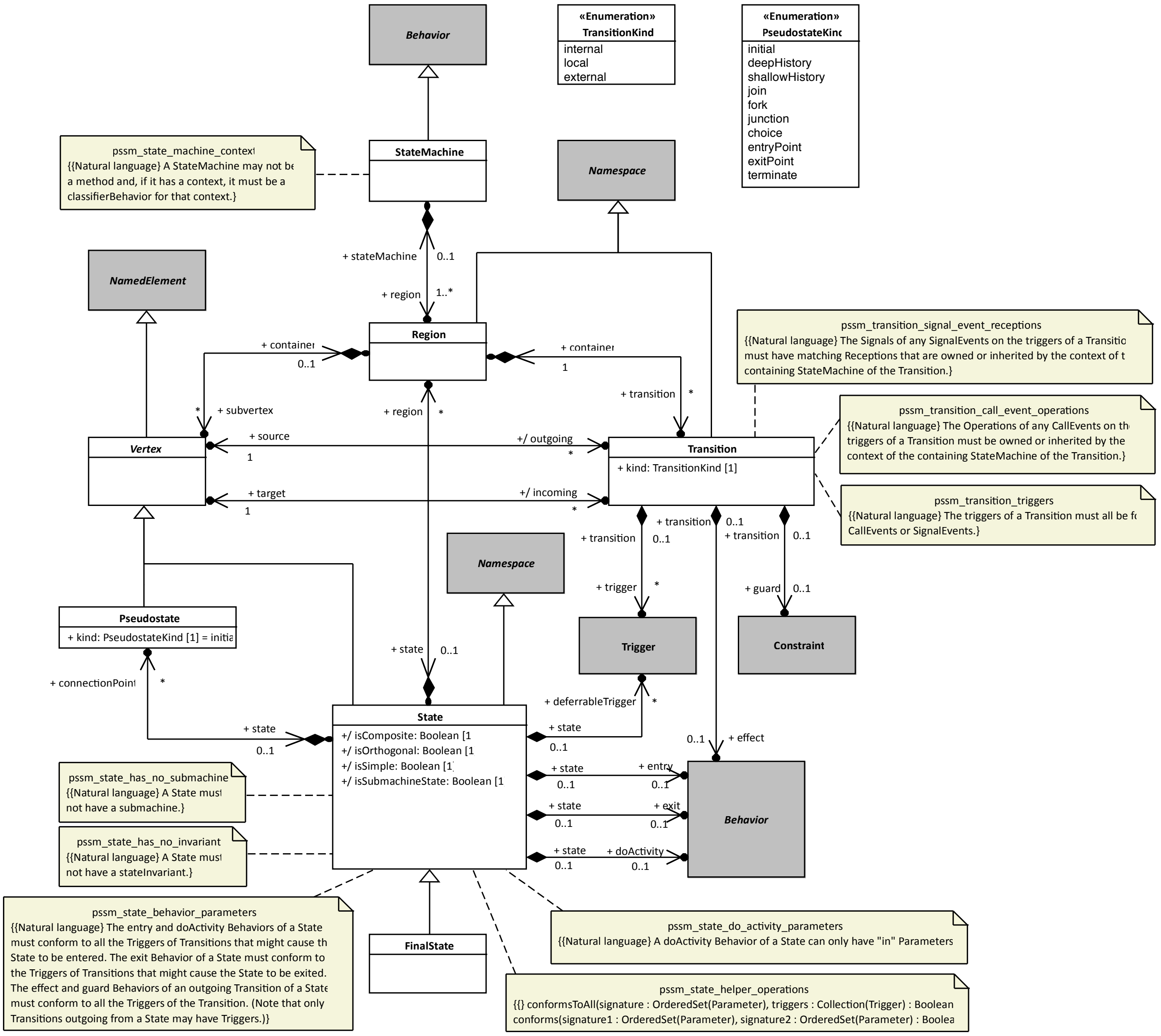

Behavior
«Enumeration» TransitionKind
internal
local
external
«Enumeration» PseudostateKind
initial
deepHistory
shallowHistory
join
fork
junction
choice
entryPoint
exitPoint
terminate
pssm_state_machine_context
{{Natural language} A StateMachine may not be a method and, if it has a context, it must be a classifierBehavior for that context.}
StateMachine
Namespace
+ stateMachine
0..1
+ region
1..*
NamedElement
Region
+ container
0..1
+ container
1
+ transition
*
pssm_transition_signal_event_receptions
{{Natural language} The Signals of any SignalEvents on the triggers of a Transition must have matching Receptions that are owned or inherited by the context of the containing StateMachine of the Transition.}
pssm_transition_call_event_operations
{{Natural language} The Operations of any CallEvents on the triggers of a Transition must be owned or inherited by the context of the containing StateMachine of the Transition.}
pssm_transition_triggers
{{Natural language} The triggers of a Transition must all be for CallEvents or SignalEvents.}
* + subvertex
+ region
*
+ source
1
+/ outgoing
*
Vertex
Transition
+ kind: TransitionKind [1]
+ target
1
+/ incoming
*
+ transition
0..1
+ transition
0..1
+ transition
0..1
+ trigger
*
+ guard
0..1
Namespace
Trigger
Constraint
Pseudostate
+ kind: PseudostateKind [1] = initial
+ connectionPoint
*
+ state
0..1
+ deferrableTrigger
*
+ state
0..1
State
+/ isComposite: Boolean [1]
+/ isOrthogonal: Boolean [1]
+/ isSimple: Boolean [1]
+/ isSubmachineState: Boolean [1]
+ state
0..1
0..1
+ effect
+ state
0..1
+ entry
0..1
+ state
0..1
+ exit
0..1
+ state
0..1
+ doActivity
0..1
Behavior
pssm_state_has_no_submachine
{{Natural language} A State must not have a submachine.}
pssm_state_has_no_invariant
{{Natural language} A State must not have a stateInvariant.}
pssm_state_behavior_parameters
{{Natural language} The entry and doActivity Behaviors of a State must conform to all the Triggers of Transitions that might cause the State to be entered. The exit Behavior of a State must conform to the Triggers of Transitions that might cause the State to be exited. The effect and guard Behaviors of an outgoing Transition of a State must conform to all the Triggers of the Transition. (Note that only Transitions outgoing from a State may have Triggers.)}
FinalState
pssm_state_do_activity_parameters
{{Natural language} A doActivity Behavior of a State can only have "in" Parameters
pssm_state_helper_operations
{{} conformsToAll(signature : OrderedSet(Parameter), triggers : Collection(Trigger) : Boolean
conforms(signature1 : OrderedSet(Parameter), signature2 : OrderedSet(Parameter)) : Boolean

Figure 7.7 - Behavior StateMachines

In particular, the Error: Reference source not found constraint defines the rules for the conformance of the Parameters of `entry`, `doActivity`, and `exit` Behaviors on States and `effect` Behaviors on Transitions outgoing from States with relevant Triggers that might cause those Behaviors to execute. This allows data from event occurrences to be passed to executions of such Behaviors as Parameter values (and, in the case of synchronous calls, for data to be returned, too). Requirements for this event data passing are listed in 9.4.20, and the semantics for data passing are covered in 8.5.10.

7.6.2.2 Constraints

pssm_state_machine_context

A StateMachine may not be a `method` and, if it has a `context`, it must be a `classifierBehavior` for that `context`.

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

pssm_transition_triggers

The triggers of a Transition must all be for CallEvents or SignalEvents.

```
context UML::StateMachines::Transition inv:
     self.trigger.event->forAll(
          oclIsKindOf(UML::CommonBehavior::CallEvent) or
          oclIsKindOf(UML::CommonBehavior::SignalEvent)
     )
```

pssm_transition_call_event_operations

The Operations of any CallEvents on the triggers of a Transition must be owned or inherited by the context of the containing StateMachine.

```
context UML::StateMachines::Transition inv:
     let stateMachine = self.containingStateMachine() in
     let context_ =
          if stateMachine._'context' = null then stateMachine
          else stateMachine._'context'
          endif in
     context_.allFeatures()->includesAll(
          self.trigger->select(oclIsKindOf(UML::CommonBehavior::CallEvent)).
               oclAsType(UML::CommonBehavior::CallEvent).operation
     )
```

pssm_transition_signal_event_receptions

The Signals of any SignalEvents on the `triggers` of a Transition must have matching Receptions that are owned or inherited by the `context` of the containing StateMachine of the Transition.

```
context UML::StateMachines::Transition inv:
     let stateMachine = self.containingStateMachine() in
     let context_ =
          if stateMachine._'context' = null then stateMachine
          else stateMachine._'context'
          endif in
     context_.allFeatures()->select(oclIsKindOf(UML::SimpleClassifiers::Reception)).
     oclAsType(UML::SimpleClassifiers::Reception).signal->includesAll(
          self.trigger->select(oclIsKindOf(UML::CommonBehavior::SignalEvent)).
               oclAsType(UML::CommonBehavior::SignalEvent).signal
     )
```

pssm_state_has_no_submachine

A State must not have a `submachine`.

```
context UML::StateMachines::State inv:
     not self.isSubmachineState
```

pssm_state_has_no_invariant

A State must not have a `stateInvariant`.

```
context UML::StateMachines::State inv:
     self.stateInvariant = null
```

pssm_state_do_activity_parameters

A doActivity Behavior of a State can only have `in` parameters.

```
context UML::StateMachines::State inv:
     self.doActivity <> null implies
          self.doActivity.ownedParameter->forAll(direction = ParameterDirectionKind::_'in')
```

pssm_state_behavior_parameters

The definition of this constraint is given below, followed by the definition of two helper operations it uses.

The `entry` and `doActivity` Behaviors of a State must conform to all the Triggers of Transitions that might cause the State to be entered. The `exit` Behavior of a State must conform to all the Triggers of Transitions that might cause the State to be exited. The `effect` and `guard` Behaviors of an outgoing Transition of a State must conform to all the Triggers of the Transition. (Note that only Transitions outgoing from a State may have triggers.)

```
context UML::StateMachines::State inv:
     -- Collect this State and all containing States.
     let allStates = self->asSet()->closure(container.state) in

     -- Get all the incoming Transitions of the collected States, including
     -- Transitions incoming to entryPoint Pseudostates owned by the States and
     -- all segments of compound Transitions.
     let allIncoming = allStates.incoming->union(
               allStates.connectionPoint->
                    select(kind = UML::StateMachines::PseudostateKind::entryPoint).incoming
          )->asSet()->closure(
               if source.oclIsKindOf(UML::StateMachines::Pseudostate) then source.incoming
               else Set{} endif
          ) in

     -- Get all the outgoing Transitions of the collected States, including
     -- Transitions outgoing from exitPoint Pseudostates owned by the States and
     -- all segments of compound Transitions.
     let allOutgoing = allStates.outgoing->union(
               allStates.connectionPoint->
                    select(kind = UML::StateMachines::PseudostateKind::exitPoint).outgoing
          )->asSet()->closure(
               if target.oclIsKindOf(UML::StateMachines::Pseudostate) then target.outgoing
               else Set{} endif
          ) in

     -- Check the conformance of the various State Behaviors. (Note that
     -- doActivity Behaviors are separately required to have only "in" Parameters.)
     (self.entry <> null implies
          conformsToAll(self.entry.ownedParameter, allIncoming.trigger)) and
     (self.doActivity <> null implies
          conformsToAll(self.doActivity.ownedParameter, allIncoming.trigger)) and
     (self.exit <> null implies
          conformsToAll(self.exit.ownedParameter, allOutgoing.trigger)) and

     -- Check the conformance of the effect and guard Behaviors on outgoing
     -- Transitions. (Note that the behavior on an OpaqueExpression is
     -- separately required to have only "in" Parameters, other than a single
     -- return parameter.)
     allOutgoing->forAll(transition |
          (transition.effect <> null implies
               conformsToAll(transition.effect.ownedParameter, allOutgoing.trigger)) and
          (transition.guard <> null and
           transition.guard.specification.oclIsKindOf(UML::Values::OpaqueExpression) implies
               let behavior = transition.guard.specification.
                    oclAsType(UML::Values::OpaqueExpression).behavior in
               behavior <> null implies
```

```
            conformsToAll(
                behavior.ownedParameter->reject(
                    direction = UML::Classification::ParameterDirectionKind::return),
                allOutgoing.trigger
        ))
    )
```

A signature (set of Parameters) conforms to a collection of Triggers if one of the following is true: the signature is empty; all the Triggers are for SignalEvents and the signature has exactly one Parameter of direction `in`, has multiplicity upper bound of 1 and is either untyped or has a `type` that is a Signal that conforms to all the Signals of the Triggers; or all theTriggers are for CallEvents and the signature conforms to or input-conforms to all the signatures of the Operations of the CallEvents. (A signature input-conforms to another if the first signature conforms to the signature containing only the `in` Parameters from the second signature).

```
conformsToAll(
    signature : OrderedSet(UML::Classification::Parameter),
    triggers : Collection(UML::CommonBehavior::Trigger)) : Boolean =
    signature->isEmpty() or
    triggers.event->forAll(oclIsKindOf(UML::CommonBehavior::SignalEvent)) and
        signature->size() = 1 and
        (let parameter = signature->at(1) in
            parameter.direction = UML::Classification::ParameterDirectionKind::_'in' and
            parameter.is(1,1) and
            (parameter.type = null or
            triggers.event.oclAsType(UML::CommonBehavior::SignalEvent).signal->forAll(s |
                parameter.type.conformsTo(s)
            ))) or
    triggers.event->forAll(oclIsKindOf(UML::CommmonBehavior::CallEvent)) and
        triggers.event.oclAsType(UML::CommmonBehavior::CallEvent).operation->
            forAll(operation |
                conforms(signature, operation.ownedParameter) or
                conforms(signature, operation.ownedParameter->select(
                    direction = UML::Classification::ParameterDirectionKind::_'in'
            )))
```

One signature conforms to another if the first signature has the same number of Parameters as the second signature, and each Parameter of the first signature has the same direction, ordering and uniqueness as the corresponding Parameter (in order) from the second signature and a type and multiplicity that are compatible with those of the corresponding Parameter (depending on the Parameter direction).

```
conforms(
    signature1 : OrderedSet(UML::Classification::Parameter),
    signature2 : OrderedSet(UML::Classification::Parameter)) : Boolean =
    signature1->size() = signature2->size() and
    Sequence{1..signature1->size()} -> forAll(i |
         let parameter1 = signature1->at(i) in
         let parameter2 = signature2->at(i) in

         parameter1.direction = parameter2.direction and
         parameter1.isOrdered = parameter2.isOrdered and
         parameter1.isUnique = parameter2.isUnique and
         (parameter2.direction = UML::Classification::ParameterDirectionKind::_'in' implies
               parameter2.type = null or
                parameter2.type <> null and
                   parameter2.type.conformsTo(parameter1.type) and
               parameter2.compatibleWith(parameter1)) and
         (parameter1.direction = UML::Classification::ParameterDirectionKind::out or
         parameter1.direction = UML::Classification::ParameterDirectionKind::return implies
               parameter1.type = null or
               parameter1.type <> null and
                   parameter1.type.conformsTo(parameter2.type) and
              parameter1.compatibleWith(parameter2)) and
         (parameter1.direction = UML::Classification::ParameterDirectionKind::inout implies
             parameter1.type = parameter2.type and
             parameter2.compatibleWith(parameter1) and
               parameter1.compatibleWith(parameter2))
    )
```

7.6.3 State Machine Redefinition

7.6.3.1 Overview

The capability for StateMachine redefinition actually does not require any other metaclasses than those already included for behavior StateMachines. However, for clarity, the diagram for StateMachine redefinition from the UML specification is repeated here, showing the additional meta-associations involved (Figure 7.8). This diagram also shows the additional constraint (formally defined in 7.6.3.2) required for the PSSM semantics, which only support the ability of a StateMachine to extend at most one other StateMachine.

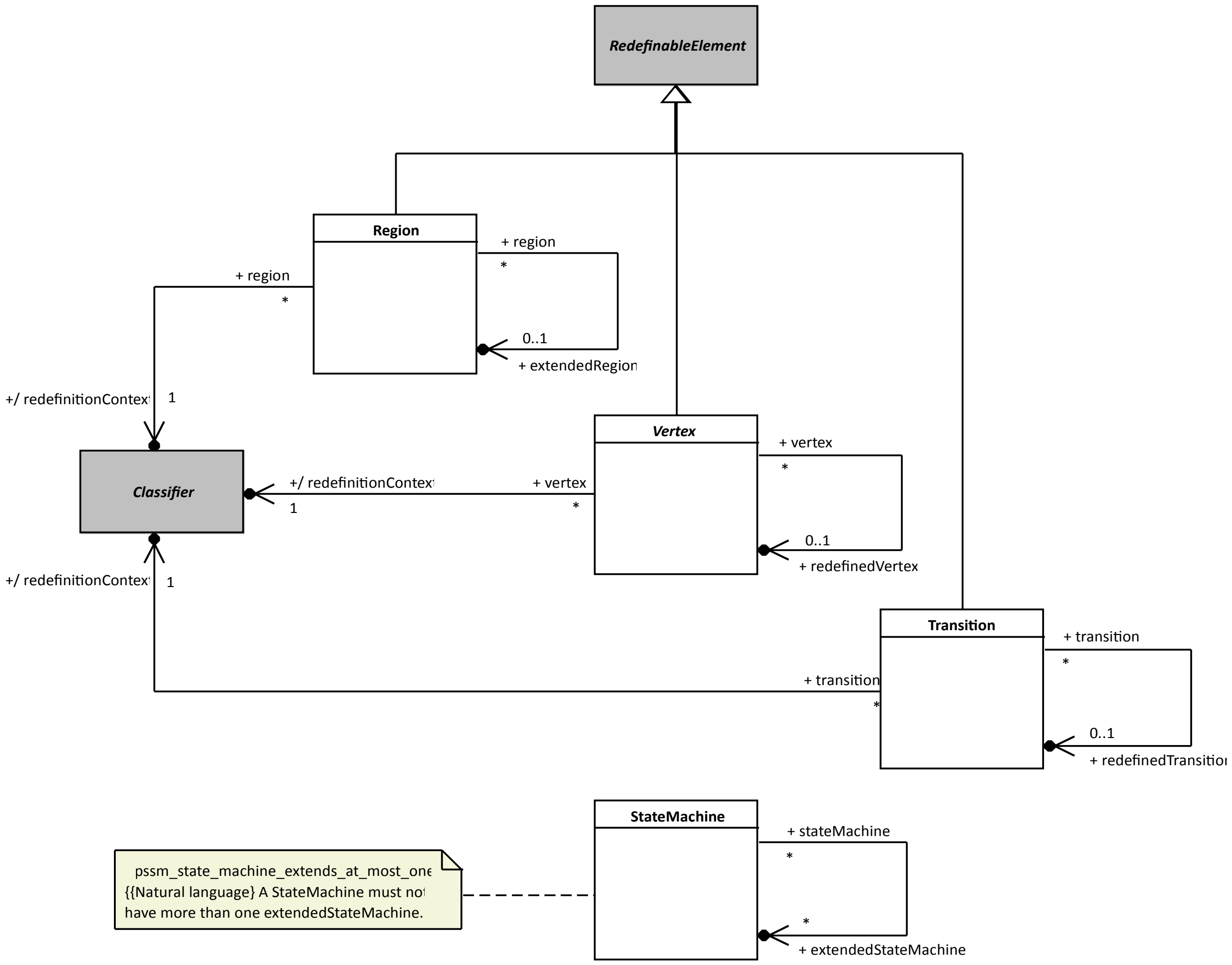


Figure 7.8 - StateMachine Redefinition

7.6.3.2 Constraints

pssm_state_machine_extends_at_most_one

A StateMachine must not have more than one `extendedStateMachine`.

```
context UML::StateMachines::StateMachine inv:
    self.extendedStateMachine->size() <= 1
```

8 Execution Model

8.1 Overview

This clause defines the precise semantics of the abstract syntax subset specified in Clause 7. This semantic definition is given as an extension to the semantic model for PSCS (see [PSCS], Clause 8), which is itself an extension of the execution model for fUML (see [fUML], Clause 8). This clause includes only the extensions to the PSCS model necessary for PSSM. However, the full semantics for PSSM are given by the fUML execution model as extended for both PSCS and PSSM, which is then a complete, executable fUML model of the operational semantics for the combined PSCS and PSSM subset.

The PSSM execution model is given as an extension of the PSCS model in order to ensure that PSSM semantics are compatible with PSCS semantics. However, the only point at which the PSSM semantic functionality actually depends on PSCS is in the definition of the behavior of Triggers that reference one or more Ports, using the Trigger `port` property (see [UML], 13.3). An execution tool that conforms at the "PSSM-only Conformance" level (see 2.2.1) is not required to implement the `port` functionality (since the PSSM-only abstract syntax subset does not include Ports), and none of the rest of the semantic functionality for PSSM depends on the functionality provided by the PSCS execution model extensions. Therefore, a tool conforming at the "PSSM-only" level can effectively treat the PSSM execution model as directly extending the fUML execution model, ignoring all inherited PSCS-specific functionality. A tool conforming at the "Joint PSSM and PSCS Conformance" level (see 2.2.2), on the other hand, must implement the semantics as specified in the entire extension of the fUML execution model by both PSCS and as given for PSSM in this clause.

The circularity of defining PSSM semantics by extending the fUML execution model, which is itself an fUML model, is handled as it is in fUML. That is, the execution model is defined using only the further subset of fUML whose semantics are separately specified by the fUML base semantics (see [fUML], Clause 10), which do not need to be extended further for the purposes of PSSM. This further subset, known as *Base UML* (or "bUML") includes a subset of UML activity modeling that is used to specify the detailed behavior of all concrete operations in the execution model. However, rather than using activity diagram notation to represent such activity models, they are specified in the execution model extensions for PSSM using the Java-syntax textual notation whose mapping to UML is given in Annex A of [fUML].

The PSSM extensions to the PSCS execution model are organized into six packages, which are named according to corresponding UML abstract syntax packages. Figure 8.1 shows each of these packages and their dependencies on packages from the fUML, PSCS and PSSM syntactic and semantic models. These dependencies are represented as package-import relationships, which also make the unqualified names of the necessary syntactic and semantics elements visible for use in the detailed behavioral code of each of the PSSM semantics packages. Each PSSM semantic package also publicly exports all its imported members, allowing those packages to be organized in a layered fashion. This layering starts with the small semantic extensions defined in the Values, StructuredClassifiers and CommonBehavior packages, which are used in defining the primary PSSM semantics in the StateMachines package and one extension to the actions semantics in the Action package, all of which are then fully integrated into the semantic infrastructure of the fUML execution model in the Loci package.

The subsequent subclauses in this clause describe each of the PSSM semantics packages in turn. The description includes a class model for the contents of the package and an explanation of the operational semantics defined by the functionality of the classes in the model. The detailed behavior code for the operations of the classes is not included in this document, but can be found in the associated normative XMI file for the PSSM semantic model that is described in this clause.

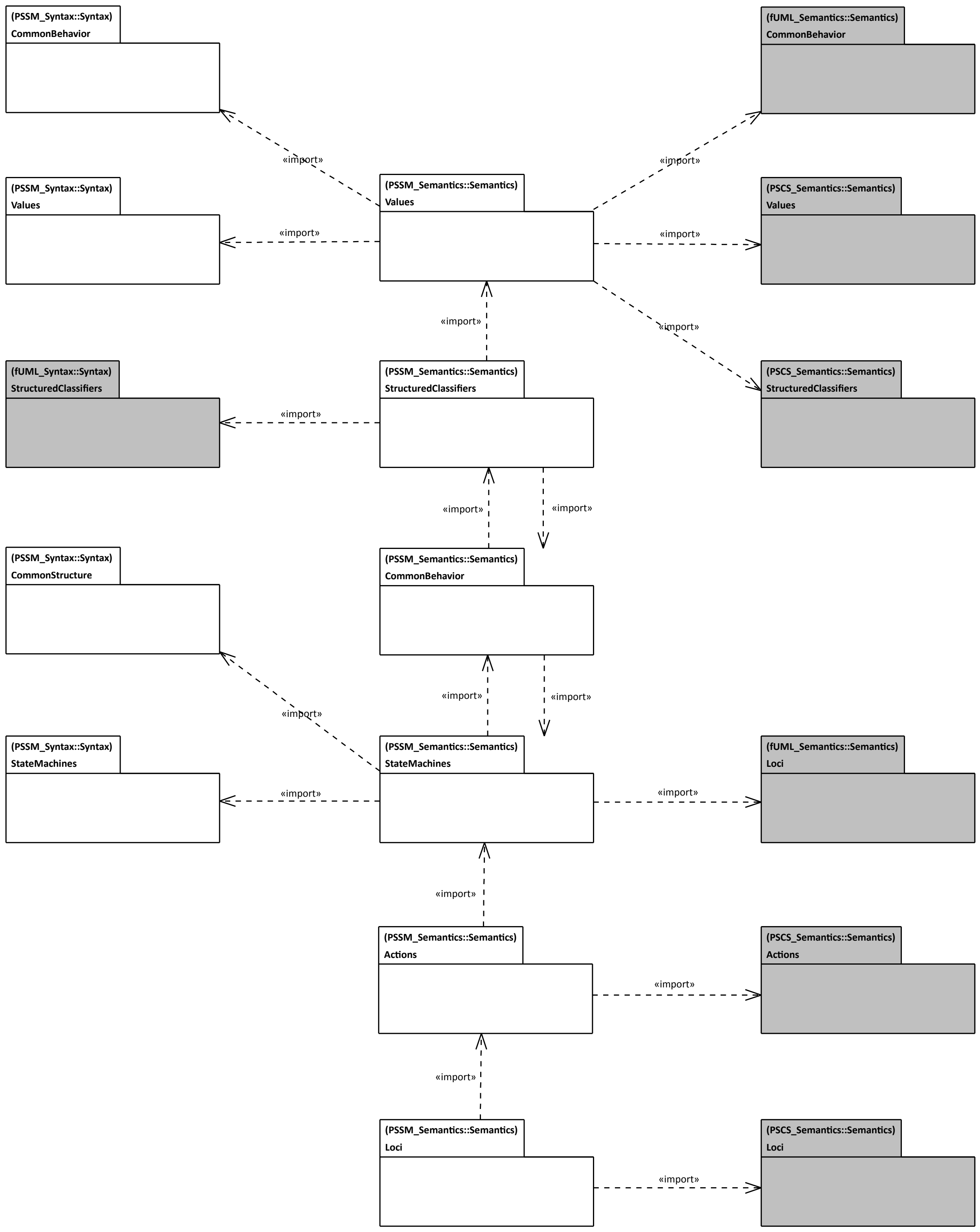

(PSSM_Syntax::Syntax) CommonBehavior
(fUML_Semantics::Semantics) CommonBehavior
(PSSM_Syntax::Syntax) Values
(PSSM_Semantics::Semantics) Values
(PSCS_Semantics::Semantics) Values
(fUML_Syntax::Syntax) StructuredClassifiers
(PSSM_Semantics::Semantics) StructuredClassifiers
(PSCS_Semantics::Semantics) StructuredClassifiers
(PSSM_Syntax::Syntax) CommonStructure
(PSSM_Semantics::Semantics) CommonBehavior
(PSSM_Syntax::Syntax) StateMachines
(PSSM_Semantics::Semantics) StateMachines
(fUML_Semantics::Semantics) Loci
(PSSM_Semantics::Semantics) Actions
(PSCS_Semantics::Semantics) Actions
(PSSM_Semantics::Semantics) Loci
(PSCS_Semantics::Semantics) Loci
«import»

Figure 8.1 - PSSM Semantics Package

8.2 Values

The Values package in the PSSM abstract syntax subset extends the Values package from the fUML abstract syntax by adding the OpaqueExpression metaclass (see 7.3). As shown in Figure 8.2, the semantics for OpaqueExpressions in PSSM is defined by the corresponding SemanticVisitor class SM_OpaqueExpressionEvaluation. This class is a specialization of the CS_OpaqueExpressionEvaluation class defined in the PSCS semantics (see [PSCS], 8.3.1.2.2).

An OpaqueExpression in PSSM is required to have an associated `behavior`. As in PSCS, the evaluation of an OpaqueExpression consists in executing its associated `behavior`. The functionality for this is provided by the `executeExpressionBehavior` operation defined in the CS_OpaqueExpressionEvaluation class. The PSSM SM_OpaqueExpressionEvaluation adds a new `context` attribute and redefines the `executeExpressionBehavior` operation such that the OpaqueExpression `behavior` is executed with the given context Object. This extension is necessary to allow an OpaqueExpression used as the specification of a `guard` Constraint on a StateMachine Transition to be evaluated in the proper context for that StateMachine (see 8.5.8).

In addition, as of UML 2.5.1, UML allows the `behavior` of an OpaqueExpression to optionally have `in` Parameters (in addition to a mandatory `return` Parameter). In PSSM, if the `behavior` of an OpaqueExpression used to specify a `guard` Constraint has `in` Parameters, then these Parameters are used to pass event data into the Behavior execution (see 8.5.10. The `initialize` operation is used to create `parameterValues` on an SM_OpaqueExpressionEvaluation corresponding to the data available from an EventOccurrence (either the SignalInstance from a SignalEventOccurrence or the input ParameterValues from a CallEventOccurrence). Then, when `executeExpressionBehavior` is called, these `parameterValues` are passed to the newly created Behavior Execution.

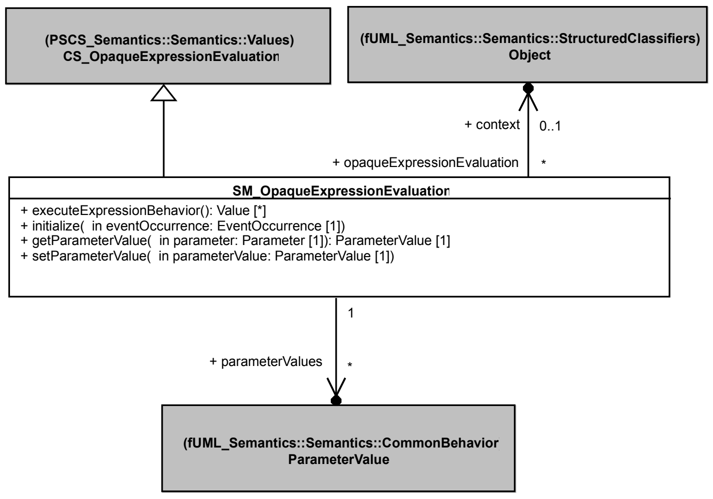

(PSCS_Semantics::Semantics::Values)
CS_OpaqueExpressionEvaluation
(fUML_Semantics::Semantics::StructuredClassifiers)
Object
+ context
0..1
+ opaqueExpressionEvaluation
*
SM_OpaqueExpressionEvaluation
+ executeExpressionBehavior(): Value [*]
+ initialize(in eventOccurrence: EventOccurrence [1])
+ getParameterValue(in parameter: Parameter [1]): ParameterValue [1]
+ setParameterValue(in parameterValue: ParameterValue [1])
1
+ parameterValues
*
(fUML_Semantics::Semantics::CommonBehavior
ParameterValue

Figure 8.2 - Values Extension

8.3 Structured Classifiers

The PSSM abstract syntax subset does not extend the StructuredClassifiers package from the fUML abstract syntax. However, the PSSM execution model StructuredClassifiers package includes a specialization of the semantics of Objects, as shown in Figure 8.3. The SM_Object class redefines the `startBehavior` operation from the fUML Object class (which is inherited without change by the PSCS CS_Object class) such that, when the Behavior of an active Object is started, an SM_ObjectActivation (as defined in 8.4) is instantiated for it, rather than the usual fUML ObjectActivation.

SM_Object also redefines the `destroy` operation from the fUML Object class to ensure that, when an Object is destroyed, any EventOccurrences remaining in the `eventPool` of the ObjectActivation are also removed, before carrying out the functionality of stopping the ObjectActivation for the Object and removing the Object from its Locus. This avoids the possibility of the event-dispatch loop of the ObjectActivation still getting a next event even once the Object has been removed from the Locus.

The SM_RedefinitionBasedDispatchStrategy class redefines the `dispatch` and `getMethod` operations from the fUML RedefinitionBasedDispatchStrategy class. The redefined `getMethod` operation has the same functionality as the fUML operation except that it returns null if an Operation does not have any associated `method` (rather than this being an error, as in fUML). The redefined `dispatch` operation handles the case in which `getMethod` returns a null value by creating a CallEventExecution (see 8.5.9). In any other case, the `dispatch` operation behaves as in fUML and PSCS: it creates an Execution for the resolved `method` Behavior of the given Operation.

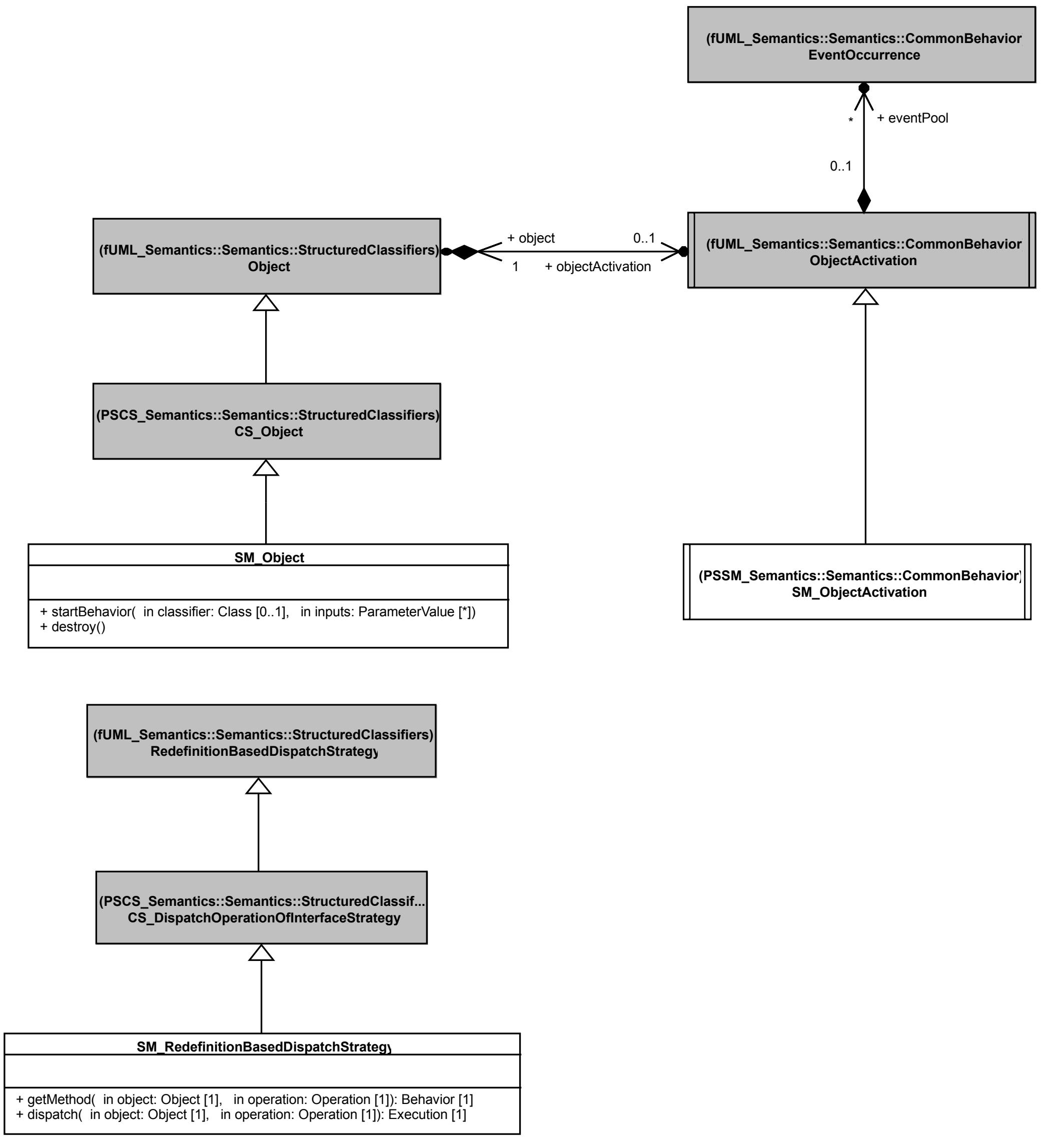

(fUML_Semantics::Semantics::CommonBehavior
EventOccurrence
*
+ eventPool
0..1
(fUML_Semantics::Semantics::StructuredClassifiers)
Object
+ object
0..1
1
+ objectActivation
(fUML_Semantics::Semantics::CommonBehavior
ObjectActivation
(PSCS_Semantics::Semantics::StructuredClassifiers)
CS_Object
SM_Object
+ startBehavior(in classifier: Class [0..1], in inputs: ParameterValue [*])
+ destroy()
(PSSM_Semantics::Semantics::CommonBehavior)
SM_ObjectActivation
(fUML_Semantics::Semantics::StructuredClassifiers)
RedefinitionBasedDispatchStrategy
(PSCS_Semantics::Semantics::StructuredClassif...
CS_DispatchOperationOfInterfaceStrategy
SM_RedefinitionBasedDispatchStrategy
+ getMethod(in object: Object [1], in operation: Operation [1]): Behavior [1]
+ dispatch(in object: Object [1], in operation: Operation [1]): Execution [1]

Figure 8.3 - StructuredClassifiers Extension

8.4 Common Behavior

The CommonBehavior package in the PSSM abstract syntax subset extends the CommonBehavior package from the fUML abstract syntax by adding the CallEvent metaclass (see 7.5). However, the semantics of CallEvent is defined as part of the semantics of the triggering of StateMachine Transitions (see 8.5.8), not in the CommonBehavior semantics. Instead, as shown in Figure 8.4, the CommonBehavior package in the PSSM execution model includes the SM_ObjectActivation class, which specializes the ObjectActivation class from fUML (see [fUML], 8.4.3.2.7). The SM_Object class in the StructuredClassifiers package of the PSSM execution model provides the functionality for instantiating an SM_ObjectActivation instead of a regular ObjectActivation when the Behavior of an active Object is started (see 8.3).

The SM_ObjectActivation class adds semantics for handling two types of EventOccurrences that are specific to StateMachines: CompletionEventOccurrence and DeferredEventOccurrence (see 8.5.9). To do this, the class redefines operations provided by the fUML ObjectActivation class and also adds new attributes and operations.

The new `deferredEventPool` contains the set of DeferredEventOccurrences that are deferred in the current configuration of a StateMachine, which is used in the specification of the semantics of the `deferredEvents` of a State (see 8.5.5)

The `getNextEvent` is redefined to extend the way that events are retrieved from the `eventPool` to account for CompletionEventOccurrences and DeferredEventOccurrences that may be in the pool, as follows:

- While there are CompletionEventOccurrences in the `eventPool`, they are dispatched before any other EventOccurrences. The dispatching order is the order in which the CompletionEventOccurrences were added to the `eventPool`.
- When there are no remaining CompletionEventOccurrences in the `eventPool`, then regular EventOccurrences are dispatched according to the chosen GetNextEventStrategy (see 8.4.3.1 in [fUML]). EventOccurrences are handled as in fUML, except for DeferredEventOccurrences, for which the EventOccurrence that is returned is the one that is referenced by the DeferredEventOccurrence (which is the actual EventOccurrence that was originally deferred).

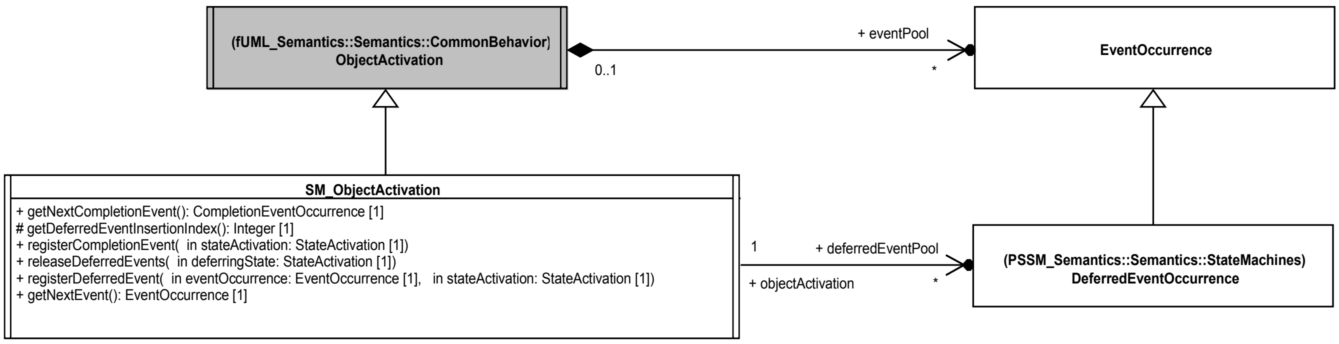

(fUML_Semantics::Semantics::CommonBehavior)
ObjectActivation
+ eventPool
0..1
*
EventOccurrence
SM_ObjectActivation
+ getNextCompletionEvent(): CompletionEventOccurrence [1]
# getDeferredEventInsertionIndex(): Integer [1]
+ registerCompletionEvent(in stateActivation: StateActivation [1])
+ releaseDeferredEvents(in deferringState: StateActivation [1])
+ registerDeferredEvent(in eventOccurrence: EventOccurrence [1], in stateActivation: StateActivation [1])
+ getNextEvent(): EventOccurrence [1]
1
+ deferredEventPool
+ objectActivation
*
(PSSM_Semantics::Semantics::StateMachines)
DeferredEventOccurrence

Figure 8.4 - CommonBehavior Extension

The SM_ObjectActivation class also adds the following operations:

- The `registerCompletionEvent` operation is used to add a CompletionEventOccurrence to the `eventPool` of the SM_ObjectActivation when the activation of a StateMachine State completes (see 8.5.5). When added to the `eventPool`, a CompletionEventOccurrence is always placed after all CompletionEventOccurrences already in the pool.

- The `registerDeferredEvent` operation is used to add an EventOccurrence o be deferred by a StateActivation to the `deferredEventPool` of the SM_ObjectActivation. The EventOccurrence that is deferred is wrapped in a DeferredEventOccurrence and added at the end of the `deferredEventPool`.
- The `releaseDeferredEvent` operation is used to release all EventOccurrences that were deferred by a StateActivation. The DeferredEventOccurrences are removed from the `deferredEventPool` and added to the regular `eventPool`. All events returning to the regular `eventPool` are placed in that pool after all existing CompletionEventOccurrences, but before any other EventOccurrence already in the pool, in the order in which the DeferredEventOccurrences had in the deferredEventPool.

The PSSM CommonBehavior package also contains the EventTriggeredExecution class, a specialization of the fUML Execution class that is used as part of the model for passing data from an EventOccurrence to the Executions of Behaviors within a StateMachine that may have been invoked due to the handling of that EventOccurrence. This class is discussed in 8.5.10.1 as part of the description of StateMachine semantics.

8.5 State Machines

8.5.1 Overview

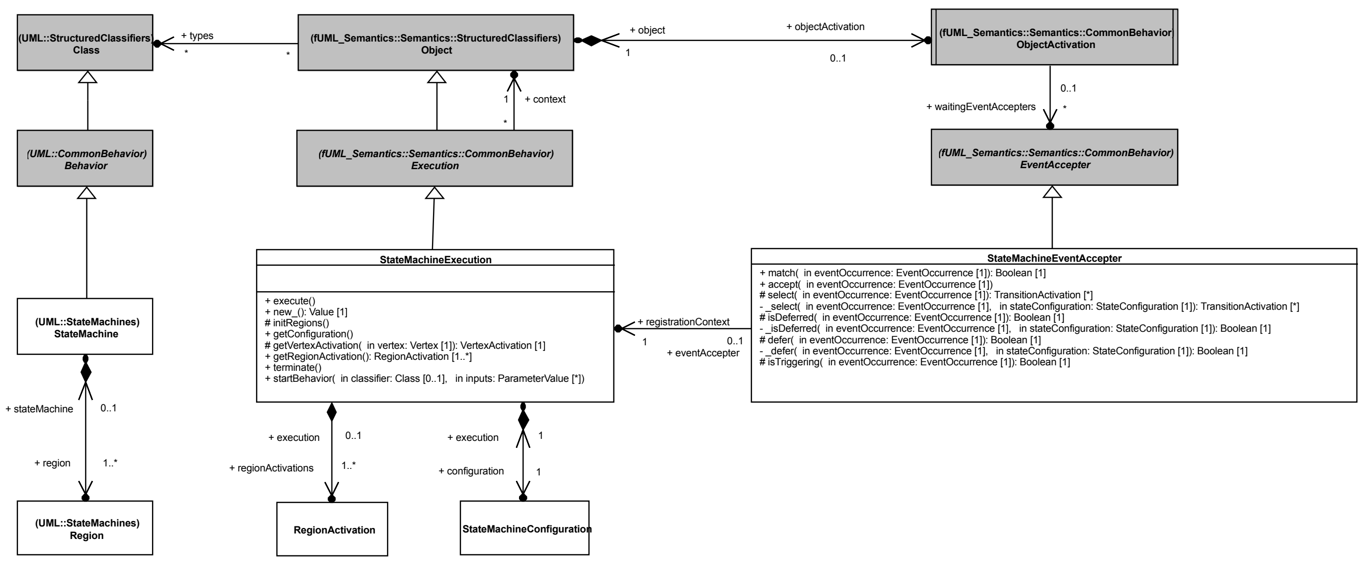
The StateMachines package of the PSSM abstract syntax (see 7.6) defines the subset of the UML abstract syntax for StateMachines that is covered by the PSSM semantics. This subset includes primarily the syntax for so-called behavior StateMachines (see 7.6.2), the primary kind of executable StateMachine included in UML. The semantics for behavior StateMachines are modeled in the StateMachines package of the PSSM execution model, which is described in this subclause (8.5).

The PSSM subset also includes the additional meta-associations from the UML abstract syntax required in models that use StateMachine redefinition (see 7.6.3). The semantics for StateMachine redefinition is included in the functionality provided by the RegionActivation in the StateMachines package of the PSSM execution model (see 8.5.3).

Finally, the UML abstract syntax also includes the syntax for ProtocolStateMachines (see [UML], 14.4). However, the operational execution of ProtocolStateMachines requires the raising of exceptions when the protocol defined by the StateMachine is violated, but the semantics of exceptions are not currently included in fUML. Therefore, the PSSM syntax subset does not include ProtocolStateMachines, and the PSSM execution model does not include formal operational semantics for them. Instead, Annex A gives a non-normative description of the semantics that is more precise than that given in the UML specification, but without a formal execution model.

8.5.2 State Machine Execution

Figure 8.5 shows the root classes in the StateMachines package of the PSSM execution model related to the execution of a StateMachine.



(UML::StructuredClassifiers) Class
+ types
*
*
(fUML_Semantics::Semantics::StructuredClassifiers) Object
+ object
1
+ objectActivation
0..1
(fUML_Semantics::Semantics::CommonBehavior) ObjectActivation
0..1
+ waitingEventAccepters
*
1
+ context
*
(UML::CommonBehavior) Behavior
(fUML_Semantics::Semantics::CommonBehavior) Execution
(fUML_Semantics::Semantics::CommonBehavior) EventAccepter
StateMachineExecution
+ execute()
+ new_(): Value [1]
# initRegions()
+ getConfiguration()
# getVertexActivation(in vertex: Vertex [1]): VertexActivation [1]
+ getRegionActivation(): RegionActivation [1..*]
+ terminate()
+ startBehavior(in classifier: Class [0..1], in inputs: ParameterValue [*])
+ registrationContext
1
0..1
+ eventAccepter
StateMachineEventAccepter
+ match(in eventOccurrence: EventOccurrence [1]): Boolean [1]
+ accept(in eventOccurrence: EventOccurrence [1])
# select(in eventOccurrence: EventOccurrence [1]): TransitionActivation [*]
- _select(in eventOccurrence: EventOccurrence [1], in stateConfiguration: StateConfiguration [1]): TransitionActivation [*]
# isDeferred(in eventOccurrence: EventOccurrence [1]): Boolean [1]
- _isDeferred(in eventOccurrence: EventOccurrence [1], in stateConfiguration: StateConfiguration [1]): Boolean [1]
# defer(in eventOccurrence: EventOccurrence [1]): Boolean [1]
- _defer(in eventOccurrence: EventOccurrence [1], in stateConfiguration: StateConfiguration [1]): Boolean [1]
# isTriggering(in eventOccurrence: EventOccurrence [1]): Boolean [1]
(UML::StateMachines) StateMachine
+ stateMachine
0..1
+ region
1..*
(UML::StateMachines) Region
+ execution
0..1
+ regionActivations
1..*
RegionActivation
+ execution
1
+ configuration
1
StateMachineConfiguration

Figure 8.5 - StateMachineExecution

StateMachineExecution

In the fUML execution model, the abstract Execution class represents that execution of any kind of Behavior (see [fUML], 8.4.2.1.1). The Execution class is specialized to ActivityExecution to specify the semantics of Activities in fUML. Similarly, the PSSM execution model includes a StateMachineExecution class that acts as the root element for specifying the execution semantics for StateMachines.

In the UML abstract syntax, Behaviors are a kind of Class. Correspondingly, in the fUML execution model, Executions are a kind of Object, and the Behavior being Executed is associated as the (single) `type` of the Execution, considered as an Object. For a StateMachineExecution, this `type` will always be a StateMachine.

A StateMachine is composed of one or more Regions (see 7.6.2.1). The semantics of the Regions in a StateMachine are captured in corresponding RegionActivation classes associated with a StateMachine execution for the StateMachine (see 8.5.3). The StateMachineExecution is responsible for creating a RegionActivation for each of the Regions of its StateMachine, and the RegionActivations then create (in a cascade) SemanticVisitors for all their contained elements.

The execution of a StateMachine starts when the `execute` operation is called on a StateMachineExecution for it. Since a StateMachine is always invoked asynchronously in fUML, the CommonBehavior semantics of fUML (see [fUML], 8.4.3) ensure that the invocation of the `execute` operation of a StateMachineExecution will always take place as part of a run-to-completion (RTC) step for an initial InvocationEventOccurrence for the StateMachineExecution.

The execution of the StateMachine then proceeds by concurrently entering all RegionActivations of the StateMachineExecution (as discussed in 8.5.3). The initial RTC step completes once the StateMachine has reached a *stable configuration.*

The current `configuration` of a StateMachineExecution is represented as an instance of the StateMachineConfiguration class. A StateMachineConfiguration represents the hierarchy of active States that the StateMachineExecution currently is in (as discussed further in 8.5.4). A `configuration` is *stable* once all the Transitions triggered in an RTC step have been traversed and any invoked `entry` Behaviors have completed (see [UML], 14.2.3.4.2). This `configuration` is used to determine how the StateMachineExecution will proceed in response to subsequent dispatched EventOccurrences accepted by the StateMachineExecution.

A StateMachineExecution *completes* when all its RegionActivations have themselves completed. A StateMachineExecution may also be *terminated* when a terminate Pseudostate is reached, regardless of its level of nesting. The termination of a StateMachineExecution implies the termination of all of its RegionActivations, as captured in the behavior of the `terminate` operation of the StateMachineExecution class. This operation is also called when a StateMachine is terminated due to its context Object being destroyed.

StateMachineEventAccepter

Once the initial RTC step has completed for a StateMachineExecution, the StateMachineExecution will generally need to be able to handle EventOccurrences for Events linked to Triggers on the Transitions of the StateMachine being executed. The fUML CommonBehavior semantics provides a general model for the registration of EventAccepters with the ObjectActivation of an active Object, in order to allow an Execution to respond to EventOccurrences dispatched from the `eventPool` for that ObjectActivation (see [fUML], 8.4.3). A StateMachineExecution uses this mechanism by registering a single, specialized StateMachineEventAccepter instance with the ObjectActivation of its `context` Object.

Note. In the fUML execution model, each AcceptEventActionActivation that fires within an ActivityExecution will register its own AcceptEventActionEventAccepter with the ObjectActivation of the context of the ActivityExecution (see [fUML], 8.6.4.2.1 and 8.6.4.2.2). Thus, an ActivityExecution can potentially have several registered EventAccepters associated with it at any one time. In contrast, an executing StateMachineExecution will always have exactly one registered StateMachineEventAccepter associated with it. The reason for this is that, in order to account for priorities, conflicts, etc. between Transitions between these active States, how a StateMachineExecution responds to any specific EventOccurrence requires an analysis of the entire current StateMachineConfiguration. It is therefore not possible to associate separate, independent EventAccepters with, say, each individual Transition within the StateMachine being executed.

The fUML EventAccepter class has two abstract operations, `match` and `accept`, whose concrete behavior must be provided by any concrete subclass of EventAccepter. The `match` operation is used to determine if the EventAccepter is able to `accept` a given EventOccurrence. If the EventAccepter does match an EventOccurrence, and is chosen to actually handle that EventOccurrence, then an RTC step for handling the EventOccurrence is initiated by calling the `accept` operation on the chosen EventAccepter.

The specified behavior of the match and accept operations for a StateMachineEventAccepter rely on the association of the StateMachineEventAccepter with its registrationContext, that is, the StateMachineExecution that originally registered the StateMachineEventAccepter. A StateMachineEventAccepter will match a dispatched EventOccurrence in the following situations:

1. The EventOccurrence is deferred in the current StateMachineConfiguration of the `registrationContext`. In this case, if the EventOccurrence is subsequently accepted by the StateMachineEventAccepter, it is placed in the `deferredEventPool` for the ObjectActivation (which, therefore, must be an SM_ObjectActivation, as described in 8.4).

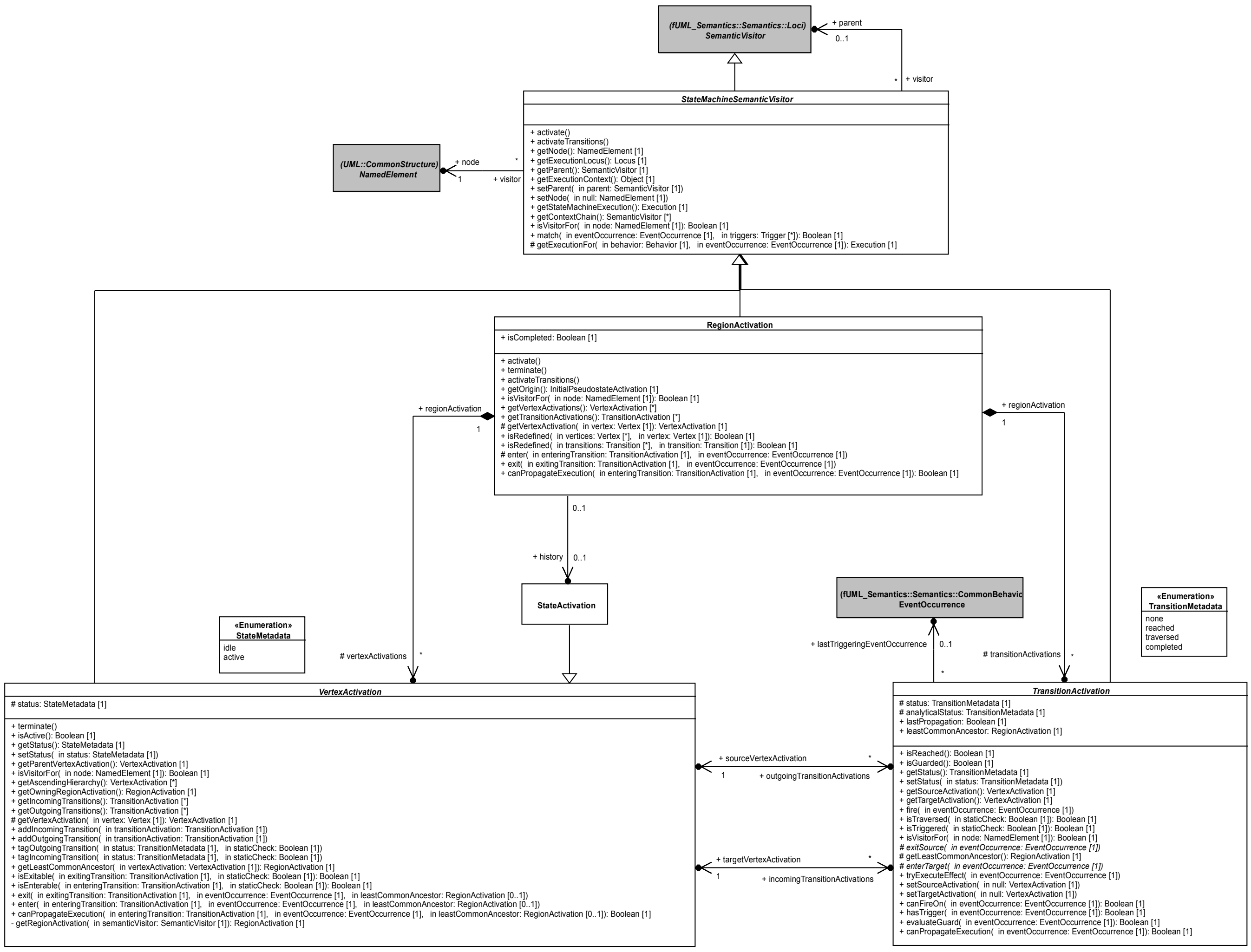
2. The EventOccurrence triggers one or more Transitions in the StateMachine of the `registrationContext`. In this case, if the EventOccurrence is subsequently accepted by the StateMachineEventAccepter, the functionality of the SemanticVisitors associated with various elements of the StateMachine being executed results in the StateMachineExecution moving to a new stable `configuration` during the course of the RTC step (as described in 8.5.3 and following subclauses).

The above two situations are identified by the analysis of the StateMachineConfiguration. This analysis is based on a recursive algorithm that starts from the most nested StateActivations referenced as being active in the current StateMachineConfiguration. This enables the algorithm to account for Transition priorities, which are relative to the level of nesting of their source States. The StateMachineEventAccepter class provides two main operations dedicated to the analysis of the StateMachineConfiguration: `isDeferred` and `isTriggering`. The `isDeferred` operation returns true if the proposed EventOccurrence must be deferred in the current StateMachineConfiguration. The `isTriggering` operation returns true if the proposed EventOccurrence triggers at least one Transition in the current StateMachineConfiguration.

Both operations rely on the `select` operation, which is responsible for building the set of Transitions that can be fired using the proposed EventOccurrence. This set only contains Transitions that lead from the current StateMachineConfiguration to a valid StateMachineConfiguration. Indeed, before being placed in the set of Transitions to be fired, an entire compound Transition is analyzed to determine if it is possible to find at least one valid path to a target StateMachineConfiguration.

8.5.3 State Machine Semantic Visitors

Figure 8.6 shows the base SemanticVisitors introduced in the PSSM semantic model to specify the semantics of the various elements of a StateMachine. VertexActivation captures the basic semantics for Vertexes, and TransitionActivation captures the basic semantics for Transitions. These visitors are both further specialized in the semantic model to respectively capture semantics of different kinds of Vertexes and Transitions. VertexActivations and TransitionActivations are always owned by a RegionActivation, which captures the semantics of the Region that owns the corresponding Vertexes and Transitions.



(fUML_Semantics::Semantics::Loci)
SemanticVisitor
+ parent
0..1
*
+ visitor
StateMachineSemanticVisitor
+ activate()
+ activateTransitions()
+ getNode(): NamedElement [1]
+ getExecutionLocus(): Locus [1]
+ getParent(): SemanticVisitor [1]
+ getExecutionContext(): Object [1]
+ setParent(in parent: SemanticVisitor [1])
+ setNode(in null: NamedElement [1])
+ getStateMachineExecution(): Execution [1]
+ getContextChain(): SemanticVisitor [*]
+ isVisitorFor(in node: NamedElement [1]): Boolean [1]
+ match(in eventOccurrence: EventOccurrence [1],   in triggers: Trigger [*]): Boolean [1]
# getExecutionFor(in behavior: Behavior [1],   in eventOccurrence: EventOccurrence [1]): Execution [1]
(UML::CommonStructure)
NamedElement
+ node
1
*
+ visitor
RegionActivation
+ isCompleted: Boolean [1]
+ activate()
+ terminate()
+ activateTransitions()
+ getOrigin(): InitialPseudostateActivation [1]
+ isVisitorFor(in node: NamedElement [1]): Boolean [1]
+ getVertexActivations(): VertexActivation [*]
+ getTransitionActivations(): TransitionActivation [*]
# getVertexActivation(in vertex: Vertex [1]): VertexActivation [1]
+ isRedefined(in vertices: Vertex [*],   in vertex: Vertex [1]): Boolean [1]
+ isRedefined(in transitions: Transition [*],   in transition: Transition [1]): Boolean [1]
# enter(in enteringTransition: TransitionActivation [1],   in eventOccurrence: EventOccurrence [1])
+ exit(in exitingTransition: TransitionActivation [1],   in eventOccurrence: EventOccurrence [1])
+ canPropagateExecution(in enteringTransition: TransitionActivation [1],   in eventOccurrence: EventOccurrence [1]): Boolean [1]
+ regionActivation
1
+ regionActivation
1
0..1
+ history
0..1
StateActivation
«Enumeration»
StateMetadata
idle
active
# vertexActivations
*
(fUML_Semantics::Semantics::CommonBehavic
EventOccurrence
+ lastTriggeringEventOccurrence
0..1
*
# transitionActivations
*
«Enumeration»
TransitionMetadata
none
reached
traversed
completed
VertexActivation
# status: StateMetadata [1]
+ terminate()
+ isActive(): Boolean [1]
+ getStatus(): StateMetadata [1]
+ setStatus(in status: StateMetadata [1])
+ getParentVertexActivation(): VertexActivation [1]
+ isVisitorFor(in node: NamedElement [1]): Boolean [1]
+ getAscendingHierarchy(): VertexActivation [*]
+ getOwningRegionActivation(): RegionActivation [1]
+ getIncomingTransitions(): TransitionActivation [*]
+ getOutgoingTransitions(): TransitionActivation [*]
# getVertexActivation(in vertex: Vertex [1]): VertexActivation [1]
+ addIncomingTransition(in transitionActivation: TransitionActivation [1])
+ addOutgoingTransition(in transitionActivation: TransitionActivation [1])
+ tagOutgoingTransition(in status: TransitionMetadata [1],   in staticCheck: Boolean [1])
+ tagIncomingTransition(in status: TransitionMetadata [1],   in staticCheck: Boolean [1])
+ getLeastCommonAncestor(in vertexActivation: VertexActivation [1]): RegionActivation [1]
+ isExitable(in exitingTransition: TransitionActivation [1],   in staticCheck: Boolean [1]): Boolean [1]
+ isEnterable(in enteringTransition: TransitionActivation [1],   in staticCheck: Boolean [1]): Boolean [1]
+ exit(in exitingTransition: TransitionActivation [1],   in eventOccurrence: EventOccurrence [1],   in leastCommonAncestor: RegionActivation [0..1])
+ enter(in enteringTransition: TransitionActivation [1],   in eventOccurrence: EventOccurrence [1],   in leastCommonAncestor: RegionActivation [0..1])
+ canPropagateExecution(in enteringTransition: TransitionActivation [1],   in eventOccurrence: EventOccurrence [1],   in leastCommonAncestor: RegionActivation [0..1]): Boolean [1]
- getRegionActivation(in semanticVisitor: SemanticVisitor [1]): RegionActivation [1]
+ sourceVertexActivation
*
1
+ outgoingTransitionActivations
+ targetVertexActivation
*
1
+ incomingTransitionActivations
TransitionActivation
# status: TransitionMetadata [1]
# analyticalStatus: TransitionMetadata [1]
+ lastPropagation: Boolean [1]
+ leastCommonAncestor: RegionActivation [1]
+ isReached(): Boolean [1]
+ isGuarded(): Boolean [1]
+ getStatus(): TransitionMetadata [1]
+ setStatus(in status: TransitionMetadata [1])
+ getSourceActivation(): VertexActivation [1]
+ getTargetActivation(): VertexActivation [1]
+ fire(in eventOccurrence: EventOccurrence [1])
+ isTraversed(in staticCheck: Boolean [1]): Boolean [1]
+ isTriggered(in staticCheck: Boolean [1]): Boolean [1]
+ isVisitorFor(in node: NamedElement [1]): Boolean [1]
# exitSource(in eventOccurrence: EventOccurrence [1])
# getLeastCommonAncestor(): RegionActivation [1]
# enterTarget(in eventOccurrence: EventOccurrence [1])
+ tryExecuteEffect(in eventOccurrence: EventOccurrence [1])
+ setSourceActivation(in null: VertexActivation [1])
+ setTargetActivation(in null: VertexActivation [1])
+ canFireOn(in eventOccurrence: EventOccurrence [1]): Boolean [1]
+ hasTrigger(in eventOccurrence: EventOccurrence [1]): Boolean [1]
+ evaluateGuard(in eventOccurrence: EventOccurrence [1]): Boolean [1]
+ canPropagateExecution(in eventOccurrence: EventOccurrence [1]): Boolean [1]

Figure 8.6 - StateMachine SemanticVisitors

StateMachineSemanticVisitor

A StateMachineSemanticVisitor is an fUML SemanticVisitor for an element within a StateMachine (as opposed to the StateMachine itself, whose SemanticVisitor is a StateMachineExecution, as discussed in 8.5.2). A StateMachineSemanticVisitor is actually generically associated with a NamedElement, because this is the most specialized kind of UML syntax element that is common to all the elements within a StateMachine that need to be given semantics (e.g., Regions, Vertexes and Transitions). However, as specified for each of the various kinds of StateMachineSemanticVisitor in the following, the `node` of each kind of StateMachineSemanticVisitor will always be a corresponding kind of StateMachine element.

A StateMachineSemanticVisitor may also generically have another SemanticVisitor as its `parent` (which will be either itself a kind of StateMachineSemanticVisitor or a StateMachineExecution). The parent-child hierarchy of the StateMachineSemanticVisitors for a StateMachine reflects the hierarchical organization of the StateMachine syntactic elements associated with those StateMachineSemanticVisitors. For example, the StateActivations for all States in a Region will have the RegionActivation for that Region as their `parent`, and the RegionActivations for all Regions in a composite State will have the StateActivation for that State as their `parent`. Ultimately, this tree structure of StateMachineSemanticVisitors is rooted in the StateMachineExecution for the StateMachine being executed.

The StateMachineSemanticVisitor class also takes advantage of the generic tree-structured hierarchy of StateMachineSemanticVisitors for a StateMachineExecution in order to provide certain utility operations that are inherited by all specialized StateMachineSemanticVisitors.

- The `getStateMachineExecution` operation returns the StateMachineExecution at the root of the tree.
- The `getExecutionContext` operation returns the context object of the root StateMachineExecution.
- The `getExecutionLocus` operation return the locus at which the root StateMachineExecution resides (in fUML, every Object, including every Execution, resides at a specific Locus; see [fUML], 8.2.2.2.6).
- The `isVisitorFor` operation returns true if the current StateMachineSemanticVisitor is a SemanticVisitor for the given NamedElement. By default ,this operation returns true if the `node` of the StateMachineSemanticVisitor is the same as that given as the argument to the operation. This default functionality is overridden in certain StateMachineSemantics Visitor subclasses.
- The `getExecutionFor` operation returns an Execution for the behavior provided in Parameter. If an EventOccurrence is also passed to this operation, the returned Execution is an EventTriggeredExecution (see 8.5.10.1), which is able to pass any data embedded in the given EventOccurrence to the Behavior to be executed. Otherwise, the returned Execution is the usual kind for the given Behavior (e.g. an ActivityExecution if the Behavior is an Activity).

Each specialized kind of StateMachineSemanticVisitor adds functionality to the base StateMachineSemanticVisitor class to capture the semantics of a specific kind of StateMachine element. These semantics are always split into two distinct parts:

1. After a StateMachineSemanticVisitor is instantiated, it is activated by calling the `activate` and `activateTransitions` operations. These operations are redefined in each kind of StateMachineSemanticVisitor in order to specify the appropriate activation semantics.
2. Additional operations are defined for each kind of StateMachineSemanticVisitor in order to specify the execution semantics specific to the kind of StateMachine element associated with that kind of StateMachineSemanticVisitor.

RegionActivation

A RegionActivation captures the semantics of a Region. Thus, the `node` of a RegionActivation is always a Region. The instances of all other kinds of StateMachineSemanticVisitors are always contained in a RegionActivation.

Using functionality added to the `activate` and `activateTransitions` operations, a RegionActivation instantiates all visitors required to execute model elements contained in the associated Region. Hence, during the execution of a StateMachine, a RegionActivation owns a set of VertexActivations and TransitionActivations that are the visitors for the Vertexes and Transitions contained in the associated Region.

A RegionActivation is entered by calling the `enter` operation and exited by calling the `exit` operation. A RegionActivation may be entered or exited implicitly or explicitly.

Entering a RegionActivation

A RegionActivation can be entered either implicitly or explicitly.

- An implicit entry consists in starting the Region execution using the initial Pseudostate (if any) and firing its continuation Transition. Note that if the initial Pseudostate does not exist, then the RegionActivation is considered as being immediately completed. This leads the execution to properly ignore the execution of that particular RegionActivation.
- An explicit entry occurs when the Region is entered via a Transition with a `source` outside the Region and a `target` inside the Region. The Region execution does not then start with an initial Pseudostate. Instead, the RegionActivation is considered to be entered when the VertexActivation for the target Vertex internal to the Region is entered. Note that this case can only happen if the RegionActivation is owned by a StateActivation. The owning StateActivation will always have been entered before any of its RegionActivations are actually started (see 8.5.5).

Exiting a RegionActivation

A RegionActivation can be exited either implicitly or explicitly.

- An implicit exit of one or more RegionActivations occurs when a TransitionActivation exits a StateActivation for a composite state. In this case, all RegionActivations currently executing in the StateActivation are exited. This implies that all VertexActivations located within the RegionActivation are also exited. The exiting sequence for each RegionActivation starts by exiting the most nested VertexActivations.
- An explicit exit of a RegionActivation occurs when a TransitionActivation exits a VertexActivation located in that RegionActivation and the target VertexActivation is located outside the RegionActivation. In this case, the RegionActivation that is exited explicitly starts the exiting sequence using the source VertexActivation (note that if the StateActivation is for a composite state, then active VertexActivation(s) located within are exited first). Other RegionActivations (if any) start their exiting phase using their innermost VertexActivations.

The final point of exiting either implicitly or explicitly one or more RegionActivations owned by a StateActivation consists in executing the `exit` behavior (if any) attached to the associated State and traversing the exiting Transition.

Completion of a region activation

RegionActivations never reach completion by being exited either implicitly or explicitly. There are two ways to complete the execution of a region.

1. The general rule is that a RegionActivation can only complete if a FinalStateActivation (see 8.5.5) for a FinalState owned by the Region is executed. This leads the RegionActivation to be marked as being completed (its `isCompleted` attribute is set to true).
2. The above general rule is violated only in the situation where a VertexActivation owned by a RegionActivation is exited and the TransitionActivation that exits that VertexActivation has as its target the StateActivation owning the RegionActivation. In this case, and only in this case, does the RegionActivation that owns the exited VertexActivation complete.

Termination of a RegionActivation

A RegionActivation can be terminated (using its `terminate` operation). The termination of a RegionActivation occurs as the result of the termination of the StateMachineExecution. It consists in terminating all VertexActivations owned by the RegionActivation. Finally, the terminated VertexActivations are destroyed.

History of a RegionActivation

The `history` StateActivation associated with a RegionActivation is the last known StateActivation in that RegionActivation. This `history` is non-empty when the RegionActivation is exited while a non-final StateActivation is active, in which case it can be used to restore the RegionActivation if it is re-entered via a shallow or deep history Pseudostate (see 8.5.7.4). The `history` of a RegionActivation is updated in two situations:

1. A StateActivation (other than a FinalStateActivation) that is directly owned by the RegionActivation is entered. During its entry sequence, the StateActivation updates the `history` of its containing RegionActivation to itself, the StateActivation being entered.

2. A FinalStateActivation that is directly owned by the RegionActivation is entered. During its entry sequence, the FinalStateActivation removes any `history` the RegionActivation might have. If a FinalState is reached, a Region is considered to have no history.

Extension and RegionActivation

A Region can extend another Region using redefinition (see 7.6.3.1). In this case, the RegionActivation for the extension Region also acts as the visitor for all the Regions directly and indirectly redefined by the extension Region.

For example, suppose that Region R1 is extended by Region R2, which is itself extended by Region R3. The RegionActivation instantiated for R3 is then not only the visitor for R3, but also a visitor for R2 and R1. To make this possible, the RegionActivation class redefines the `isVisitorFor` operation.

Further, the RegionActivation for an extension Region not only instantiates visitors for the Vertices and Transitions directly owned by the extension Region, but also for Vertices and Transitions owned by any extended Region but are not redefined in the extension Region. In this way, the RegionActivation constructs a set of visitors that represents an effective “dynamic merge” of all the extended Regions with the extension Region. This set of visitors is then used to perform the interpretation of the extension Region, just as if the visitors had been instantiated for elements directly owned by the Region.

Evaluation of a RegionActivation

During the static analysis of compound Transitions that takes place when a StateMachineEventAccepter checks the matching of a particular EventOccurrence, the analysis of Transitions and Vertices within a Region is handled by the `canPropagateExecution` operation of the RegionActivation for that Region. The following two situations can be encountered:

1. The `target` of the Transition that is used to enter the Region is an internal Vertex of that Region. This means that the Region is going to be entered explicitly, so no implicit path starting from an initial Pseudostate needs to be evaluated.

2. The `target` of the Transition that is used to enter the Region is an internal Vertex of that Region. This means that the static analysis must be propagated through the PseudoStateActivation for the initial Pseudostate owned by the Region. If the propagation of the static analysis through this path is acceptable, then the path is also valid for the Region. Conversely if this propagation is not acceptable, then the path is considered as being invalid for the Region.

VertexActivation

VertexActivation is the base class for all StateMachineSemanticVisitors capturing semantics of specializations of Vertex (i.e., State, FinalState and Pseudostate). A VertexActivation is always owned by a RegionActivation. It is associated with a set of TransitionActivations for the `outgoing` Transitions of its Vertex and another set for the incoming Transitions.

VertexActivations and StateMachineConfiguration

A VertexActivation captures the status of a Vertex. The Vertex is either *idle* or *active*. In essence, if the Vertex is a State then to be *active,* the State must be in the current StateMachine configuration. Conversely, if the State is not in the StateMachine `configuration`, it is *idle*.

VertexActivation entry and exit

The VertexActivation class defines the common way to enter and exit any kind of Vertex.

- A Vertex can only be entered if its prerequisites (specific to each kind of Vertex, based on its redefinition of the VertexActivation `isEnterable` operation) have been fulfilled. In this case, and only in this case, can the VertexActivation be entered (using its `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.
- 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 its `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 before it.

VertexActivation termination

The VertexActivation class does not enforce a particular termination semantics. These semantics are specific to each subclass of VertexActivation. They are captured through the different redefinitions of the `terminate` operation.

Evaluation of a VertexActivation

During static analysis of compound Transitions that takes place when a StateMachineEventAccepter checks the matching of a particular EventOccurrence, the analysis of a path that traverses a specific Vertex is handled by the `canPropagateExecution` operation of the VertexActivation for that Vertex. This consists of propagating the analysis to the parent VertexActivation. The propagation in the parent is constrained by the common ancestor computed between the current VertexActivation and the VertexActivation that was the `source` of the entering Transition. As long as the common ancestor is not encountered, the analysis continues to be propagated to the parent. The verdict of the analysis is the verdict of the propagation made to the parent VertexActivation. Subclasses of VertexActivation add further functionality to the static analysis by redefining the `canPropagateExecution` operation.

TransitionActivation

A TransitionActivation is the base class for all StateMachineSemanticVisitors capturing Transition semantics. TransitionActivations have the responsibility to link VertexActivations that capture Vertex semantics. A TransitionActivation references the VertexActivation for the `source` and `target` Vertices of its Transition. It also has a status that defines the current situation of the Transition. For instance, `reach` means that the TransitionActivations originates from a VertexActivation that is currently active.

Evaluation of a TransitionActivation

A TransitionActivation can evaluate the guard of its associated Transition (using its `evaluateGuard` operation), if there is one on the visited transition. It is also capable of determining if the Transition can be triggered by a specific EventOccurrence (using its `canFireOn` operation). These evaluation semantics are common to all kinds of TransitionActivations.

The evaluation sequence of a TransitionActivation always takes place during the analysis of the StateMachineConfiguration. In addition to determining whether the Transition has a `trigger` matching the dispatched EventOccurrence and a `guard` evaluating to true, the evaluation checks that, if the TransitionActivation is fired, the result will be a valid StateMachineConfiguration. If this is not the case, the TransitionActivation will not be included in the set of TransitionActivations to be fired in the next run-to-completion step.

The `canPropagate` operation of the TransitionActivation class is responsible for propagating the static analysis to the `target` VertexActivation of the TransitionActivation. It is required that the static analysis can be propagated through the `target` VertexActivation, meaning a valid path has been found by the static analysis, so the TransitionActivation can be part of the set of TransitionActivation to be fired.

In addition to the `canPropagate` operation, the TransitionActivation class also maintains additional information to deal with the static analysis:

1. The `analyticalStatus` attribute captures the status of the static analysis of the TransitionActivation.

2. The `lastTriggeringEventOccurrence` association references the last EventOccurrence that was used during the static analysis of the TransitionActivation. This enables the detection of whether the TransitionActivation was already explored using the same EventOccurrence.

3. The `lastPropagation` flag captures the result of the last static-analysis propagation when this TransitionActivation was explored. If the TransitionActivation was explored using the same EventOccurrence, then the current analysis can simply return the previous result, rather than performing a detailed analysis again.

Firing

The Transition firing sequence is also common to all kinds of TransitionActivations (using the `fire` operation). It always consists of the following steps:

1. The Transition `source` may be exited. This depends on the kind of Transition that is actually firing. The exit sequence performed by the exited VertexActivation depends on the type of the `source` Vertex.

2. The `effect` Behavior of the Transition is always executed.

3. The Transition `target` may be entered. This depends on the kind of Transition that is actually firing. The entry sequence performed by the entered VertexActivation depends on the type of the `target` Vertex.

A firing sequence can thus be viewed as a chain of calls:

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

```
end
```

The `fire` operation of a TransitionActivation initiates the exit sequence of the `source` VertexActivation. Assuming that the prerequisites of the `source` Vertex are fulfilled, the exit sequence consists of a call to the `exit` behavior of the `source` VertexActivation. This exit sequence can nest a number of `exit` calls. These nested calls propagate the exit sequence to parent VertexActivations as long as the common ancestor of the `source` and the `target` VertexActivations has not been reached. As soon as the exit sequence is terminated, if the Transition has an associated `effect` Behavior, it is executed. After the execution of this Behavior, the `target` VertexActivation is entered via a call to its `entry` Behavior (assuming that the prerequisites for entering the `target` VertexActivation are fulfilled). This call can lead to a number of nested `enter` calls that are used to enter parent VertexActivations before the actual `target` VertexActivation is entered. The call nesting ends when the common ancestor between the `source` and the `target` VertexActivations is reached.

Note. The triggering of the exit or entry sequence of a VertexActivation depends on the kind of Transition. Each specialization of a TransitionActivation is intended to provide the appropriate semantics be redefining the operations `exitSource` and `enterTarget`.

8.5.4 State Machine Configuration

A StateMachineExecution always has an associated StateMachineConfiguration. As shown in Figure 8.7, this configuration represents the hierarchy of active States in which the currently executed StateMachine is.

The view that is provided through the StateMachineConfiguration offers a simple way to evaluate:

- Transitions that can be fired using the dispatched EventOccurrence.
- If the currently dispatched EventOccurrence can be deferred.

The StateMachineConfiguration is always evaluated through the StateMachineEventAccepter that is registered in the ObjectActivation attached to the StateMachineExecution context. It determines the way that dispatched EventOccurrences are handled.

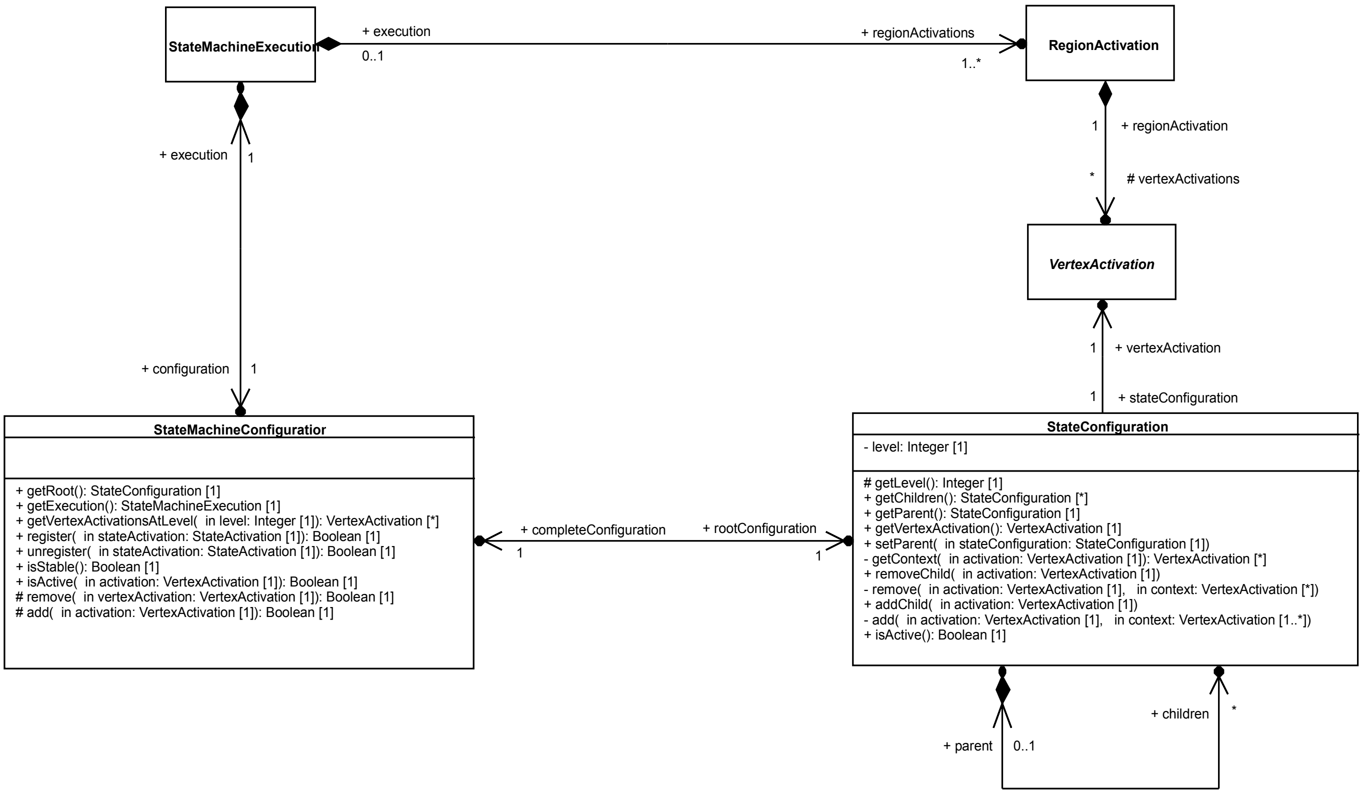

StateMachineExecution
+ execution
0..1
+ regionActivations
1..*
RegionActivation
+ execution
1
1
+ regionActivation
*
# vertexActivations
VertexActivation
1
+ vertexActivation
1
+ stateConfiguration
+ configuration
1
StateMachineConfiguration
+ getRoot(): StateConfiguration [1]
+ getExecution(): StateMachineExecution [1]
+ getVertexActivationsAtLevel(in level: Integer [1]): VertexActivation [*]
+ register(in stateActivation: StateActivation [1]): Boolean [1]
+ unregister(in stateActivation: StateActivation [1]): Boolean [1]
+ isStable(): Boolean [1]
+ isActive(in activation: VertexActivation [1]): Boolean [1]
# remove(in vertexActivation: VertexActivation [1]): Boolean [1]
# add(in activation: VertexActivation [1]): Boolean [1]
+ completeConfiguration
1
+ rootConfiguration
1
StateConfiguration
- level: Integer [1]
# getLevel(): Integer [1]
+ getChildren(): StateConfiguration [*]
+ getParent(): StateConfiguration [1]
+ getVertexActivation(): VertexActivation [1]
+ setParent(in stateConfiguration: StateConfiguration [1])
- getContext(in activation: VertexActivation [1]): VertexActivation [*]
+ removeChild(in activation: VertexActivation [1])
- remove(in activation: VertexActivation [1], in context: VertexActivation [*])
+ addChild(in activation: VertexActivation [1])
- add(in activation: VertexActivation [1], in context: VertexActivation [1..*])
+ isActive(): Boolean [1]
+ children
*
+ parent
0..1

Figure 8.7 - StateMachineConfiguration

StateMachineConfiguration

The StateMachineConfiguration attached to a StateMachineExecution is modified either when a StateActivation is entered or when a StateActivation is exited (via its `register` and `unregister` operations, respectively).

Note that a StateMachineConfiguration does not evolve between run-to-completion steps. A StateMachineConfiguration only evolves during a run-to-completion step initiated via the acceptance of an event occurrence dispatched from the event pool.

The internal structure of a StateMachineConfiguration is represented as a hierarchy of StateConfigurations. Each StateConfiguration included in the hierarchy actually references a VertexActivation that is active in the currently executed StateMachine.

Note. The additional level of nesting introduced by the presence of RegionActivations is not captured by the StateMachineConfiguration. Nevertheless it is inherent in the StateMachineConfiguration tree structure, since each branch of the tree denotes the presence of a Region.

StateConfiguration

A StateConfiguration is a basic unit of a StateMachineConfiguration, representing the membership of a VertexActivation in the configuration of the executed StateMachine. Each StateConfiguration has a single `parent` StateConfiguration (if any) and may have zero or more children StateConfiguration(s).

8.5.5 State Activations

Figure 8.8 shows the SemanticVisitor StateActivation and its further specialization, FinalStateActivation. StateActivation captures simple and composite State semantics while FinalStateActivation captures FinalState semantics.

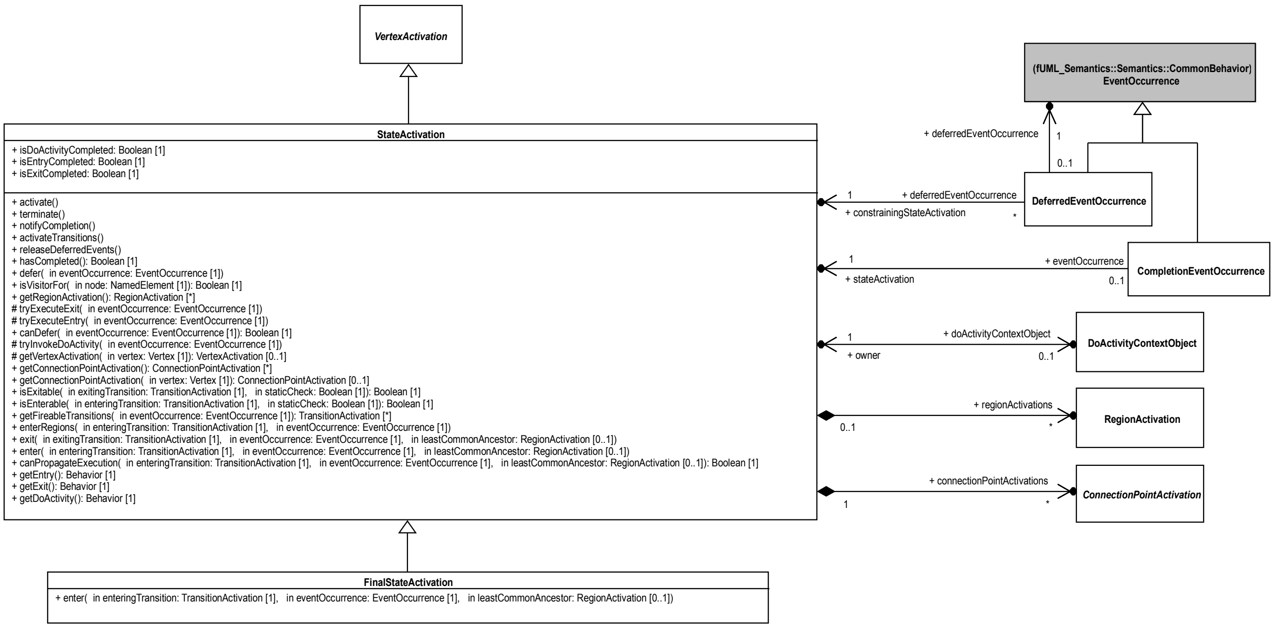

VertexActivation
StateActivation
+ isDoActivityCompleted: Boolean [1]
+ isEntryCompleted: Boolean [1]
+ isExitCompleted: Boolean [1]
+ activate()
+ terminate()
+ notifyCompletion()
+ activateTransitions()
+ releaseDeferredEvents()
+ hasCompleted(): Boolean [1]
+ defer(in eventOccurrence: EventOccurrence [1])
+ isVisitorFor(in node: NamedElement [1]): Boolean [1]
+ getRegionActivation(): RegionActivation [*]
# tryExecuteExit(in eventOccurrence: EventOccurrence [1])
# tryExecuteEntry(in eventOccurrence: EventOccurrence [1])
+ canDefer(in eventOccurrence: EventOccurrence [1]): Boolean [1]
# tryInvokeDoActivity(in eventOccurrence: EventOccurrence [1])
# getVertexActivation(in vertex: Vertex [1]): VertexActivation [0..1]
+ getConnectionPointActivation(): ConnectionPointActivation [*]
+ getConnectionPointActivation(in vertex: Vertex [1]): ConnectionPointActivation [0..1]
+ isExitable(in exitingTransition: TransitionActivation [1], in staticCheck: Boolean [1]): Boolean [1]
+ isEnterable(in enteringTransition: TransitionActivation [1], in staticCheck: Boolean [1]): Boolean [1]
+ getFireableTransitions(in eventOccurrence: EventOccurrence [1]): TransitionActivation [*]
+ enterRegions(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1])
+ exit(in exitingTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
+ enter(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
+ canPropagateExecution(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1]): Boolean [1]
+ getEntry(): Behavior [1]
+ getExit(): Behavior [1]
+ getDoActivity(): Behavior [1]
FinalStateActivation
+ enter(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
(fUML_Semantics::Semantics::CommonBehavior)
EventOccurrence
+ deferredEventOccurrence
1
0..1
DeferredEventOccurrence
1
+ deferredEventOccurrence
+ constrainingStateActivation
*
CompletionEventOccurrence
1
+ eventOccurrence
+ stateActivation
0..1
DoActivityContextObject
1
+ doActivityContextObject
+ owner
0..1
RegionActivation
+ regionActivations
0..1
*
ConnectionPointActivation
+ connectionPointActivations
1
*

Figure 8.8 - StateActivations

StateActivation

A StateActivation is used to execute a State that is either simple or composite, but not a FinalState.

- A StateActivation can have ConnectPointActivations (see 8.5.7), which are SemanticVisitors for EntryPoints and ExitPoints.
- A StateActivation for a composite State owns one or more RegionActivations (see 8.5.3), one for each Region contained in the composite State. A StateActivation for a simple State does not have any RegionActivations.
- A StateActivation for a State with a `doActivity` Behavior will have a DoActivityContextObject (see 8.5.6) to manage the execution of that Behavior.

StateActivation entry

The *common ancestor rule* requires that, before a StateActivation can be entered, all parent VertexActivations of the StateActivation must be entered recursively, until the common ancestor (which is a RegionActivation) of the StateActivation being entered and the source VertexActivation is reached. The entry of a StateActivation then involves the following sequential steps:

1. If the State of the StateActivation has an `entry` Behavior, then this Behavior is executed synchronously.

2. If the State of the StateActivation has a `doActivity` Behavior, then this Behavior is invoked asynchronously. A DoActivityContextObject is created, to act as the context object for the Behavior Execution, and associated with the StateActivation (see also 8.5.6 on `doActivity` Behavior execution).

3. If the State of the StateActivation is composite, then RegionActivations are started concurrently for each Region of the composite State. How each RegionActivation is started depends on whether it is entered explicitly or implicitly (see 8.5.3).

Once a StateActivation is entered, it is then also registered with the StateMachineConfiguration associated with the containing StateMachineExecution and set as the `history` of its RegionActivation (see 8.5.3).

StateActivation exit

Exiting a StateActivation involves the following sequential steps:

1. If the StateActivation owns any RegionsActivations, they are exited.

2. If the StateActivation has a running `doActivity`, it is aborted.

3. If the State of the StateActivation has an `exit` Behavior, this Behavior is executed synchronously.

The common ancestor rule also applies during exit. All parent VertexActivations located at a more nested level than the common ancestor of the StateActivation being exited and the target VertexActivation are also exited.

Once a StateActivation is exited, it is then unregistered from the StateMachineConfiguration associated with the containing StateMachineExecution.

StateActivation completion

The completion of a StateActivation means that a CompletionEventOccurrence is generated by that StateActivation and placed in the `eventPool` handled by the ObjectActivation associated with the `context` Object of the containing StateMachineExecution.

The completion of a StateActivation occurs in the following situations, depending on the structure of the associated State:

- The State is simple and has no associated `entry` or `doActivity` Behaviors. The StateActivation generates a CompletionEventOccurrence as soon as it is entered.
- The State is simple with an associated `entry` Behavior but no `doActivity` Behavior. The StateActivation generates a CompletionEventOccurrence upon the termination of the `entry` Behavior Execution.
- The State is simple and has an associated `doActivity` Behavior. The StateActivation generates a CompletionEventOccurrence only when the `doActivity` Behavior has completed.
- The State is composite and has no associated `doActivity` Behavior. The StateActivation can only generate a CompletionEventOccurrence when all RegionActivations for the Regions of the composite State have completed.
- The State is composite and has an associated `doActivity` Behavior. The StateActivation can only generate a CompletionEventOccurrence when all RegionActivations for Regions of the composite state have completed and the `doActivity` Behavior has completed.

StateActivation and deferred events

A StateActivation can defer an EventOccurrence when the following conditions are met:

1. At least one StateActivation in the active StateMachineConfiguration is for a State with a `deferrableTrigger` that matches the EventOccurrence.

2. There is no Transition with a higher priority and able to react to the EventOccurrence in the active StateMachineConfiguration.

When deferred, an EventOccurrence is "captured" by the deferring StateActivation. This means it is placed into the `deferredEventPool` of the ObjectActivation of the `context` Object of the containing StateMachineExecution (see 8.4) and will only return to the regular `eventPool` when the StateActivation that deferred it leaves the StateMachineConfiguration.

Note. The UML specification states that "A State may specify a set of Event types that may be deferred in that State. This means that Event occurrences of those types will not be dispatched as long as that State remains active. Instead, these Event occurrences remain in the event pool." ([UML], 14.2.3.4.4). However fUML CommonBehavior semantics [fUML] define a dispatching strategy that does not account for deferred events, since these are StateMachine specific. In fUML, once an EventOccurrence is taken from the `eventPool`, it must either be accepted or it is lost. In order to introduce semantics for deferred events, without changing the base fUML CommonBehavior semantic model, instead of leaving deferred EventOccurrences in the `eventPool`, the model defined here moves them to a separate `deferredEventPool` (which is defined on the class SM_ObjectActivation described in 8.4). This solution provides effectively the same semantics as defined in the UML specification, at least for the default first-in-first-out dispatching strategy. However, it may not be compatible with other dispatching strategies, unless they are modified to explicitly account for DeferredEventOccurrences (see also the further discussion of DeferredEventOccurrences in 8.5.9).

Evaluation of a StateActivation

When the static analysis used in the evaluation process of a compound Transition reaches a StateActivation, the analysis proceeds as follows:

1. First, the analysis is propagated to the `parent` of the StateActivation. If the propagation is accepted by the `parent`, the analysis continues.

2. If the StateActivation is for a simple State (i.e., one with no Regions), then the analysis ends. The analysis is considered to have identified an acceptable Transition path, because the StateActivation that has been reached cannot be left in any way other than by the dispatching of an EventOccurrence.

3. If the StateActivation is for a composite State, then the analysis is propagated to the RegionActivations for the Regions owned by the State. In order for the analysis to be acceptable for the composite StateActivation, the analysis of each RegionActivation must find an acceptable Transition path.

FinalStateActivation

A FinalStateActivation specifies the semantics of a FinalState.

As for a regular StateActivation, the common ancestor rule applies when a FinalStateActivation is entered. This means that parent VertexActivations are entered recursively until the common ancestor of the source VertexActivation and the FinalStateActivation is reached.

Once a FinalStateActivation is finally entered, it completes the execution of the RegionActivation in which it is located and clears its `history` (see 8.5.3). If this RegionActivation is owned by a StateActivation, and all RegionActivations owned by that StateActivation have completed, then a CompletionEventOccurrence is generated for that StateActivation (see also the discussion on StateActivation completion above).

Evaluation of a FinalStateActivation

The way the propagation analysis must be performed when a final state is reached is a subset of the propagation sequence described for a State. Before propagating the analysis to the final state, the analysis is propagated to the parent vertex. If the propagation is accepted by the parent vertex then propagation is also accepted by the final state.

8.5.6 "doActivity" Behavior Execution

Figure 8.9 shows part of the PSSM execution model related to the execution of a `doActivity` behavior.

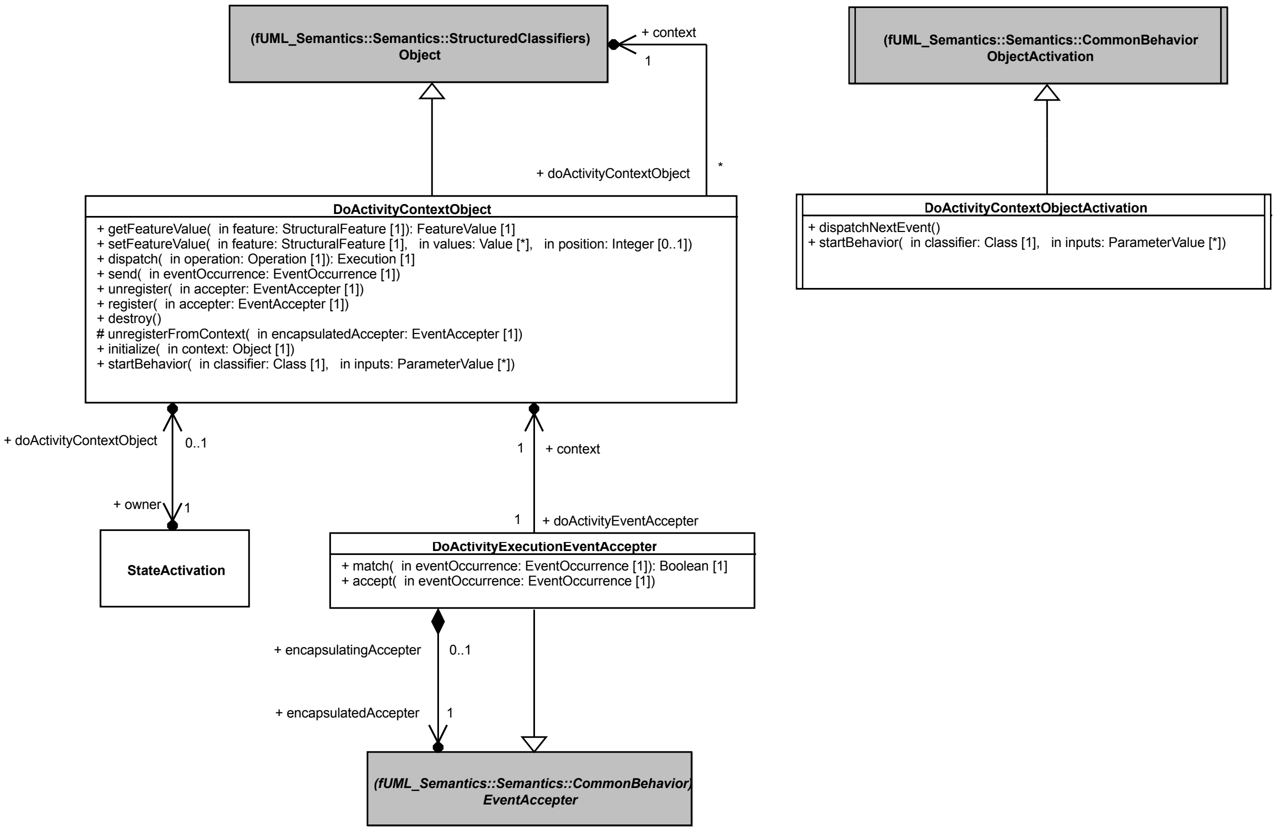

(fUML_Semantics::Semantics::StructuredClassifiers)
Object
+ context
1
*
+ doActivityContextObject
(fUML_Semantics::Semantics::CommonBehavior
ObjectActivation
DoActivityContextObject
+ getFeatureValue(in feature: StructuralFeature [1]): FeatureValue [1]
+ setFeatureValue(in feature: StructuralFeature [1], in values: Value [*], in position: Integer [0..1])
+ dispatch(in operation: Operation [1]): Execution [1]
+ send(in eventOccurrence: EventOccurrence [1])
+ unregister(in accepter: EventAccepter [1])
+ register(in accepter: EventAccepter [1])
+ destroy()
# unregisterFromContext(in encapsulatedAccepter: EventAccepter [1])
+ initialize(in context: Object [1])
+ startBehavior(in classifier: Class [1], in inputs: ParameterValue [*])
DoActivityContextObjectActivation
+ dispatchNextEvent()
+ startBehavior(in classifier: Class [1], in inputs: ParameterValue [*])
+ doActivityContextObject
0..1
+ owner
1
StateActivation
1
+ context
1
+ doActivityEventAccepter
DoActivityExecutionEventAccepter
+ match(in eventOccurrence: EventOccurrence [1]): Boolean [1]
+ accept(in eventOccurrence: EventOccurrence [1])
+ encapsulatingAccepter
0..1
+ encapsulatedAccepter
1
(fUML_Semantics::Semantics::CommonBehavior)
EventAccepter

Figure 8.9 - doActivity Behavior Execution

DoActivityContextObject

Since a `doActivity` Behavior is asynchronous, it is executed on its own thread of execution. The purpose of the DoActivityContextObject is to provide a specialized `context` Object in which the `doActivity` Behavior will be executed. The DoActivityContextObject class is therefore a specialization of the fUML Object class ([fUML], Clause 8).

- The `context` of a DoActivityContextObject is the `context` Object of the StateMachineExecution from which the `doActivity` Behavior was invoked.
- A DoActivityContextObject references the StateActivation that invoked `doActivity` Behavior.

Feature access context

Even though a `doActivity` Behavior is executed on its own thread of execution, it still must be able to access Features (e.g. Properties and Operations) of the context StateMachine from which it was invoked. To allow this, the DoActivityContextObject class redefines operations from the Object to delegate various functions to its own context:

- `getFeature`, for reading a Feature
- `setFeature`, for updating a Feature
- `dispatch`, for calling an Operation
- `send`, for sending an Event

doActivity accepter registration

While a doActivity Behavior is executing, it may need to register EventAccepters for specific EventOccurrences. An accepter registered by a doActivityBehavior is registered in two places:

1. The EventAccepter is registered first as a `waitingEventAccepter` of the SM_ObjectActivation of the StateMachineExecution `context` Object. This is necessary, since EventOccurrences cannot be sent directly to an executing doActivity. Instead, the doActivity Behavior Execution may accept EventOccurrences sent to the `context` Object of its invoking StateMachineExecution and, to be able to do so, it must have its EventAccepters registered with the ObjectActivation of that `context` Object.

2. The EventAccepter is also registered as a `waitingEventAccepter` of the DoActivityContextObjectActivation for the DoActivityContextObject. This is necessary so that, when an EventOccurrence is dispatched from the StateMachineExecution context Object's `eventPool` and accepted by the doActivity Execution, it triggers a run-to-completion step for the doActivity.

doActivity run-to-completion step

A run-to-completion step in an executing `doActivity` Behavior is triggered by the acceptance of an EventOccurrence dispatched from the StateMachineExecution `context` Object's `eventPool`. The acceptance process implies that one of the DoActivityEventAccepter registered by the `doActivity` matched the dispatched EventOccurrence and that the matching accepter has been removed as a waitingEventAccepter for the StateMachineExecution `context` ObjectActivation.

The dispatched EventOccurrence is transferred to the `eventPool` of the DoActivityContextObjectActivation for the DoActivityContextObject for the `doActivity`. Since the originally matching DoActivityEventAccepter is also registered with the DoActivityContextObjectActivation, it will again match and accept the EventOccurrence, but, this time, in the context of the DoActivityContextObject. This starts a new run-to-completion step for the doActivity Execution, asynchronously from the StateMachineExecution.

Note that, in general, an executing `doActivity` Behavior will compete with the executing StateMachine that invoked it to accept EventOccurrences dispatched from the same eventPool. Nevertheless, in some situations it is necessary to ensure that a `doActivity` is able to accept certain EventOccurrences instead of the StateMachine. To allow this, a `deferredTrigger` should be used on the State that owns the `doActivity`, in which case any EventOccurrences deferred while the StateMachine is in that State may be consumed by the executing `doActivity`.

The `doActivity` priority for the consumption of an EventOccurrence is given by the following semantic rules:

- If the StateMachine is about to defer an EventOccurrence for which the `doActivity` has also registered an accepter, the StateMachine is not allowed to defer the EventOccurrence. Instead, the EventOccurrence can then be accepted by the `doActivity`.

- If the StateMachine has deferred an EventOccurrence for which the `doActivity` registers an accepter, then the deferred EventOccurrence can be accepted by the `doActivity` directly from the `deferredEventPool` for the StateMachine.

doActivity finalization

There are two ways for a `doActivity` to finalize its execution:

1. *Completion:* This means the `doActivity` ended its execution naturally (i.e., the execution reached a point where there is no possibility to continue). Completion occurs when, after a run-to-completion step, there are no more event accepter registered for the `doActivity` with its DoActivityContextObjectActivation. When a `doActivity` execution completes, the StateActivation that invoked that `doActivity` may have to complete too. In this situation, upon the completion of the `doActivity` execution, a CompletionEventOccurrence is generated for the StateActivation and placed in StateMachine `context`'s `eventPool`.

2. *Destruction:* This means the StateActivation from which the `doActivity` was invoked is exited. In this situation, the execution of the running `doActivity` is aborted, via a call to the DoActivityContextObject `destroy` operation. In addition to the semantics provided by fUML when an Object is destroyed, all accepters registered by the `doActivity` with the StateMachine `context` ObjectActivation are also destroyed.

DoActivityContextObjectActivation

The DoActivityContextObjectActivation class is a specialized ObjectActivation. Each DoActivityContextObject has a DoActivityContextObjectActivation.

The DoActivityContextObjectActivation class redefines the `dispatchNextEvent` operation provided by the fUML ObjectActivation class. It adds functionality to this operation in order to check if the `doActivity` has completed after the last run-to-completion step.

DoActivityExecutionEventAccepter

A DoActivityExecutionEventAccepter is a specialized EventAccepter.

- A DoActivityEventAccepter references its original creation `context`, a DoActivityContextObject.
- A DoActivityEventAccepter references the original EventAccepter that was registered by an executing `doActivity,` in the DoActivityContextObjectActivation associated with the DoActivityContextObject for the `doActivity` Execution.

DoActivityEventAccepter registration

When an executing doActivity Behavior registers an EventAcceptor with its DoActivityContextObject, the EventAccepter is added to the `waitingEventAccepters` for the associated DoActivityContextObjectActivation. In addition, it is wrapped in a DoActivityEventAccepter, which is then also registered with the StateMachine `context` Object.

DoActivityEventAccepter matching

A DoActivityEventAccepter delegates its check for a matching EventOccurrence to the `match` operation of the wrapped EventAccepter. A DoActivityEventAccepter therefore matches any EventOccurrence that would be matched by its wrapped EventAccepter.

DoActivityEventAccepter acceptance

- When a DoActivityEventAccepter accepts an EventOccurrence, this EventOccurrence is transferred to the `eventPool` of the DoActivityContextObjectActivation of the `context` of the DoActivityEventAccepter. Since the EventAccepter wrapped by the DoActivityEventAccepter will also be registered with this DoActivityContextObjectActivation, this

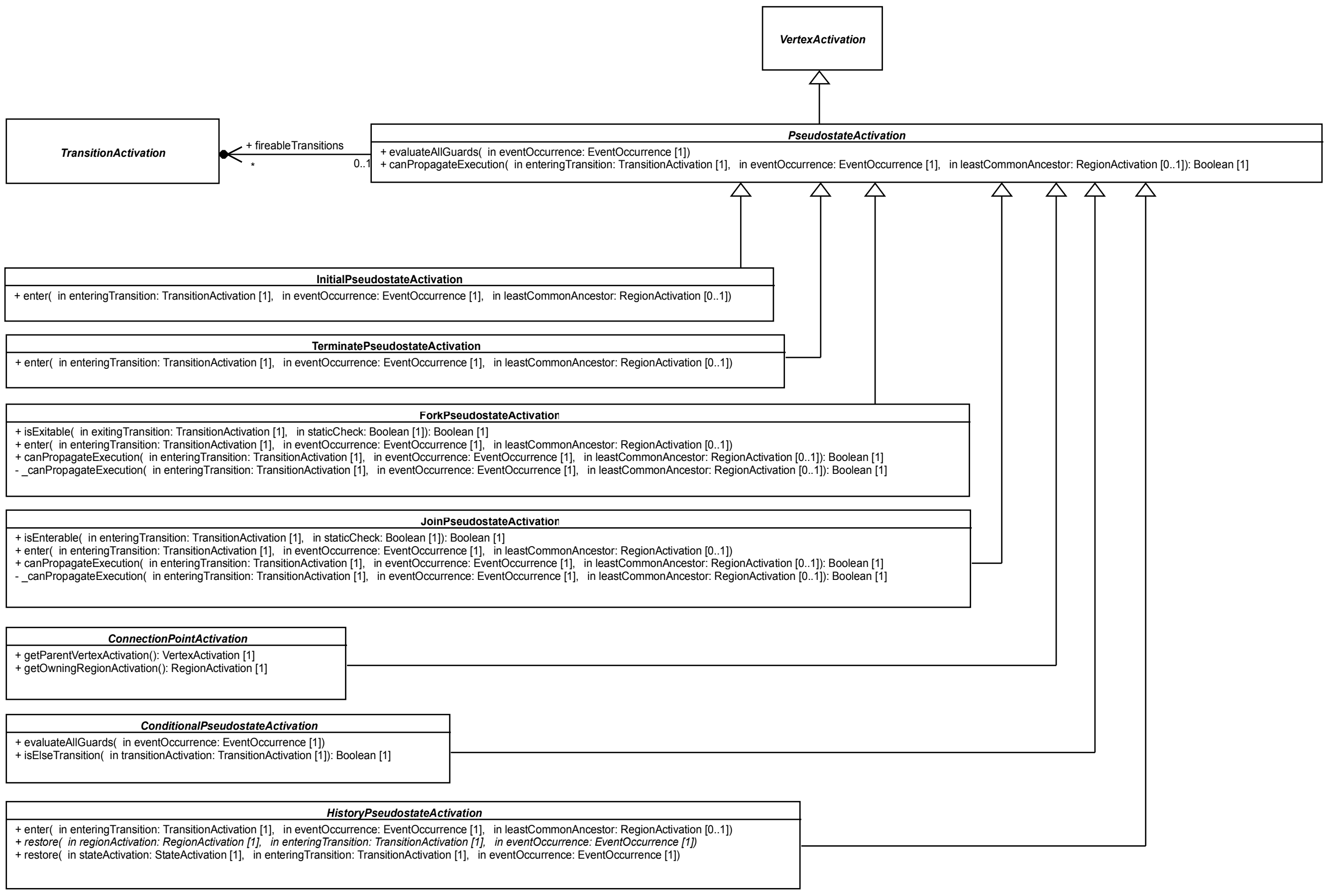
EventAccepter will also match the EventOccurrence, triggering a run-to-completion step in the doActivity Execution without blocking the containing StateMachineExecution.

8.5.7 Pseudostate Activations

8.5.7.1 Basic Pseudostate Activations

PseudostateActivation

The PseudostateActivation class (see Figure 8.10) is a specialization of VertexActivation that specifies the common semantics for Pseudostates. A PseudostateActivation references a set of TransitionActivations corresponding to the set of outgoing Transitions of the Pseudostate whose `guards` have evaluated to true during the static analysis. This set is computed each time the analysis is performed (i.e. each time an evaluation is made as to whether a compound Transition should be added in the set of Transitions to be fired in the next run-to-completion step). PseudostateActivation also redefines the `canPropagateExecution` operation, adding functionality for performing the static analysis in the context of a Pseudostate.



VertexActivation
TransitionActivation
+ fireableTransitions
*
0..1
PseudostateActivation
+ evaluateAllGuards(in eventOccurrence: EventOccurrence [1])
+ canPropagateExecution(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1]): Boolean [1]
InitialPseudostateActivation
+ enter(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
TerminatePseudostateActivation
+ enter(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
ForkPseudostateActivation
+ isExitable(in exitingTransition: TransitionActivation [1], in staticCheck: Boolean [1]): Boolean [1]
+ enter(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
+ canPropagateExecution(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1]): Boolean [1]
- _canPropagateExecution(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1]): Boolean [1]
JoinPseudostateActivation
+ isEnterable(in enteringTransition: TransitionActivation [1], in staticCheck: Boolean [1]): Boolean [1]
+ enter(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
+ canPropagateExecution(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1]): Boolean [1]
- _canPropagateExecution(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1]): Boolean [1]
ConnectionPointActivation
+ getParentVertexActivation(): VertexActivation [1]
+ getOwningRegionActivation(): RegionActivation [1]
ConditionalPseudostateActivation
+ evaluateAllGuards(in eventOccurrence: EventOccurrence [1])
+ isElseTransition(in transitionActivation: TransitionActivation [1]): Boolean [1]
HistoryPseudostateActivation
+ enter(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
+ restore(in regionActivation: RegionActivation [1], in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1])
+ restore(in stateActivation: StateActivation [1], in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1])

Figure 8.10 - PseudostateActivations

StateMachineConfiguration and PseudostateActivation

Although a Pseudostate is a Vertex, PseudostateActivations never enter the StateMachineConfiguration. While Pseudostates are traversed during a run-to-completion step, a run-to-completion step never ends on a Pseudostate.

Evaluation of a PseudostateActivation

The general sequence to propagate the static analysis through a PseudostateActivation is given by the following steps:

1. Propagate the analysis to the `parent` of the PseudostateActivation.

2. If the analysis of the `parent` has an acceptable result and the PseudostateActivation can be entered (i.e., its preconditions to be entered are all fulfilled), then:

    a. If the Pseudostate has no outgoing Transitions, then the analysis is considered to have found an acceptable path.

    b. If it has outgoing Transitions but the set of fireable TransitionActivations remains empty, then no acceptable path can be found through this PseudostateActivation.

    c. If it has outgoing Transitions and the set of fireable TransitionActivations is not empty, then the static analysis of at least one of the TransitionActivations in that set must find an acceptable path. If no such path is found, then there is no acceptable path through this PseudostateActivation.

InitialPseudostateActivation

The InitialPseudostateActivation class (see Figure 8.10) is a specialization of PseudostateActivation that specifies the semantics of a Pseudostate whose `kind` is `initial`.

Entry

The InitialPseudostateActivation class redefines the `enter` operation, such that entrance to an InitialPseudostateActivation results in the firing of its outgoing TransitionActivation.

Note: UML allows an initial Pseudostate to have at most a single outgoing Transition (see [UML], 14.5.6.6). Any other model is ill-formed according to the constraints of the UML specification.

ForkPseudostateActivation

The ForkPseudostateActivation class (see Figure 8.10) is a specialization of PseudostateActivation that specifies the semantics of a Pseudostate whose `kind` is `fork`.

Entry

The ForkPseudostateActivation class redefines the `enter` operation so that the ForkPseudostateActivation is entered by the following sequential steps:

1. Enter the `parent` of the ForkPseudostateActivation, if it has not already been entered. The common ancestor rule applies.

2. Concurrently fire all the outgoing TransitionActivations of the ForkPseudostateActivation. The TransitionActivations are fired without any `guard` evaluation, since UML does not allow Transitions outgoing a fork Pseudostate to have `guards` (see [UML], 14.5.11.8).

Exit

The ForkPseudostateActivation class does not redefine the `exit` operation provided by VertexActivation. Nevertheless it imposes a constraint on when the generic exit sequence can be performed: a ForkPseudostateActivation cannot be exited until all of its outgoing transitions have been fired.

Evaluation

The ForkPseudostateActivation class specifies that a static analysis is propagated by the following steps:

1. Propagate the analysis to the `parent` of the ForkPseudostateActivation. The common ancestor rule applies.

2. If the analysis of the parent has an acceptable result, then an acceptable path can be found through the ForkPseudostateActivation if the static analysis returns acceptable results for all the outgoing TransitionActivations of the ForkPseudostateActivation. If a path fails to be found through any one of the outgoing TransitionActivations, then an acceptable path cannot be found through the ForkPseudostateActivation.

JoinPseudostateActivations

The JoinPseudostateActivation class (see Figure 8.10) is a specialization of PseudostateActivation that specifies the semantics of a Pseudostate whose `kind` is `join`.

Entry

The JoinPseudostateActivation redefines the `enter` operation to check that all TransitionActivations incoming to the JoinPseudostateActivation have been previously fired. If this precondition is satisfied, then the JoinPseudostateActivation is entered by the following steps:

1. Enter the `parent` of the JoinPseudostateActivation, if it has not already been entered. The common ancestor rule applies.

2. Fire one of the TransitionActivations outgoing from the JoinPseudostateActivation. If more than one TransitionActivation is ready to fire, then one is selected nondeterministically (using the ChoiceStrategy mechanism from fUML – see [fUML], 8.2.2.1).

Evaluation

The JoinPseudostateActivation class specifies that a static analysis is propagated by the following steps:

1. Propagate the analysis to the `parent` of the JoinPseudostateActivation. The common ancestor rule applies.

2. If the analysis of the `parent` has an acceptable result, but the JoinPseudostateActivation cannot be entered, then the result of the analysis of the JoinPseudostateActivation is considered to have found an acceptable path ending there.

3. If the analysis of the `parent` has an acceptable result, and the JoinPseudostateActivation can be entered, then the analysis of at least one of the TransitionActivations outgoing from the JoinPseudostateActivation must have an acceptable result. If a path fails to be found through any one of the outgoing TransitionActivations, then an acceptable path cannot be found through the JoinPseudostateActivation.

TerminatePseudostateActivation

The TerminatePseudostateActivation class (see Figure 8.10) is a specialization of PseudostateActivation that specifies the semantics of a Pseudostate whose `kind` is `terminate`.

Entry

The TerminatePseudostateActivation class redefines the `enter` operation so that a TerminatePseudostateActivation is entered by the following steps:

1. Enter the `parent` of the JoinPseudostateActivation, if it has not already been entered. The common ancestor rule applies.

2. Terminate the containing StateMachineExecution. The termination process occurs *without* the execution of `exit` Behaviors of States currently active in the StateMachineConfiguration. It ends with the destruction of the entire StateMachineSemanticVisitors hierarchy.

3. Destroy the `context` Object of the StateMachineExecution. As a result, the ObjectActivation associated with the `context` Object has its `eventPool` cleared and it is stopped. No further execution is possible after this step.

8.5.7.2 Connection Point Activations

ConnectionPointActivation

The ConnectionPointActivation class (see Figure 8.11) is a specialization of PseudostateActivation that specifies the common semantics for entry-point and exit-point Pseudostates. These common semantics define how to determine the parent VertexActivation and the owning RegionActivation of an EntryPointPseudostateActivation or an ExitPointPseudostateActivation.

- The `parent` of a ConnectionPointActivation is the StateActivation on which this ConnectionPointActivation is placed.
- The RegionActivation which is said to own the ConnectionPointActivation is the `parent` RegionActivation of the StateActivation on which the ConnectionPointActivation is placed.

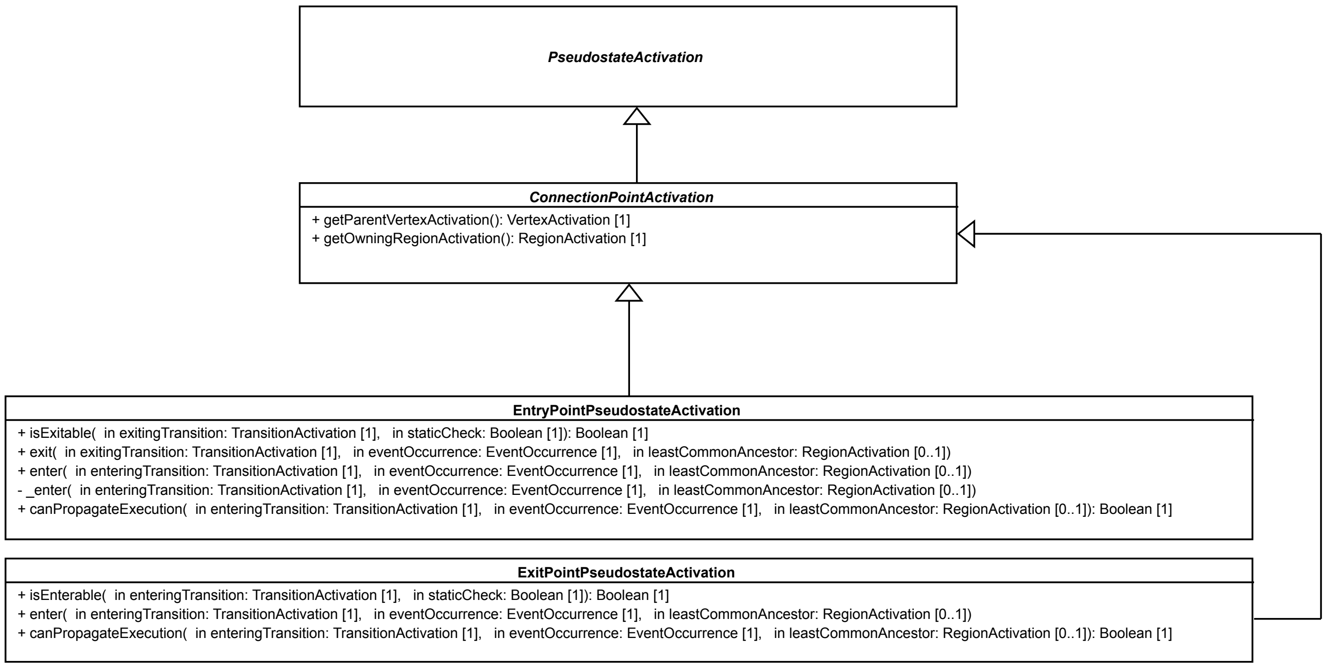

PseudostateActivation
ConnectionPointActivation
+ getParentVertexActivation(): VertexActivation [1]
+ getOwningRegionActivation(): RegionActivation [1]
EntryPointPseudostateActivation
+ isExitable(in exitingTransition: TransitionActivation [1], in staticCheck: Boolean [1]): Boolean [1]
+ exit(in exitingTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
+ enter(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
- _enter(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
+ canPropagateExecution(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1]): Boolean [1]
ExitPointPseudostateActivation
+ isEnterable(in enteringTransition: TransitionActivation [1], in staticCheck: Boolean [1]): Boolean [1]
+ enter(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
+ canPropagateExecution(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1]): Boolean [1]

Figure 8.11 - EntryPointActivation and ExitPointActivation

EntryPointPseudostateActivation

The EntryPointPseudostateActivation class is a specialization of ConnectionPointActivation that specifies the semantics of a Pseudostate whose `kind` is `entryPoint`.

Entry

The EntryPointPseudostateActivation class (see Figure 8.11) redefines the `enter` operation so that an EntryPointPseudostateActivation is entered by the following steps:

1. Enter the `parent` StateActivation for the EntryPointPseudostateActivation. The common ancestor rule applies (i.e., the `parent` of that StateActivation may also need to be entered).

2. If the EntryPointPseudostateActivation has no outgoing TransitionActivations, then the parent StateActivation performs a default entry (see StateActivation in 8.5.5).

3. If the EntryPointPseudostateActivation has outgoing TransitionActivations, then one of two situation can occur:

    a. If the State on which the Pseudostate is placed is not orthogonal, then one of the outgoing TransitionActivations that is fireable is chosen to be fired. If more than one outgoing TransitionActivation is fireable, then one is chosen nondeterministically (using the fUML ChoiceStrategy mechanism – see [fUML]. 8.2.2.1).

    b. If the State on which the Pseudostate is place is orthogonal (i.e, it has multiple Regions), then all TransitionActivations outgoing from the EntryPointPseudostateActivation are fired concurrently.

Exit

The EntryPointPseudostateActivation class specifies the precondition that an EntryPointPseudostateActivation can only be exited after all its outgoing TransitionActivations have fired. This precondition only applies if the entry point is on an orthogonal State.

Evaluation

The EntryPseudostateActivation class specifies that a static analysis is propagated by the following steps:

1. Propagate the analysis to the `parent` StateActivation of the EntryPseudostateActivation.

2. If the analysis of the parent has an acceptable result and the EntryPointPseudostateActivation has no outgoing TransitionActivations, then the analysis is considered to have found an acceptable path ending at the EntryPointPseudostateActivation.

3. If the analysis of the parent has an acceptable result and the EntryPointPseudostateActivation has outgoing TransitionActivations, then one of two situations can occur:

    a. If the State being entered is not orthogonal, then only one of the analyses of the outgoing TransitionActivations must have an acceptable result in order for there to be an acceptable path through the EntryPointPseudostateActivation.

    b. If the State being entered is not orthogonal, then the analyses of the outgoing TransitionActivations must all have acceptable results in order for there to be an acceptable path through the EntryPointPseudostateActivation.

ExitPointPseudostateActivation

The ExitPseudostateActivation class (see Figure 8.11) is a specialization of ConnectionPointActivation that specifies the semantics of a Pseudostate whose `kind` is `exitPoint`.

Enter

The ExitPointPseudostateActivation class redefines the `enter` operation so that an ExitPointPseudostateActivation is entered by the following steps:

1. Nondeterministically select one of the fireableTransitions for the ExitPointPseudostateActivation (using the ChoiceStrategy mechanism from fUML – see [fUML], 8.2.2.1).

2. Exit only the `parent` StateActivation of the ExitPointPseudostateActivation.

3. Fire the selected TransitionActivation.

An ExitPointPseudostateActivation can only be entered if all of its incoming TransitionActivations have been fired.

Evaluation

The ExitPointPseudostateActivation class specifies that a static analysis is propagated by the following steps:

1. If the ExitPointPseudostateActivation cannot be entered, then the analysis is considered to have an acceptable result.

2. If the ExitPointPseudostateActivation can be entered, then at least one of the analyses of the outgoing TransitionActivations must have an acceptable result in order for the analysis of the ExitPseudostateActivation to have an acceptable result.

8.5.7.3 Conditional Pseudostate Activations

ConditionalPseudostateActivation

The ConditionalPseudostateActivation class (see Figure 8.12) is a specialization of a PseudostateActivation that specifies the semantics common to choice and junction Pseudostates.

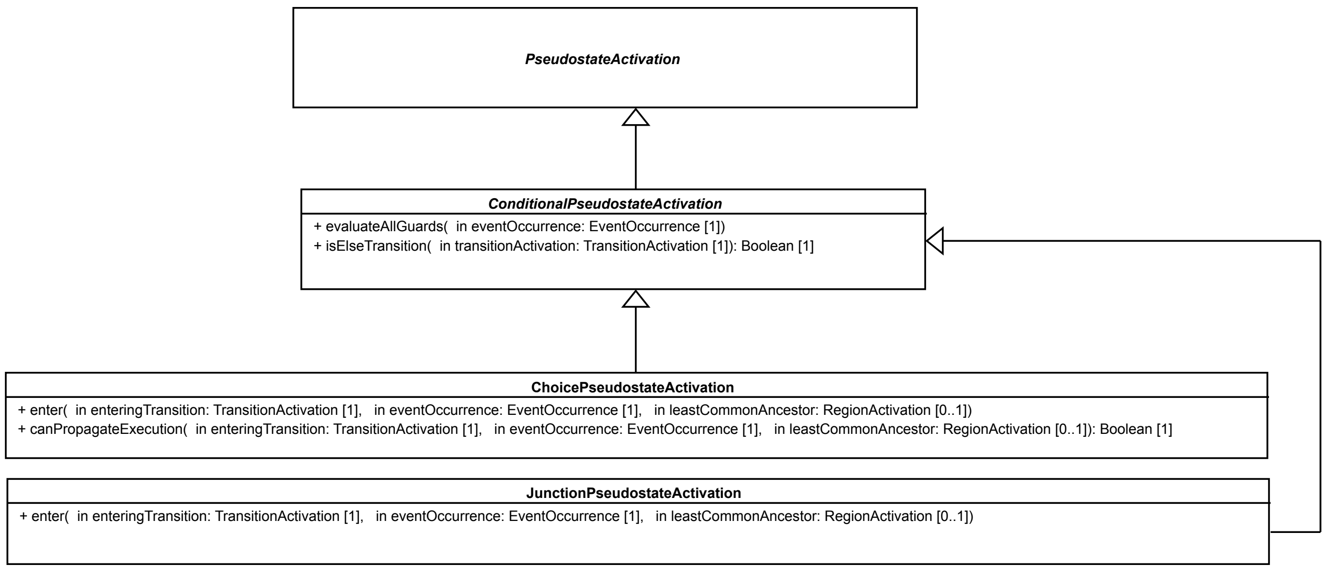

PseudostateActivation
ConditionalPseudostateActivation
+ evaluateAllGuards(in eventOccurrence: EventOccurrence [1])
+ isElseTransition(in transitionActivation: TransitionActivation [1]): Boolean [1]
ChoicePseudostateActivation
+ enter(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
+ canPropagateExecution(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1]): Boolean [1]
JunctionPseudostateActivation
+ enter(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])

Figure 8.12 - ChoicePseudostateActivation and JunctionPseudostateActivation

ConditionalPseudostateActivation redefines the `evaluateAllGuards` operation from the PseudostateActivation so that all `guards` of outgoing Transitions are evaluated. TransitionActivations for Transitions whose `guard` evaluates to true are added to the set of `fireableTransitions`. If this produces no `fireableTransitions`, but there is is an outgoing "else" Transition, then the TransitionActivation for this Transition is added to the set of `fireableTransitions`. An "else" Transition is one with a `guard` Constraint whose `specification` is an Expression whose `symbol` is the string "else" and which has no `operands` (see also 7.3.1).

ChoicePseudostateActivation

The ChoicePseudostateActivation class is a specialization of ConditionalPseudostateActivation that specifies the semantics of a Pseudostate whose `kind` is `choice`.

Entry

The ChoicePointPseudostateActivation class (see Figure 8.12) redefines the `enter` operation so that a ChoicePointPseudostateActivation is entered by the following steps:

1. Enter the `parent` of the ChoicePointPseudostateActivation, if it has not already been entered. The common ancestor rule applies.

2. Evaluate all `guards` of Transitions outgoing the choice Pseudostate. Note that it is specific to choice Pseudostates that the guards of outgoing Transitions are only evaluated when the Pseudostate is reached during the course of a run-to-completion step. This is known as *dynamic evaluation,* as opposed to the *static evaluation* performed during static analysis.

3. Nondeterministically select one TransitionActivation from the set of (dynamically) fireable TransitionActivations (using the fUML ChoiceStrategy mechanism – see [fUML], 8.2.2.1).

Evaluation

The ChoicePseudostateActivation class specifies that the static analysis of the ChoicePseudostateActivation has an acceptable result if the analysis of the `parent` of the ChoicePseudostateActivation does. The static analysis is not propagated to outgoing TransitionActivations, since the `guards` of Transitions outgoing from a choice Pseudostate are dynamically evaluated.

JunctionPseudostateActivation

The JunctionPseudostateActivation class (see Figure 8.12) is a specialization of ConditionalPseudostateActivation that specifies the semantics of a Pseudostate whose `kind` is `junction`.

Entry

The JunctionPointPseudostateActivation class redefines the `enter` operation so that a JunctionPseudostateActivation is entered by the following steps:

1. Enter the `parent` of the JunctionPointPseudostateActivation, if it has not already been entered. The common ancestor rule applies.

2. Nondeterministically select one TransitionActivation from the set of `fireableTransitions` (using the fUML ChoiceStrategy mechanism – see [fUML], 8.2.2.1).

8.5.7.4 History Pseudostate Activations

HistoryPseudostateActivation

The HistoryPseudostateActivation class (see Figure 8.13) is a specialization of PseudostateActivation that specifies the common semantics of ShallowHistoryPseudostateActivation and DeepHistoryPseudostateActivation.

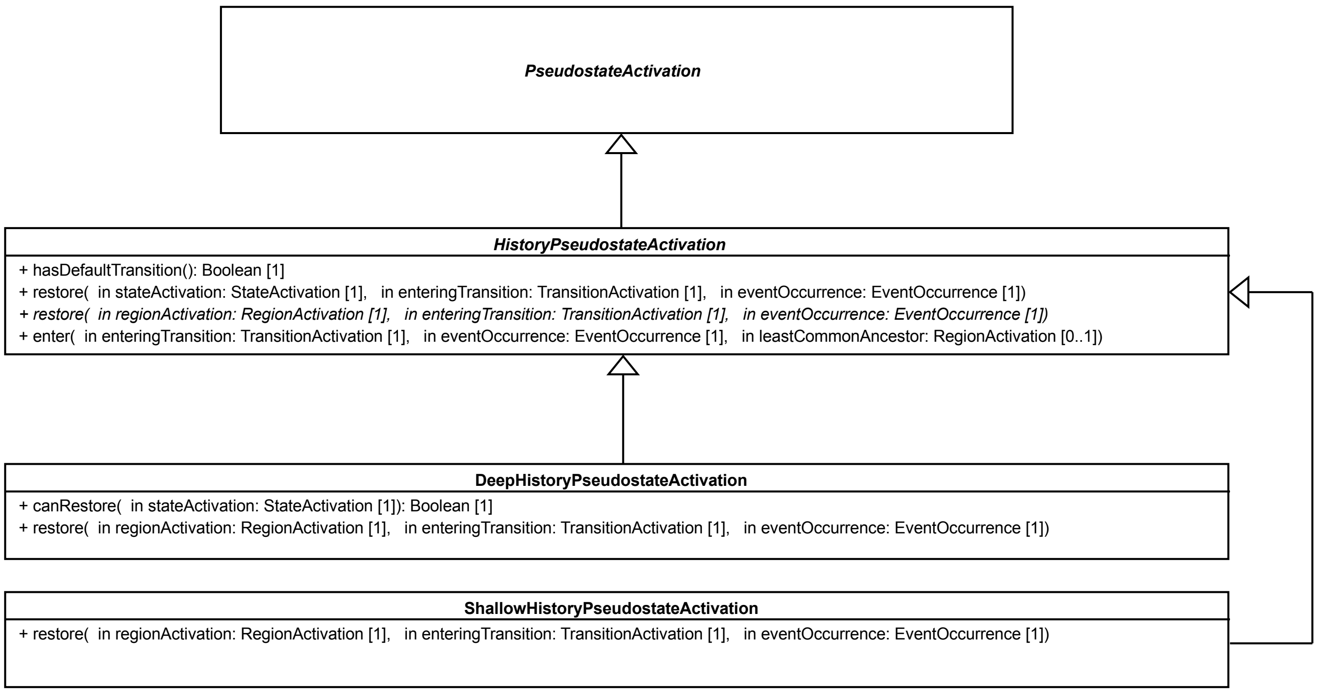

PseudostateActivation
HistoryPseudostateActivation
+ hasDefaultTransition(): Boolean [1]
+ restore(in stateActivation: StateActivation [1], in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1])
+ restore(in regionActivation: RegionActivation [1], in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1])
+ enter(in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1], in leastCommonAncestor: RegionActivation [0..1])
DeepHistoryPseudostateActivation
+ canRestore(in stateActivation: StateActivation [1]): Boolean [1]
+ restore(in regionActivation: RegionActivation [1], in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1])
ShallowHistoryPseudostateActivation
+ restore(in regionActivation: RegionActivation [1], in enteringTransition: TransitionActivation [1], in eventOccurrence: EventOccurrence [1])

Figure 8.13 - DeepHistoryPseudostateActivation and ShallowHistoryPseudostateActivation

Entry

The HistoryPseudostateActivation class redefines the `enter` operation so that a HistoryPseudostateActivation (deep or shallow) is entered as follows:

- If the `parent` RegionActivation of the HistoryPseudostateActivation has no `history`, and the history Pseudostate has no default Transition (i.e. an outgoing Transition that targets a Vertex directly or indirectly owned by the Region that owns the history Pseudostate), then
  - If the history Pseudostate is nested in a State hierarchy, then this is entered. The common ancestor rule applies.
  - If the history Pseudostate is owned by a top-level Region (i.e. a Region owned by a StateMachine), then this Region performs an implicit entry.
- If the `parent` RegionActivation of the HistoryPseudostateActivation has a `history`, the history Pseudostate has a default Transition, then
  - If the history Pseudostate is nested in a State hierarchy, then this is entered, and the restoration process starts from the StateActivation owning the parent RegionActivation of the HistoryPseudostateActivation.

- ○ If the history Pseudostate is owned by a top-level Region, then the restoration process starts from the RegionActivation for that Region.

Restoration

HistoryPseudostateActivation provides two kinds of restoration process (see `restore` operations in Figure 8.13), one for restoration of a StateActivation and one for restoration starting of a RegionActivation. Deep and shallow history have common semantics for restoring a StateActivation, but the semantics for restoring a RegionActivation is specific to each kind of history.

The restoration of a StateActivation consists of the following steps:

1. The StateActivation is entered into the StateMachineConfiguration.
2. The entry and doActivity behaviors that are associated with the State (if any) are executed. If, after this, the StateActivation is completed, a CompletionEventOccurrence is placed in the StateMachine `context`'s event pool.
3. If the StateActivation has RegionActivations, then all of them are restored concurrently.

DeepHistoryPseudostateActivation

The DeepHistoryPseudostateActivation class (see Figure 8.13) is a specialization of HistoryPseudostateActivation that specifies the semantics of a Pseudostate whose `kind` is `deepHistory`.

Restoration

The DeepHistoryPseudostateActivation class specifies that a RegionActivation is restored by the following steps:

1. If the RegionActivation being restored is the `parent` RegionActivation of the DeepHistoryPseudostateActivation, then
    a. If the RegionActivation has a `history` (which is a StateActivation), then this `history` is restored using the generic restoration process specified by the HistoryPseudostateActivation class for a StateActivation.
    b. If the RegionActivation has no `history`, but the history Pseudostate has a default Transition, then the TransitionActivation for this Transition is fired.
2. If the RegionActivation is not the `parent` RegionActivation of the DeepHistoryPseudostateActivation, then
    a. If the RegionActivation is within the `parent` RegionActivation of the DeepHistoryPseudostateActivation, then it is restored.
    b. Otherwise the RegionActivation is not restored but, instead, performs an implicit entry.

ShallowHistoryPseudostateActivation

The ShallowHistoryPseudostateActivation class (see Figure 8.13) is a specialization of HistoryPseudostateActivation that specifies the semantics of a Pseudostate whose `kind` is `shallowHistory`.

Restoration

The ShallowHistoryPseudostateActivation class specifies that a RegionActivation is restored in a manner that is slightly different from that specified for a DeepHistoryPseudostateActivation. The parent RegionActivation of the ShallowHistoryPseudostateActivation is the only one that is restored. All other RegionActivations (for orthogonal Regions or nested Regions) perform implicit entries.

8.5.8 Transition Activations

TransitionActivation

Figure 8.14 shows the three specializations of the TransitionActivation class, which, respectively, specify the semantics for external Transitions, local Transitions and internal Transitions. The semantics of the different kinds of Transition are reflected in the way the `sourceVertexActivation` is exited and the `targetVertexActivation` is entered. The different kinds of Transactions all share the base semantics for `guard` evaluation and evaluation of the reactivity to a particular EventOccurrence.

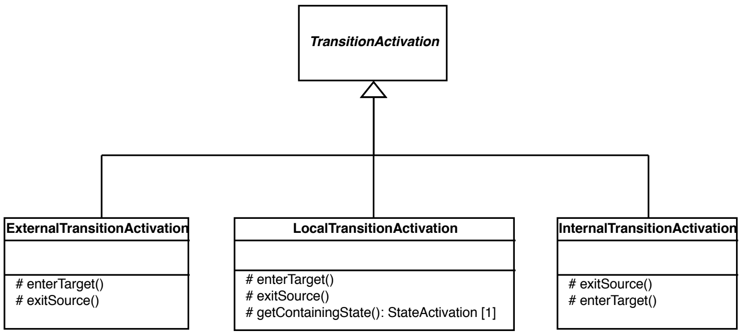

TransitionActivation
ExternalTransitionActivation
# enterTarget()
# exitSource()
LocalTransitionActivation
# enterTarget()
# exitSource()
# getContainingState(): StateActivation [1]
InternalTransitionActivation
# exitSource()
# enterTarget()

Figure 8.14 - TransitionActivations

ExternalTransitionActivation

Exit source

In the case of an ExternalTransitionActivation, the `sourceVertexActivation` is exited only if all of its prerequisites to be exited are fulfilled (e.g., a ForkPseudostateActivation can only be exited when all its outgoing TransitionActivations except this one have been traversed), otherwise it is not exited.

The way the `sourceVertexActivation` is exited also depends on whether the `targetVertexActivation` can be entered. If the `targetVertexActivation` is not ready to be entered, then the exit sequence of the `sourceVertexActivation` is limited to itself. Otherwise, if the `targetVertexActivation` is ready to be entered, then the exit sequence of the `sourceVertexActivation` follows the common ancestor rule. This implies that the exit sequence is propagated to parent VertexActivations until the common ancestor between the `sourceVertexActivation` and the `targetVertexActivation` is reached.

Enter target

If the prerequisites to enter the `targetVertexActivation` are fulfilled, then this VertexActivation is entered, following the common ancestor rule. This means that the entering sequence is propagated to parent VertexActivations until the common ancestor existing between the `sourceVertexActivation` and the `targetVertexActivation` is reached.

If the prerequisites are not fulfilled (e.g., the target is a StateActivation that is not already active), the `targetVertexActivation` is not entered. Nevertheless, if the target is a StateActivation for a composite State, then the RegionActivation owning the `sourceVertexActivation` completes. This may lead to the generation of a CompletionEventOccurrence for the StateActivation composite State (see 8.5.5 on situations in which a StateActivation is ready to complete).

LocalTransitionActivation

Containing StateActivation

For a LocalTransitionActivation, the exiting of the `sourceVertexActivation` and the entering of the `targetVertexActivation` are conditioned by the identification of the so-called *containing StateActivation*. The containing State of a local Transition can be determined in the following manner:

1. If the `sourceVertexActivation` of the local Transition is an EntryPointActivation, then the containing StateActivation is the owner of this EntryPointActivation (i.e., a StateActivation for a composite State).
2. If the `sourceVertexActivation` contains the `targetVertexActivation`, then the containing StateActivation is the `sourceVertexActivation`.
3. Otherwise the containing StateActivation is the `targetVertexActivation`.

Exit source

If the `sourceVertexActivation` has fulfilled its requirements to be exited, two cases are possible:

1. If the `sourceVertexActivation` is an EntryPointActivation, the exit sequence is trivial. Only the EntryPointActivation is exited, through one or more continuation Transitions.
2. If the `sourceVertexActivation` is a StateActivation for a composite State and the `targetVertexActivation` is a Vertex Activation located in a RegionActivation owned by the `sourceVertexActivation`, then the `sourceVertexActivation` cannot be exited since it is also the containing StateActivation for the LocalTransitionActivation. If there is already a StateActivation that is active in the same RegionActivation as the `targetVertexActivation`, then that StateActivation is exited.

Enter target

If the `targetVertexActivation` has fulfilled its requirement to be entered and it is not the containing StateActivation of the LocalTransitionActivation, then the entering sequence starts and the common ancestor rule applies.

InternalTransitionActivation

Exit source

An InternalTransitionActivation never exits its `sourceVertexActivation`.

Enter target

An InternalTransitionActivation never enters its `targetVertexActivation`.

8.5.9 Event Occurrences

Three kinds of event occurrences can be accepted by a StateMachineEventAccepter: SignalEventOccurrence, CallEventOccurrence and CompletionEventOccurrence (see Figure 8.15). In addition, the DeferredEventOccurrence class is used to wrapped deferred EventOccurrences. All these classes are specializations of the base EventOccurrence class, which is part of the fUML common model for handling events (see [fUML, 8.4.3]), as is the SignalEventOccurrence class. The other classes are added for PSSM and are described further below.

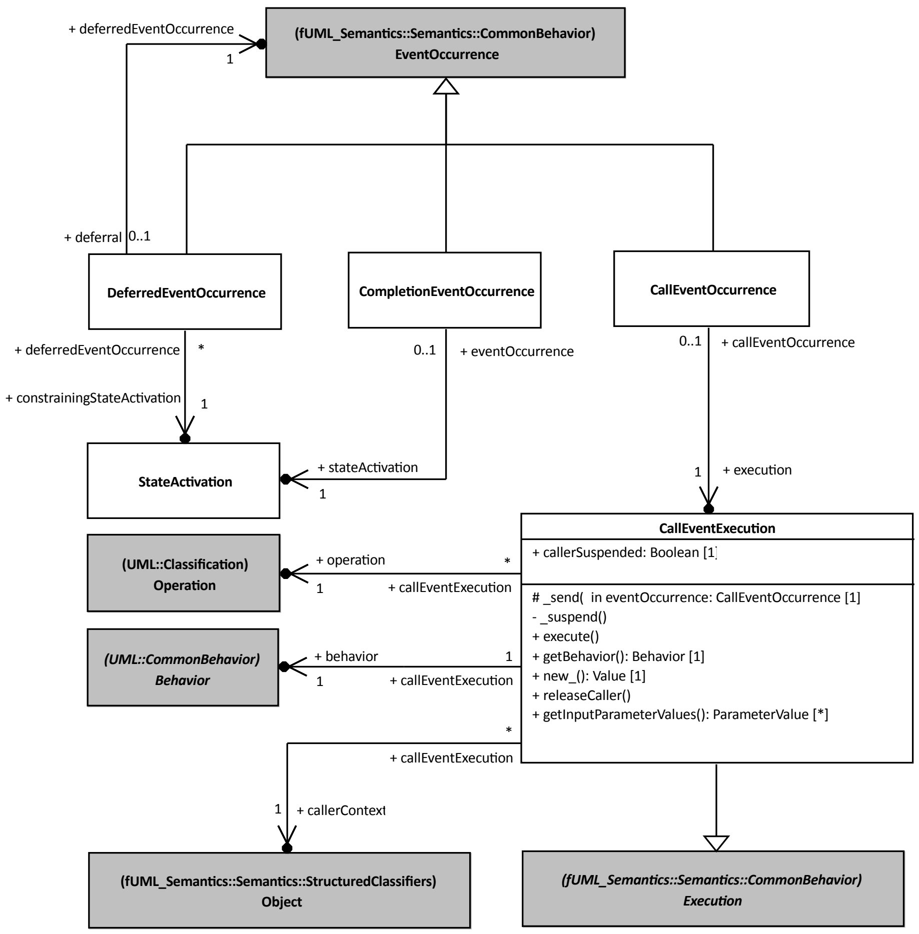


Figure 8.15 – Event Occurrences

CompletionEventOccurrence

A CompletionEventOccurrence is a specialization of EventOccurrence that denotes the completion of a StateActivation.

Scope of completion events

The scope of a CompletionEvent is limited to the StateActivation from which it was generated. This means that, when the CompletionEventOccurrence is dispatched and accepted, it can only be used to trigger a completion TransitionActivation (i.e, a TransitionActivation for a completion Transition, which has no explicit trigger – see [UML], 14.2.3.8.3) originating from the StateActivation that generated the CompletionEventOccurrence.

If the StateActivation that generated the CompletionEvent has no completion TransitionActivation, then the CompletionEventOccurrence will be lost once it is dispatched.

Priority of completion events

When generated, a CompletionEventOccurrence is placed in the `eventPool` of the ObjectActivation associated with the StateMachineExecution `context` Object. CompletionEventOccurrences added to the `eventPool` have priority over all other

EventOccurrences except other CompletionEventOccurrences. This means that a new CompletionEventOccurrence is placed into the (ordered) `eventPool` behind any CompletionEventOccurrences already in the pool, but ahead of any other EventOccurrences.

CallEventOccurrence

A CallEventOccurrence is a specialization of EventOccurrence that denotes a call to an Operation. This kind of EventOccurrence is always produced by a CallEventExecution.

CallEventExecution

The fUML semantics for calling an Operation are specified using the `dispatch` operation of the Object class, which returns an Execution for the appropriate `method` Behavior used to implement the Operation, taking any polymorphic redefinition of the Operation into account (see [fUML], 8.3.2.1). In fUML, it is an error if no `method` can be found for the Operation being called. In PSSM, however, a call to an Operation with no `method` is handled using a CallEventOccurrence.

Dispatching behavior is actually a semantic variation point in fUML, with the exact behavior provided by a DispatchStrategy class. The default DispatchStrategy class is RedefinitionBasedDispatchStrategy, which is specialized in PSCS by the CS_DispatchOperationOfInterfaceStrategy. This is further specialized in PSSM by the SM_RedefinitionBasedDispatchStrategy, whose dispatch operation creates a CallEventExecution in the case that a called Operation has no implementing `method` (see 8.3).

A CallEventExecution is a specialization of the fUML Execution class whose `execute` operation is specified to create a CallEventOccurrence. Normally, an Execution is instantiated from a Behavior, which serves as its `type`. This is not the case for a CallEventExecution, however, which, instead, creates an effective Behavior with the same Parameter signature as the called Operation. The CallEventExecution class then overrides the Execute `getBehavior` operation to return this effective Behavior.

After a CallEventOccurrence is created, it is placed into the `eventPool` of the target Object of the Operation call, from which it may be dispatched and, potentially, trigger a run-to-completion step in the target Object. However, as in fUML, PSSM semantics only provide for synchronous Operation calls, so the caller remains blocked on its calling action until the call is completed. The callerSuspended flag of the associated CallEventExecution remains true while the caller is suspended.

If the CallEventOccurrence is dispatched and it triggers a run-to-completion step, then, once the step completes (i.e., at the end of the `accept` operation of the StateMachineEventAccepter), the `releaseCaller` operation of the CallEventExecution for the CallEventOccurrence is called, which notifies the `callerContext` (the Object from which the call was made) to let it continue its own execution. It is also possible that the CallEventOccurrence is never handled (for example, if it is dispatched but cannot be accepted at that time by the StateMachineExecution), in which case the call will never return and the execution of the caller will simply hang.

Note. CallEvents in UML are not specific to StateMachines, but are part of the UML CommonBehavior model (see [UML], 13.3). However, the fUML subset does not currently include CallEvent, only allowing calls to Operations for which an implementing `method` can be found. Nevertheless, because it is a common use of StateMachines to specify the behavior operations via CallEvent triggers on Transitions, this capability is included in PSSM.

DeferredEventOccurrence

A DeferredEventOccurrence is a specialization of EventOccurrence used to wrap another EventOccurrence (the actual `deferredEventOccurrence`) that has been deferred. An EventOccurrence is always deferred by a StateActivation (see 8.5.5), which becomes the constrainingStateActivation of the DeferredEventOccurrence in which it is wrapped

An EventOccurrence is deferred under the following conditions:

1. The current StateMachineConfiguration includes a StateActivation for a State that declares a `deferrableTrigger` that matches a dispatched EventOccurrence.

2. The analysis of the StateMachineConfiguration did not find a TransitionActivation with a higher priority that could be fired by the dispatched EventOccurrence.

3. There is no “overriding” Transition (i.e. a Transition outgoing from the State declaring the `deferrableTrigger`) able to fire with the dispatched EventOccurrence.

If these conditions hold, the EventOccurrence is wrapped in a DeferredEventOccurrence and placed in the deferredEventPool of the SM_ObjectContextActivation of the context of the StateMachineExecution (see also 8.4). A DeferredEventOccurrence is returned to the regular `eventPool` when the StateActivation responsible for deferring the EventOccurrence is no longer in the StateMachineConfiguration.

8.5.10 Event Data Passing

8.5.10.1 Event Triggered Execution

A run-to-completion step is always started by the acceptance of an EventOccurrence. Then, during the run-to-completion step, a number of Behaviors may be executed. For a StateMachine, such Behaviors include `effect` Behaviors on Transitions and `entry`, `exit` and `doActivity` Behaviors on States. In addition, a `guard` Condition on a Transition may have a specification that is an OpaqueExpression that may be defined using a Behavior. These Behaviors are all considered to have *event-triggered executions* within the run-to-completion step for a given EventOccurrence.

Any of the kinds of Behaviors mentioned above can have input Parameters by which they can receive data contained in the dispatched EventOccurrence during an event-triggered execution. In addition, `effect`, `entry` and `exit` Behaviors can also have output Parameters that are used to provide data to be returned from a synchronous Operation call being handled via a CallEventOccurrence. (See 7.6.2 on the necessary syntactic constraints on the Parameters of such Behaviors.)

EventTriggeredExecution

The EventTriggeredExecution (see Figure 8.16) class is a specialization of the fUML Execution class that specifies the semantics of a `wrappedExecution` happening within the context of the run-to-completion step of a `triggeringEventOccurrence`. The wrappedExecution is the normal kind of Execution corresponding to the actual Behavior being executed (e.g., an ActivityExecution for an Activity). (See also [fUML], 8.4.2, on the Execution class and its ParameterValue mechanism.)

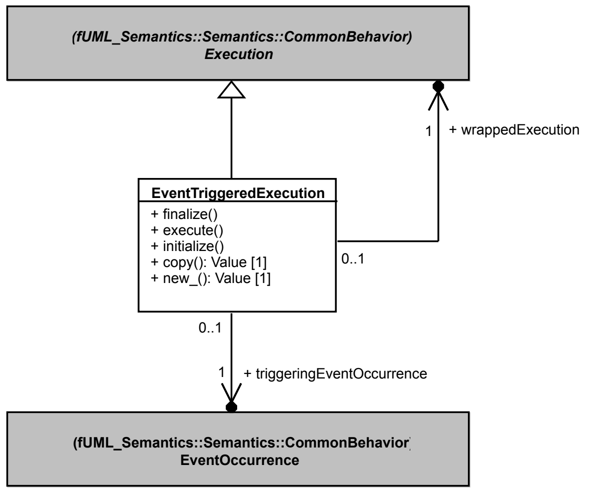


Figure 8.16 - EventTriggeredExecution

Execution

The EventTriggeredExecution class defines the `execute` operation from the Execution class to do the following:

1. If the Behavior being executed has appropriate input Parameters (see below), then extract the data contained in the `triggeringEventOccurrence` and pass it to the `wrappedExecution` as ParameterValues.
2. Execute the `wrappedExecution`.
3. If the Behavior being executed has output parameters, extract the output ParameterValues (see below).

Input ParameterValues

The initialize operation of the EventTriggeredExecution class is used to extract data from an EventOccurrence and create the corresponding ParameterValues to be passed to a `wrappedExecution`. Syntactic constraints ensure that, if the Behavior being executed has Parameters, then they are appropriate to receive the data from any possible `triggeringEventOccurrence`. (See 7.6.2.2, `pssm_state_behavior_parameters` and `pssm_transition_behavior_parameters` constraints.)

Data can be extracted from either a SignalEventOccurrence or a CallEventOccurrence.

1. If the `triggeringEventOccurrence` is a SignalEventOccurrence, then the executing Behavior must have either one Parameter or no Parameters. If the Behavior has a Parameter, the SignalInstance corresponding to the SignalEventOccurrence is passed to the `wrappedExecution` as the value of that Parameter.
2. If the `triggeringEventOccurrence` is a CallEventOccurrence, then the executing Behavior will either have no Parameters or its input ("in" or "inout") Parameters will conform, in order, to the input Parameters of the Operation of the CallEvent for the CallEventOccurrence. If the Behavior has Parameters, then the input ParameterValues of the CallEventExecution for the CallEventOccurrence (see 8.5.9) are used to set the input ParameterValues of the `wrappedExecution`.

Output ParameterValues

Output ParameterValues may only be produced when the `triggeringEventOccurrence` is a CallEventOccurrence and the Operation being called has output ("out", "inout" and "return") Parameters. In that case, an `effect`, `entry` or `exit` Behavior, in addition to having input parameters that conform to those of the called Operation, can also have output Parameters conforming to the output Parameters of the Operation (see 7.6.2.2, `pssm_state_behavior_parameters` and `pssm_transition_behavior_parameters` constraints). In such a situation, after the completion of the `wrappedExecution`, the output ParameterValues it produces are used to set the outputParameterValues of the CallEventExecution of the CallEventOccurrence (see 8.5.9).

Note. In presence of concurrency, the output ParameterValues provided to the CallEventExecution may have changed multiple times, if multiple Behaviors producing outputs are executed during the course of a run-to-completion step. The final output ParametersValues are the ones provided by the last executed Behavior. Since the order of concurrent execution is nondeterministic, which are the final outputs may also be nondeterministic. If nondeterminism is not desired, then it is a modeler responsibility to ensure that a CallEventOccurrence will never result in the concurrent execution of multiple Behaviors producing output values.

8.5.10.2 Event Data Passing and Static Analysis

While `effect` Behaviors Transitions and `entry`, `exit` and `doActivity` Behaviors on States are only executed during the realization of a run-to-completion step, `guard` evaluation (except for `guards` on the outgoing Transitions of a choice Pseudostate – see 8.5.7.3) takes place during the static analysis of the validity of the compound Transitions that might be added to the set of Transitions to be fired. If a `guard` Constraint has a `specification` that is an OpaqueExpression defined by an associated Behavior, then that Behavior may have input Parameters in order to obtain EventOccurrence data (as described above). For any such `guards` evaluated during the static analysis process, data is extracted from the EventOccurrence that has been dispatched from the `eventPool` and is being matched, even though a run-to-completion step has not actually started yet.

8.6 Actions

The PSSM abstract syntax subset includes all the Actions included in the fUML abstract syntax subset, and PSSM does not specify any different semantics for those Actions than specified in the fUML execution model (as extended in some cases by PSCS). Nevertheless, it is necessary for the PSSM execution model to define a specialization of the PSCS ReadSelfActionActivation class, in order to simply preserve the semantics of ReadSelfActions under PSSM. No other ActionActivation classes are specialized in PSSM.

As described in 8.5.6, the model for the semantics of the execution of `doActivities` includes the use of a special DoActivityContextObject class for the `context` of the Execution of a `doActivity`, wrapping the `context` Object of the StateMachineExecution that invoked the `doActivity`. If the `doActivity` is an Activity, then, in most cases, the fact that the `context` Object for the Activity is a DoActivityContextObject is entirely transparent to the Actions in the Activity, which execute as if the `context` was the same as that of the containing StateMachine.

However, if a ReadSelfAction is executed within a `doActivity`, its base semantics would be to return a reference to the actual `context` Object of the containing Activity, which would be a DoActivityContextObject. Because the DoActivityContextObject class overrides various operations of the Object class (as shown on Figure 8.9), it acts indistinguishably from the StateMachine context Object that it wraps when accessing any behavioral or structural Feature. Nevertheless, a DoActivityContextObject still has a different identity than the actual `context` Object it wraps, and it would be distinguishable from the wrapped `context` using, say, a TestIdentityAction. Further, reading Links in which the StateMachine `context` Object participates would not work using the DoActivityContextObject, because the Links being read are identified by implicitly testing the identity of the Objects at their ends.

To prevent these problems, the PSSM execution model includes the SM_ReadSelfActionActivation class, which is a specialization of the PSCS CS_ReadSelfActionActivation (see Figure 8.17). The specialized class overrides the `getExecutionContext` operation to test whether the `context` Object of the containing Behavior Execution is a DoActivityContextObject. If so, it returns the context Object wrapped by the DoActivityContextObject, rather than the DoActivityContextObject. As a result, when a ReadSelfAction is executed within a `doActivity`, its output is the StateMachine `context` Object, just as it is when a ReadSelfAction executes within any other kind of Behavior owned by a StateMachine.

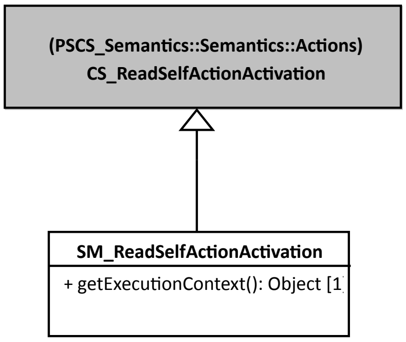

(PSCS_Semantics::Semantics::Actions)
CS_ReadSelfActionActivation
SM_ReadSelfActionActivation
+ getExecutionContext(): Object [1

Figure 8.17 - Actions Extension

8.7 Loci

The Loci package in the PSSM execution model includes specializations of the CS_Locus and CS_ExecutionFactory classes from the Loci package of the PSCS execution model. The PSCS classes are specialized, rather than the corresponding fUML execution model classes, so that the PSSM execution model can also handle the SemanticVisitor classes that provide the operational semantics for PSCS, which is necessary to execute a model at the "Joint PSSM and PSCS Conformance" level (see 2.2.2). However, a model at the "PSSM-only Conformance" level, that strictly adheres to the PSSM subset specified in Clause 7, will not include any PSCS-specific elements and, therefore, can be executed without the PSCS functionality inherited by the PSSM Loci classes.

The SM_Locus class redefines the instantiate operation such that, if the given type is not a Behavior, then it is instantiated as an SM_Object (see 8.3), rather than a regular fUML Object. The SM_ExecutionFactory class redefines the instantiateVisitor operation in order to instantiate the new SemanticVisitors for StateMachine elements (as defined in 8.5) and to instantiate the PSSM-specific SM_OpaqueExpressionEvaluation SemanticVisitor for OpaqueExpression (as defined in 8.2) and the SM_ReadSelfActionActivation SemanticVisitor for ReadSelfAction (as defined in 8.6).

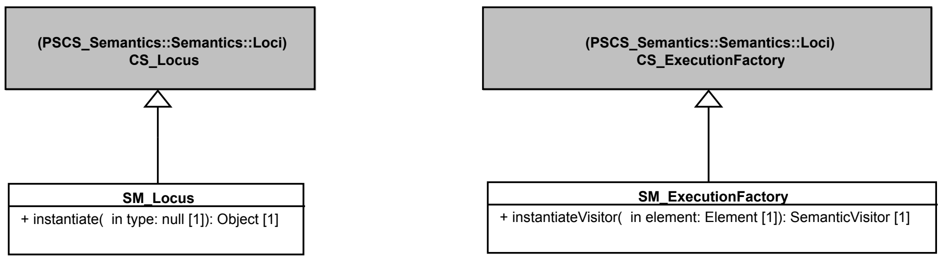

(PSCS_Semantics::Semantics::Loci)
CS_Locus
(PSCS_Semantics::Semantics::Loci)
CS_ExecutionFactory
SM_Locus
+ instantiate(in type: null [1]): Object [1]
SM_ExecutionFactory
+ instantiateVisitor(in element: Element [1]): SemanticVisitor [1]

Figure 8.18 - Loci Extensions

9 Test Suite

9.1 Overview

This clause presents a test suite to be used to validate that an execution tool conforms to the semantic model presented in Clause 8 (see also Clause 2 on the requirements for conformance). The test suite is an fUML, PSCS and PSSM conformant model comprising a set of test cases that, when executed by an execution tool, report on whether or not the expected results are obtained.

The definition of the test suite is based on an analysis of the UML specification of the semantics of state machines ([UML], Clause 14) that identified a set of requirements to be validated by the test cases in the suite. Each requirement is a textual statement about one specific part of the semantics of state machines. Each test case then verifies whether or not an execution tool meets one particular requirement, as formally interpreted according to the semantic model defined in Clause 8.

The test suite is separated into two parts.

1. The first part defines the abstract architecture of a test case. This architecture is specialized (in the UML sense) for each test case. A detailed presentation of this part of the test suite model is given in 9.2.
2. The second part of the test suite 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 test cases related to transition semantics. Each test case in this category asserts a specific part (identified in the requirements) of the transition semantics. All test cases in the test suite are described in 9.3. Each description includes a statement of the requirement covered by the test case, a model of the state machine being tested and a description of the expected result of running the test.

The purpose of having a strong coupling between the semantic requirements for state machines is to be able to (as in any software development) identify quickly and precisely what is covered by the semantic model in terms of semantics and what is not. Coverage of the requirements by the test suite is discussed in 9.4.

9.2 Utilities

9.2.1 Overview

One objective of the PSSM test suite is to define a base architecture to simplify the definition of executable tests cases. This architecture (structure and behavior) is presented in 9.2.2. The communications that take place between the different elements of the architecture are presented in 9.2.3. Finally, 9.2.4 explains the process of generating a trace that captures information about the state machine execution. This trace is used to compare the execution expected for the state machine against the trace actually generated at execution time.

9.2.2 Architecture

9.2.2.1 Architecture Concepts

This subclause presents the architecture that was defined to describe test cases to assess the PSSM semantic model. The base architecture of the PSSM test suite is inspired by concepts identified by the UML testing profile. The UML testing profile was built to provide “a standardized language based on OMG's Unified Modeling Language for designing, visualizing, specifying, analyzing, constructing, and documenting the artifacts commonly used in and required for various testing approaches, in particular model-based testing (MBT) approaches” [UTP]. The UTP concepts used in the PSSM test suite are:

- *TestComponent:* “Test components are part of the test environment and are used to communicate with the system under test (SUT) and other test components. The main function of test components is to drive a test case by stimulating the system under test through its provided interfaces and to evaluate whether the actual responses of the system under test comply with the expected ones.” (see subclause 8.2.2.2 of [UTP] to read the complete description of this concept).
- *TestCase:* “A test case is a behavioral feature or behavior specifying tests. A test case specifies how a set of test components interact with an SUT to realize a test objective. Test cases are owned by test contexts, and therefore have access to all parts of the test configuration, other global variables (e.g., data pools, etc.) or further behavioral features (e.g., auxiliary methods). A test case always returns a verdict.” (see subclause 9.2.2.4 of [UTP] to read the complete description of this concept).
- *TestContext:* “A test context acts as a grouping mechanism for a set of test cases. The composite structure of a test context is referred to as test configuration. The classifier behavior of a test context may be used for test control” (see subclause 8.2.2.3 of [UTP] to read the complete description of this concept).

Figure 9.1 shows how these concepts are used in the context of the definition of the abstract architecture of a test. Figure 9.2 shows the structure of the semantic test container for such tests. These classes and their behavior, are described in 9.2.2.2.

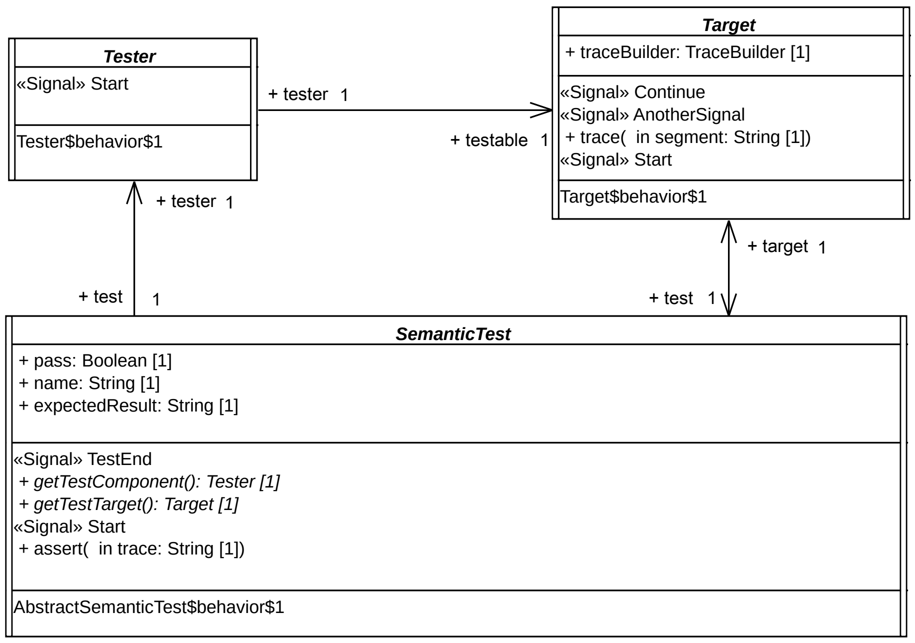


Figure 9.1 - Architecture of an Abstract Semantic Test

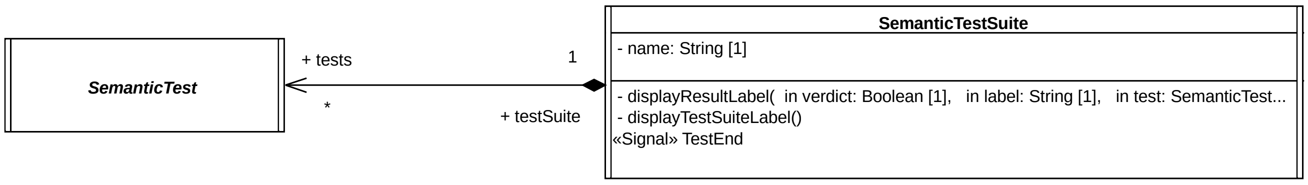

SemanticTest
+ tests
*
1
+ testSuite
SemanticTestSuite
- name: String [1]
- displayResultLabel(in verdict: Boolean [1], in label: String [1], in test: SemanticTest...
- displayTestSuiteLabel()
«Signal» TestEnd

Figure 9.2 - SemanticTest and SemanticsTestSuite

9.2.2.2 Architecture Class Descriptions

9.2.2.2.1 Tester

Description

Tester is an abstract active class which encodes in its classifier behavior the stimulation sequence (i.e, a set of event occurrences) that will be sent to the target (i.e., the system under test).

Note that this role matches what is intended for a TestComponent in UTP. This class has the stereotype “TestComponent” applied.

Association Ends

- testable: Target [1] – The SUT (System Under Test) to which the stimulation sequence is sent.
- test: SemanticTest [1] – The test case which controls the tester.

Receptions

- Start – A tester can receive a Start signal

Classifier Behavior

The classifier behavior of the abstract Tester class is empty. Specializations are intended to provide a new classifier behavior which will encode the user defined stimulation sequence.

9.2.2.2.2 Target

Description

Target defines the system under test. Specializations of this class have to provide their classifier behaviors specified as a state machine.

A target receives the stimulation sequence produced by the ester. The dispatching of the events will enable transitions of the state machine playing the role of a classifier behavior to be triggered.

Throughout its execution the state machine generates an execution trace. This trace is stored by the target and finally provided as the result of the execution to the test which controls the target.

Attributes

- traceBuilder: TraceBuilder [1] – Each test target owns a trace builder. It enables the classifier behavior of a target to build a trace of its execution.

Association Ends

- test: SemanticTest [1] - The test case which controls the target.

Operations

- trace(in segment: String[1]) – The operation enables the addition of "segment" (i.e., a new part) to the execution trace. It can be called at any time in the classifier behavior of the target to capture information relative to the executed state machine.

Receptions

- Start – The target is able to receive Start signals
- Continue – The target is able to receive Continue signals
- AnotherSignal – The target is able to receive AnotherSignal signals.
- Pending – The target is able to receive Pending signals.
- Data – The target is able to receive Data signals. Data signal has a property value which is of type Boolean.
- IntegerData – The target is able to receive IntegerData signals. IntegerData signal has a property value which is of type Integer.

Classifier Behavior

The classifier behavior of the abstract Target is empty. All specializations are intended to provide a new classifier behavior which will be the state machine whose execution is performed by the execution model defined for PSSM.

9.2.2.2.3 SemanticTest

Description

The SemanticTest class represents the main artifact of a semantic test case. A semantic test is in charge of instantiating and controlling the tester and the target (i.e. the SUT). When the execution of the SUT is completed, the execution trace that was produced is provided to the semantic test case for analysis. If the trace matches one of the expected traces for the executed state machine the test is deemed to have passed otherwise it is marked as failed.

The classifier behavior of a semantic test has the TestCase stereotype applied.

Attributes

- name: String [1] – The name of the test case.
- pass: Boolean [1] – The current status (pass or fail) of the test.
- expectedResult: String [1..*] – The set of all possible valid execution traces for the SUT.

Operations

- getTestComponent(): Tester – An abstract operation which returns an instance of Tester controlled by the semantic test whose classifier has been started. This operation is intended to be redefined by specializations of SemanticTest.
- getTestTarget(): Target – An abstract operation which returns an instance of the target controlled by the semantic test whose classifier has been started. This operation is intended to be redefined by specializations of SemanticTest.

- assert(in trace: String[1]) – This operation updates the value of the attribute “pass” by comparing the trace given as a parameter to the expected execution trace known by the semantic test.
- register(in possibleTrace: String[1]) – This operation updates the set of valid traces that can be expected to be generated by the target. The specified trace is added to the set of traces.

Receptions

- Start – The semantic test is able to receive Start signals.
- TestEnd – The semantic test is able to receive TestEnd signals.

Classifier Behavior

The classifier behavior of a semantic test is defined as a UML activity. It conforms to the fUML subset. The Alf specification corresponding to this activity is presented in Table 9.1.

The flow of this behavior is the following:

1. When the classifier behavior starts, it blocks waiting for the reception of a Start signal
2. Upon receiving the Start signal, it creates and initializes both the tester and the target.
    - The links (instances of associations) are created.
    - The tester and the target are each sent a Start signal.
    - The semantic test then blocks waiting for a TestEnd signal.
3. When the TestEnd signal is received, it computes the verdict of the test. The verdict is either PASS or FAIL. In case of a failure of the test, the semantic test displays the trace generated by the target as well as the set of valid traces that were expected. Finally, regardless of the verdict, the semantic test notifies the test suite that controls it about the termination of the test execution.

Note that specializations of SemanticTest are not intended to override this general pattern.

```
private import Util::Protocol::Messages::Start;
private import Util::Protocol::Messages::End;
activity 'AbstractSemanticTest$behavior$1'() {
     // Wait for a start signal
     accept(Start);
     // Tester and target are created and started
     target = this.getTestTarget();
     tester = this.getTestComponent();
     // Create link connection between the two
     target.tester = tester;
     tester.testable = target;
     this.target = target;
     this.tester = tester;
     // Both are started
     target.Start();
     tester.Start();
     // Wait for the arrival of the result emitted by the target
     accept(executionResult : End);
     this.pass = this.matches(executionResult.trace);
     // Result analysis
     if(!this.pass){
          WriteLine("[TEST] "+ this.name + " ** FAIL ** - {"+
               executionResult.trace + "} does not match any of the following");
          for(i in 1..this.expectedTraces->size()){
               WriteLine("-----> [" + IntegerFunctions::ToString(i)+"] - "+
                    this.expectedTraces->at(i));
          }
     }else{
          WriteLine("[TEST] " + this.name + " - PASS -");
     }
     // Provides the result back to the test suite governing this semantic test case
     this.testSuite.TestEnd(this.pass);
}
```

Table 9.1: Specification of SemanticTest Behavior

9.2.2.2.4 SemanticTestSuite

Description

An instance of SemanticTestSuite owns a set of SemanticTests. The execution of these tests is orchestrated by the test suite itself. Tests are executed one by one. At the end of each test the verdict is retrieved by the test suite that is in charge of displaying the results. This is an active class. It is not intended to be specialized.

It corresponds to the concept of TestContext proposed by UTP and, consequently, it is stereotyped as a TestContext.

Attributes

- name: String [1] – The name of the test suite.

Association Ends

- tests: SemanticTest [*] - the set of semantic tests that are handled by the semantic test suite.

Receptions

- TestEnd – The SemanticTestSuite class is able to received TestEnd signals.

Operations

- displayResultLabel(in verdict: Boolean [1], in label: String[1], in test: SemanticTest [1]) – A utility operation to display the test result on an output stream.
- displayTestSuiteLabel() - A utility operation to display the name of the suite on an output stream.

Classifier Behavior

The test suite is composed of a set of semantic tests. Each semantic test is executed in sequence. The execution of a semantic test is started by the sending of a Start signal and ends when the semantic test suite receives the notification TestEnd, which includes the information relative to the semantic test verdict. The entire semantic test suite is deemed to have failed if at least one of its test fails. In case of failure, the semantic test suite provides the number of failed semantic tests. The classifier behavior of the semantic test suite is presented in Table 9.2 using the Alf notation.

```
private import Util::Protocol::Messages::TestEnd;
private import Util::Protocol::Messages::Start;

activity 'SemanticTestSuite$behavior$1'() {
     accept(Start);
     WriteLine("\n[TEST SUITE ("+IntegerFunctions::ToString(this.tests->size())+
          " tests)] - "+this.name+"\n");
     // Execute all semantic tests registered in that test suite
     Integer failures = 0;
     for(i in 1..this.tests->size()){
          // Test gets the authorization to start
          this.tests->at(i).Start();
          // Test suite waits for the results
          accept(testResult : TestEnd){
               if(testResult.verdict == false){
                    failures++;
               }
          }
     }
     // There is at least one failure then the test suite is considered as failed
     if(failures > 0){
          WriteLine("\n[TEST SUITE] - "+this.name + " FAILURE ("+
               IntegerFunctions::ToString(failures+
               " / "+IntegerFunctions::ToString(this.tests->size())+" failed)\n");
     }
}
```

Table 9.2: Specification of SemanticTestSuite Behavior

9.2.3 Protocol

9.2.3.1 Protocol Overview

The Protocol package of the test suite has two subpackages: Messages and Events.

1. Messages contains all signals used to communicate between the different active classes:

```
namespace StateMachine_TestSuite::Util::Protocol;
package Messages {
   /* -- Synchronization --*/
   public signal Start {}
   public signal End {
        public trace: String;
   }
   public signal TestEnd {
        public verdict: Boolean;
        public label: String[0..1];
   }
   /* -- Synchronization --*/
   /* -- Stimulations --*/
   public signal Continue {}
   public signal AnotherSignal {}
   public signal Pending {}
   public signal Data{
        public value: Boolean;
   }
   public signal IntegerData{
        public value: Integer;
   }
   /* -- Stimulations --*/
}
```

2. Events contains the signal events (for the signals located in Messages) that can be directly used for triggers.

The synchronization signals in the Messages package (i.e., Start, End and TestEnd) are used to synchronize executions of different active objects. These signals are further described in 9.2.3.2.

The stimulation signals (i.e., Continue, Pending, AnotherSignal, Data and IntegerData), on the other hand, are used by the tester to stimulate the target (i.e. the system under test). None of these signals contain data except Data and IntegerData. These two signals are used to assess event data passing semantics.

9.2.3.2 Synchronization Signal Descriptions

9.2.3.2.1 Start

Description

The Start signal is used for to two purposes in the test suite context. First, it enables the test suite to start the execution of a specific semantic test. Second it enables the test to start both its tester and its target. The modeling constraint for the SemanticTest, the Tester, and the Target is that they must all register as acceptors for the Start signal at the beginning of the execution of their classifier behaviors. Note that the Start signal does not include any data (it has no attributes).

9.2.3.2.2 End

Description

The End signal enables the Target to provide its controller (i.e., the SemanticTest) with a notification containing its execution trace. The semantic test takes advantage of this notification to compute the test verdict.

Attributes

- trace: String [1] – The execution trace generated by the state machine that plays the role of a classifier behavior for the Target.

9.2.3.2.3 TestEnd

Description

The TestEnd signal enables a semantic test to notify its test suite that it has completed. This notification encapsulates two items of information: the test verdict as well as, in case of failure, a label indicating the differences between the expected trace and the trace actually produced during the execution.

Attributes

- verdict: Boolean [1] – The verdict of test that is two say pass or fail encoded as Boolean values.
- label: String [0..1] – If the test failed, the label presents the difference between the trace that was expected and the one actually produced during the execution.

9.2.4 Tracing

At run time, the target is intended to produce an execution trace. This trace will be used to compute the test verdict by comparing the trace expected by the semantic test against the one actually generated by the target. The production of this trace relies on a small utility class, TraceBuilder.

```
namespace  StateMachine_TestSuite::Util::Tracing;
class TraceBuilder {
     /*Record the trace as simple String*/
     public trace: String;
     /*Construction and destruction */
     @Create
     public TraceBuilder();
     @Destroy
     public destroy();
     /*Add a new segment in the trace*/
     public addSegment(in segment: String);
}
```

The execution information (state entered, behavior executed, etc.) that must be part of the trace is up to the designer of the test. To add new information in the trace, the designer must call the trace operation provided by the target. The latter will delegate to the trace builder. Such a call to the trace operation must take place while the state machine is running. Consequently, there are four places at which the calls to trace might occur:

1. The entry behavior of a state
2. The doActivity behavior of a state
3. The exit behavior of a state
4. The effect behavior of a transition

9.3 Tests

9.3.1 Overview

This subclause presents the different test cases that are currently included in the PSSM test suite. The tests cases are grouped into categories. Each category is related to a specific part of the semantics.

Each test included in the test suite specializes the base test architecture presented in 9.2.2.1. Figure 9.3 illustrates such a specialization, for the case of the test *Behavior 001* (which is fully described in 9.3.2.2). The *Target* class presented in 9.2.2.2.2 is specialized in order to provide a specific test target. This specialized class then defines a new classifier behavior, which is the state machine that is going to be executed.

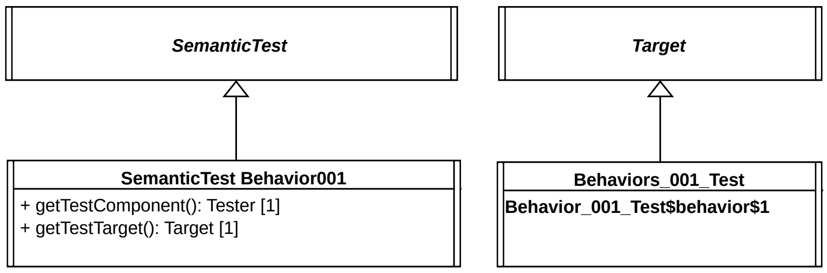

SemanticTest
Target
SemanticTest Behavior001
+ getTestComponent(): Tester [1]
+ getTestTarget(): Target [1]
Behaviors_001_Test
Behavior_001_Test$behavior$1

Figure 9.3 - Behavior 001 Test Architecture

The general class *SemanticTest* is also specialized. This enables the test to provide redefined versions of operations *getTestComponent* and *getTestTarget*. These two operations are used to instantiate and start the classifier behaviors of both the tester (see 9.2.2.2.1) and the test target (see 9.2.2.2.2).

All test cases in the test suite follow a similar test architecture, except for those in Error: Reference source not found, which test the execution of "standalone" state machines. In the standalone state machine test cases, the state machine is itself the test target, rather than being the classifier behavior of another class. However, these test cases otherwise run in a similar fashion to the other test cases.

Each of the following test descriptions includes:

- A statement of the semantic requirement covered by the test.
- A diagram of the state machine being tested.
- The event sequence that is received by the tested state machine. The order in which the event occurrences are enumerated is the order in which the event occurrences will be received. Each received event occurrence is related to a specific state machine configuration.
- The execution trace generated during the test execution. This trace does not represent the complete execution of the tested state machine. It is only composed of trace messages generated while corresponding model elements composing the state machine are executed. Although the trace built during the execution is not complete, it is always sufficient to evaluate if the state machine was executed in way that conforms to the semantics specified for UML state machines.
- An explanatory note detailing the different phases of the execution.

- A table describing the different run-to-completion (RTC) steps realized during the execution of the tested state machine. This table contains the following columns:
    1. *Steps* – The identifier of the run-to-completion step
    2. *Event pool* – The status of the event pool for the time at which the RTC step is realized. Using the default first-in-first-out dispatching strategy, the rightmost event occurrence in the pool is then the one to be dispatched for that step. In the list of event occurrences, the notation "CE" is used to identify the occurrence of a completion event.
    3. *State machine configuration* – The state configuration of the state-machine at the moment when the RTC step is realized.
    4. *Fired transition(s)* – The transitions that are fire during the RTC step.
- A list of all alternative execution traces that can be be produced by the test. If there are alternative traces, one of the traces is chosen and explained using a table describing the different RTC steps leading to the generation of that trace.

9.3.2 Behavior

9.3.2.1 Overview

Tests presented in this subclause test that semantics associated with state behaviors (i.e., entry, doActivity and exit) conform to what is specified in UML.

9.3.2.2 Test Behavior 001

Tested state machine

The state machine that plays the role of a classifier behavior for the class *Behaviors_001_Test* is presented in Figure 9.4. The entry behavior associated with the state *S1* of this state machine is intended to be executed when the state is entered. When executed, the behavior will add in the execution trace a message *S1(entry).* If the message is not part of the trace, then the test is considered to have failed.

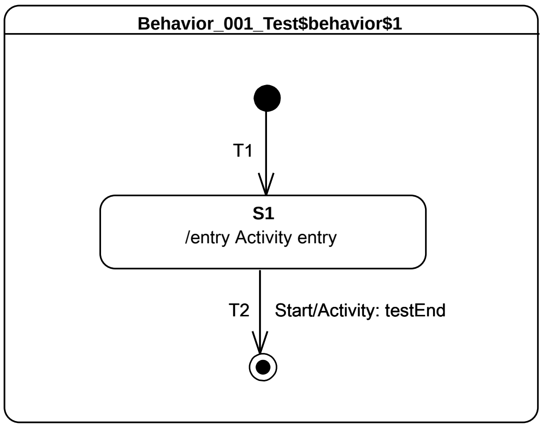


Figure 9.4 - Behavior 001 Test Classifier Behavior

The behavior which is associated as an entry to the state S1 is presented as an activity in Figure 9.5.

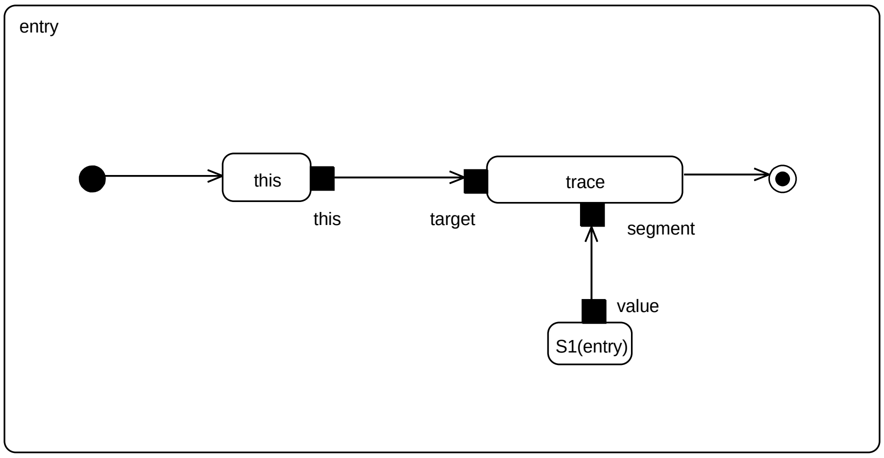


Figure 9.5 - S1 entry behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *S1*

Generated trace

- S1(entry)

Note. The purpose of this test is to assess that *S1* entry behavior is executed when the state is entered.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(S1)**]	[S1]	[]
3	[**Start]**	[S1]	[T2]

9.3.2.3 Test Behavior 002

Tested state machine

The state machine that is executed for this test is presented in Figure 9.6.

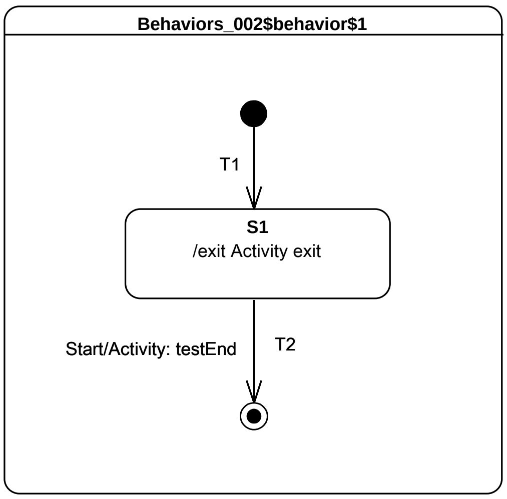


Figure 9.6 - Behavior 002 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *S1*

Generated trace

- S1(exit)

Note. The purpose of this test is to assert that *S1* exit behavior is executed when the state is exited.

RTC steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(S1)**]	[S1]	[]
3	[**Start**]	[S1]	[T2]

9.3.2.4 Test Behavior 003-A

Tested state machine

The state machine that is executed for this test is presented in Figure 9.7.

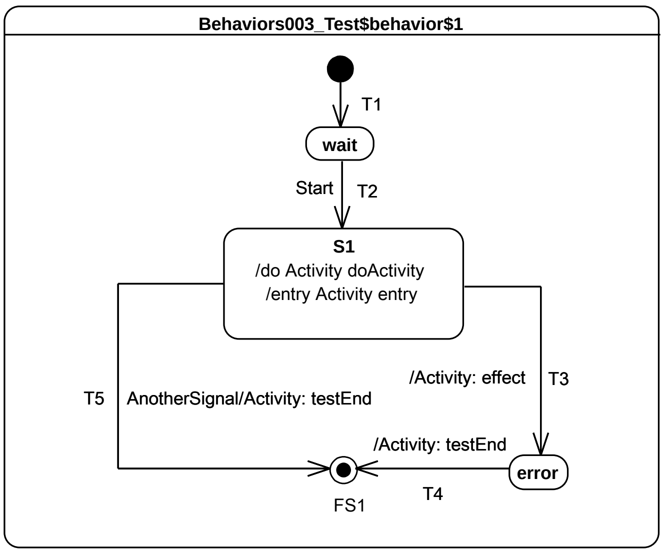


Figure 9.7 - Behavior 003 - A Test Classifier behavior

The *doActivity* behavior which is executed when the *entry* of *S1* finished is presented in Figure 9.8. When started, this behavior completes the execution trace with the message *S1(doActivityPartI)* and blocks until the reception of a *Continue* signal. Only if this signal triggers the continuation of the *doActivity* will the execution trace be completed with the message *S1(doActivityPartII)*.

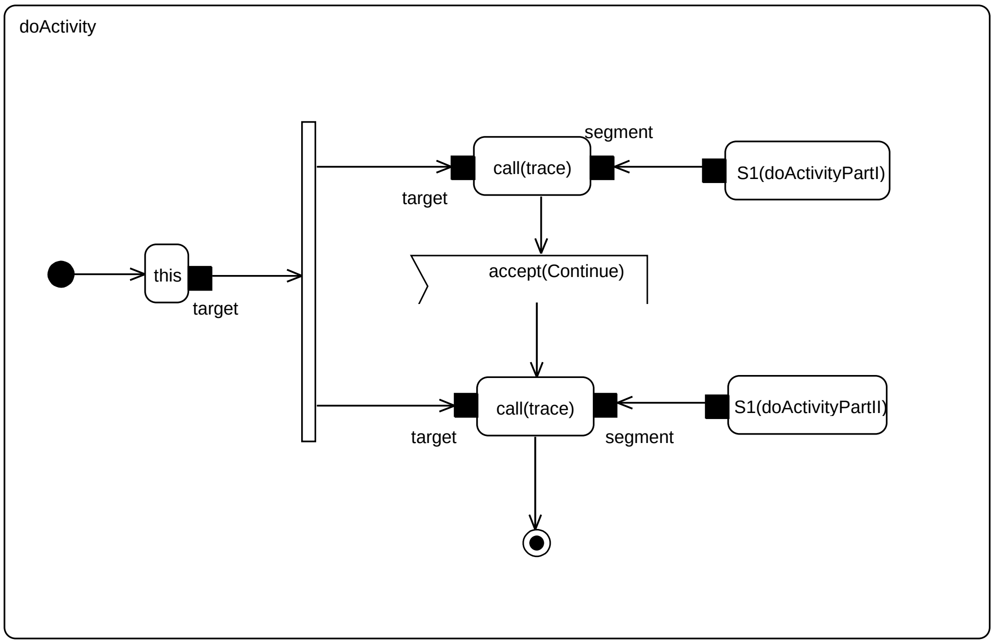


Figure 9.8 - S1 doActivity behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*
- AnotherSignal – received when in configuration *S1*

Generated trace

- S1(entry)::S1(doActivityPartI)

Note. In this test the focus is on validating the assertion that if a *doActivity* is currently being executed by a state and the latter is exited, then the *doActivity* is aborted**.** Consider that the state machine is in configuration *S1* and the *doActivity* that was executed on on its own thread of execution is now blocked waiting for a Continue signal to be dispatched. The next event to be dispatched is AnotherSignal. When dispatched it triggers transition *T5*. The triggering of the latter implies that *S1* is exited and hence its running *doActivity* is aborted. The state machine terminates its execution by reaching the final state.

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]
4	[**AnotherSignal**]	[S1]	[T5]

9.3.2.5 Test Behavior 003-B

Tested state machine

The state machine that is executed for this test is presented in Figure 9.9.

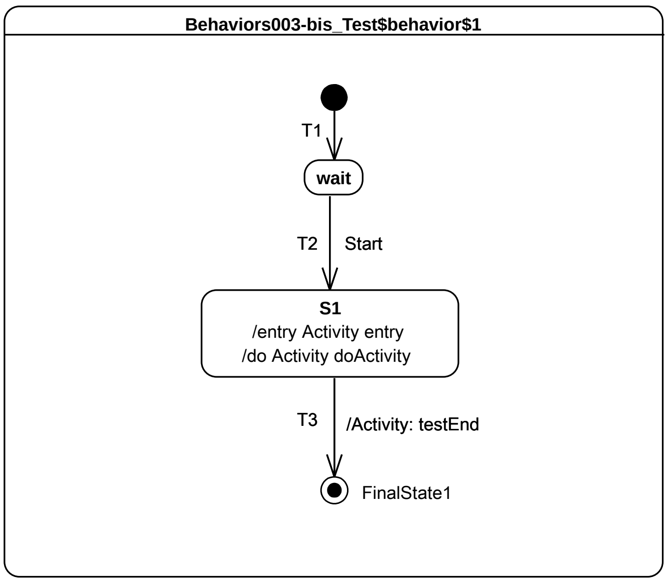


Figure 9.9 - Behavior 003 - B Test Classifier behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*
- Continue – received when in configuration *S1*

Generated trace

- S1(entry)::S1(doActivityPartI)::S1(doActivityPartII)

Note. This test focuses on validating the assertion that when the *doActivity* completes then, if the state is in a situation where it is ready to complete, it generates a completion event. For this test, the *doActivity* that is related to *S1* is the same as the one presented in Figure 9.8. In this test case, the only way for the state machine to terminate its execution is to traverse the completion transition *T3* using the completion event generated by *S1*. The completion event has to be generated so that the *doActivity* behavior started by *S1* completes. In this case, when *Continue* gets dispatched, it cannot be accepted by the state machine since there is no transition that has a trigger to react to this event. However, it can be accepted by the *doActivity* behavior which is currently blocked waiting for a *Continue* event. The acceptance of this event leads to the *doActivity* completing, which implies that, upon completion, state *S1* generates a completion event.

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]

4	[**Continue**]	[S1]	[] - RTC step in the doActivity
5	[CE(S1)]	[S1]	[T3]

9.3.2.6 Test Behavior 004

Tested state machine

The state machine that is executed for this test is presented in Figure 9.10. It is important to note the presence of the internal transition *T3* (see *AnotherSignal/Activity: effect* in *S1* on Figure 9.10) which can be triggered when the state machine is in configuration *S1* and AnotherSignal event is dispatched.. For this test the *doActivity* that is related to *S1* is the same as the one presented in Figure 9.8.

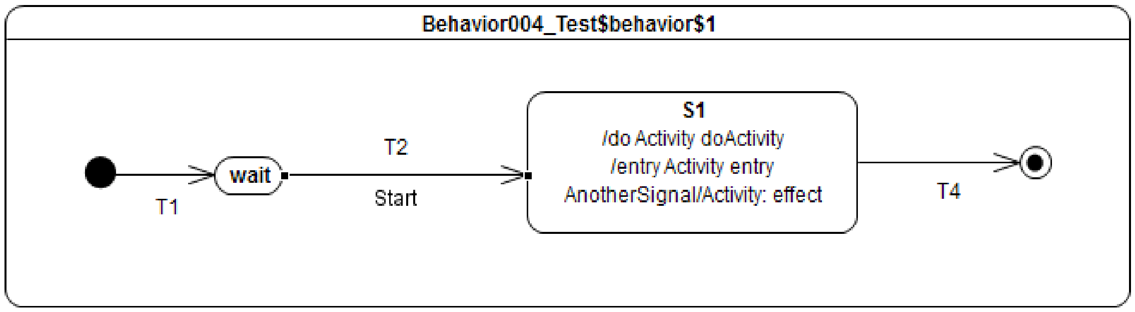


Figure 9.10 - Behavior 004 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*
- AnotherSignal – received when in configuration *S1*
- Continue – received when in configuration *S1*

Generated trace

- S1(entry)::S1(doActivityPartI)::T3(effect)::S1(doActivityPartII)

Note. When in configuration *S1*, *T3* is triggered by the dispatching of *AnotherSignal*. This triggering has no impact on the *doActivity* behavior that is currently running. Indeed the behavior is still blocked waiting for the *Continue* event occurrence. When this event is received, the *doActivity* consumes it and completes its execution.

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]
4	[Continue, **AnotherSignal**]	[S1]	[T3]
5	[**Continue**]	[S1]	[] - RTC step in the doActivity
6	[CE(S1)]	[S1]	[T4]

9.3.3 Transition

9.3.3.1 Transition 001

Tested state machine

The state machine that is executed for this test is presented in Figure 9.11.

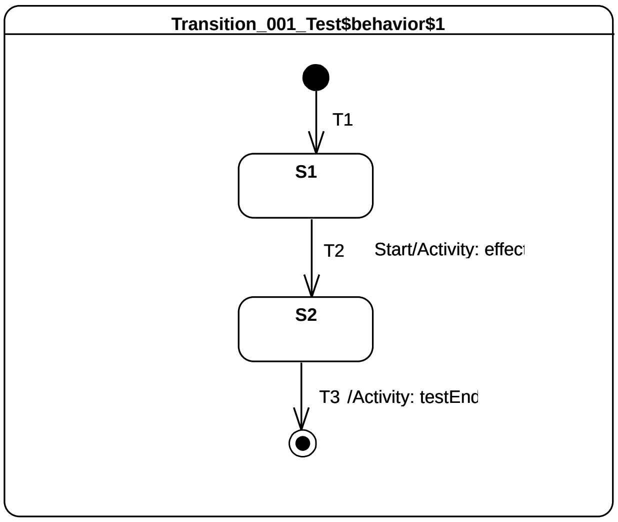


Figure 9.11 - Transition 001 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *S1*.

Generated trace

- T2(effect)

Note. The purpose of this test is to assert that the *effect* behavior of a transition is executed when the latter is traversed. In this test, when *Start* is dispatched it triggers *T2* whose execution adds the message *T2(effect)* to the execution trace.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(S1)**]	[S1]	[]
3	[**Start**]	[S1]	[T2]
4	[**CE(S2)**]	[S2]	[T3]

9.3.3.2 Transition 007

Tested state machine

The state machine that is executed for this test is presented in Figure 9.12.

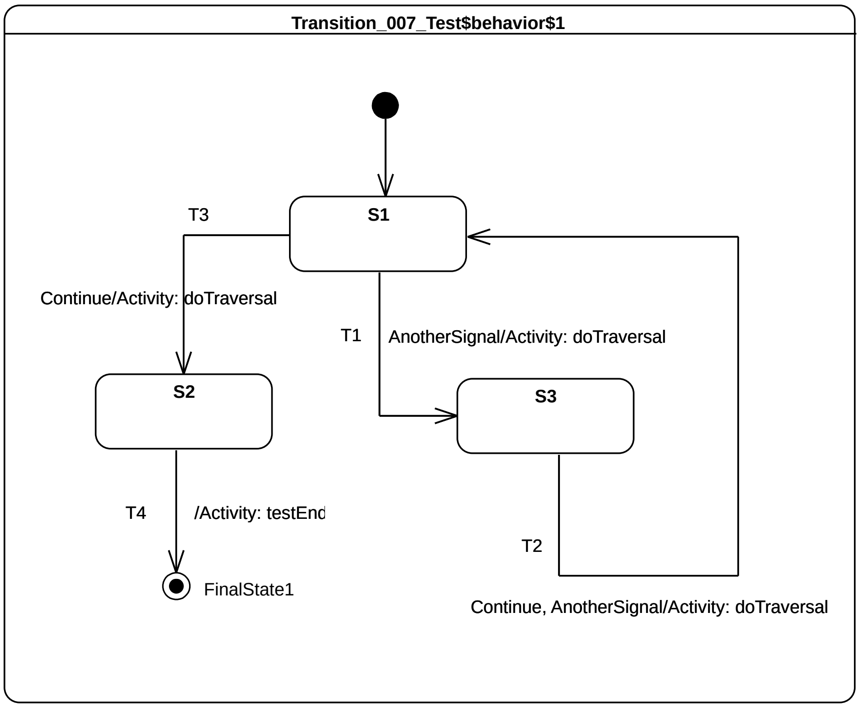

Transition_007_Test$behavior$1
S1
T3
Continue/Activity: doTraversal
T1
AnotherSignal/Activity: doTraversal
S2
S3
T4
/Activity: testEnd
T2
FinalState1
Continue, AnotherSignal/Activity: doTraversal

Figure 9.12 - Transition 007 Test Classifier Behavior

Test executions

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 assert that a transition can be triggered if at least one of its triggers *matches* the dispatched event. Consider that the state machine is in configuration *S1*. When *AnotherSignal* is dispatched, transition *T1* is triggered. This is due to the fact that *T1* declares a trigger for the signal *AnotherSignal*. The state machine then moves into configuration *S3*. There is no difference in the situation where there are multiple triggers declared for a transition (see *T2* in Figure 9.24). If the dispatched event occurrence matches at least one of them, the transition is traversed. Continue then triggers *T2*.

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]

9.3.3.3 Transition 010

Tested state machine

The state machine that is executed for this test is presented in Figure 9.13. It is important to note the presence of an internal transition for *S1* (see *AnotherSignal/Activity: effect* in *S1* on Figure 9.13). This transition can be triggered when an occurrence of *AnotherSignal* is dispatched.

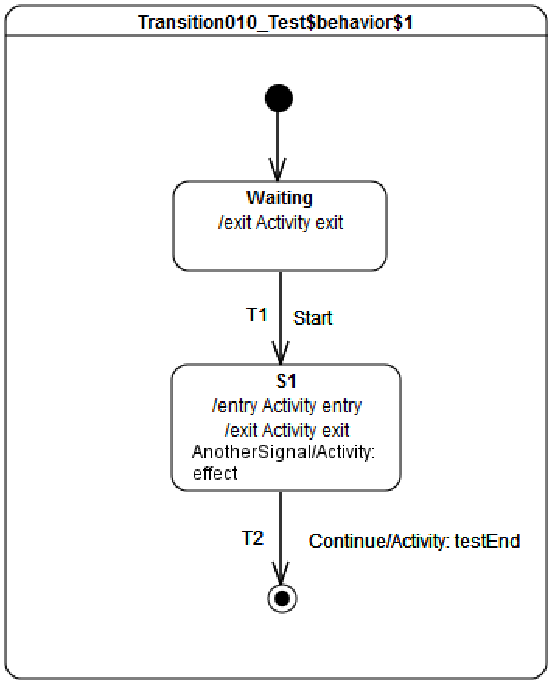

Transition010_Test$behavior$1
Waiting
/exit Activity exit
T1
Start
S1
/entry Activity entry
/exit Activity exit
AnotherSignal/Activity: effect
T2
Continue/Activity: testEnd

Figure 9.13 - Transition 010 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *waiting*.
- AnotherSignal – received when in configuration *S1*.
- AnotherSignal – received when in configuration *S1*.
- Continue – received when in configuration *S1*.

Generated trace

- waiting(exit)::S1(entry)::IT(effect)::IT(effect)::S1(exit)

Note. The purpose of this test is to assert that, when an internal transition is fired ,the source state is not exited and the target state is not entered. The generated trace demonstrates this behavior. Indeed, *S1* is not exited and re-entered when the internal transition *IT* is traversed. In the trace, one can observe that *IT(effect)* appears twice. This illustrates the fact that this transition is triggered for each dispatching of an occurrence of *AnotherSignal*. The dispatching of the *Continue* event occurrence implies that *T2* is traversed and the *S1* exit behavior is executed. The state machine execution completes by reaching the final state.

RTC steps

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

9.3.3.4 Transition 011-A

Tested state machine

The state machine that is executed for this test is presented in Figure 9.14.

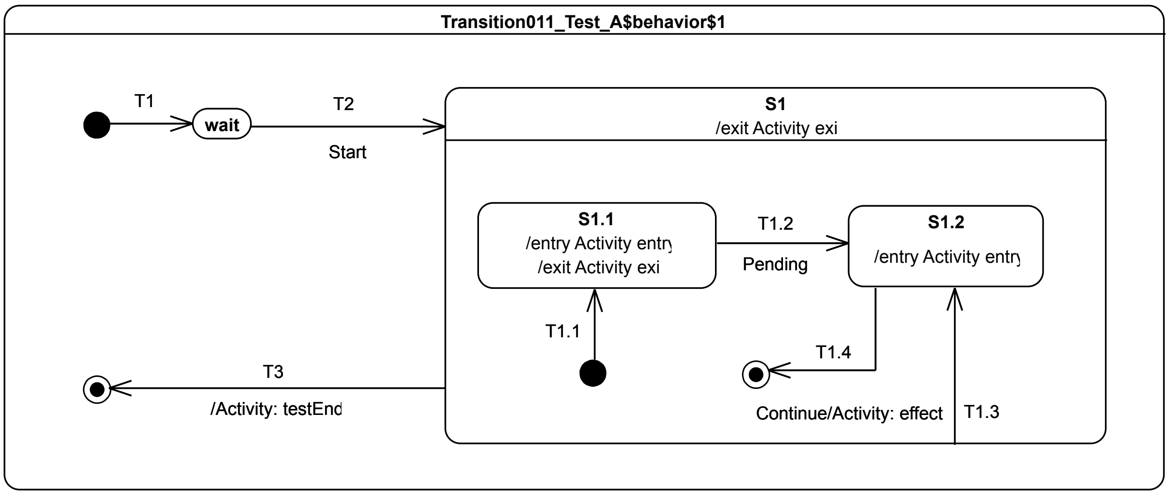


Figure 9.14 - Transition 011- A Test Classifier Behavior

Test executions

Received event occurrence(s)

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

Generated trace

- S1.1(entry)::S1.1(exit)::T1.3(effect)::S1.2(entry)

Note. The purpose of this test is to demonstrate that, when a local transition leaving the containing state is triggered, then the state is not exited. Consider that the state machine is in configuration *S1[S1.1],* and Continue is the next event to be dispatched. . The traversal of *T1.3* implies that *S1.1* is exited, the effect behavior is executed and, finally, *S1.2* is entered. Upon completion of the entry behavior a completion event is generated for *S1.2*. The latter is used to trigger *T1.4* in the next RTC step.

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)]
4	[Continue, **CE(S1.1)**]	[S1[S1.1]]	[]
5	[**Continue**]	[S1[S1.1]]	[T1.3]
6	[**CE(S1.2)**]	[S1[S1.2]]	[T1.4]
7	[**CE(S1)**]	[S1]	[T3]

9.3.3.5 Transition 011-B

Tested state machine

The state machine that is executed for this test is presented in Figure 9.15.

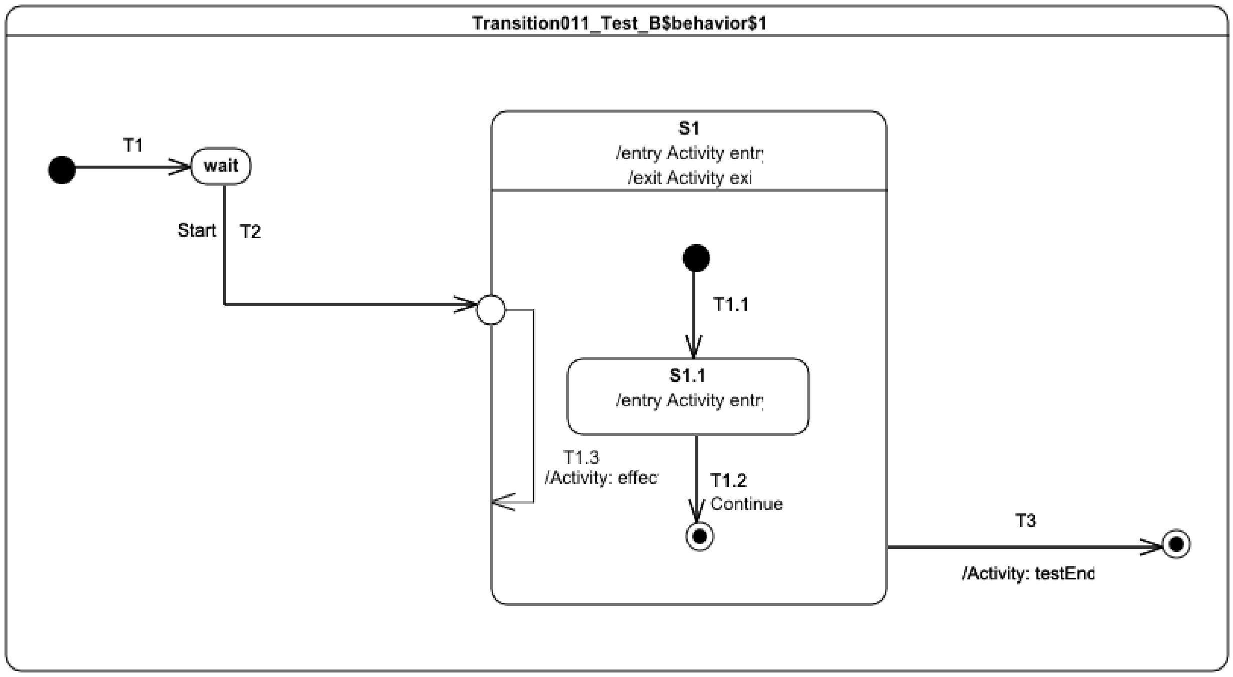


Figure 9.15 - Transition 011 - B Test Classifier behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1*.

Generated trace

- S1(entry)::S1.1(entry)::T1.3(effect)

Note. The purpose of this test is to demonstrate that, if a local transition leaves an entry point owned by the containing state and targets the inside edge of the containing state, then exiting the entry point has no effect on hierarchy of active states and the target is not re-entered since it is already active. Consider that the state machine is configuration *wait*. When *T2* is triggered, the entry point is reached, *S1* is entered and its unique region is entered. The execution of the region starts from the initial pseudostate. Next, the initial transition *T1.1* is traversed, and, finally, *S1.1* is entered. At this point, the RTC step initiated by the dispatching of the Start event occurrence is not finished. Indeed, the continuation transition *T1.3* outgoing from the entry point is traversed. Since S1 is already active, it is not re-entered. This marks the end of the current RTC step. When the Continue event occurrence is dispatched, it triggers *T1.2* which leads to the region completion and hence to the completion of *S1*. The completion event for *S1* is used to trigger T3, which leads to the completion of the state machine execution.

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, T1.3)]
4	[Continue, **CE(S1.1**)]	[S1[S1.1]]	[]
5	[**Continue**]	[S1[S1.1]]	[T1.2]
6	[**CE(S1)**]	[S1]	[T3]

9.3.3.6 Transition 011-C

Tested state machine

The state machine that is executed for this test is presented in Figure 9.16.

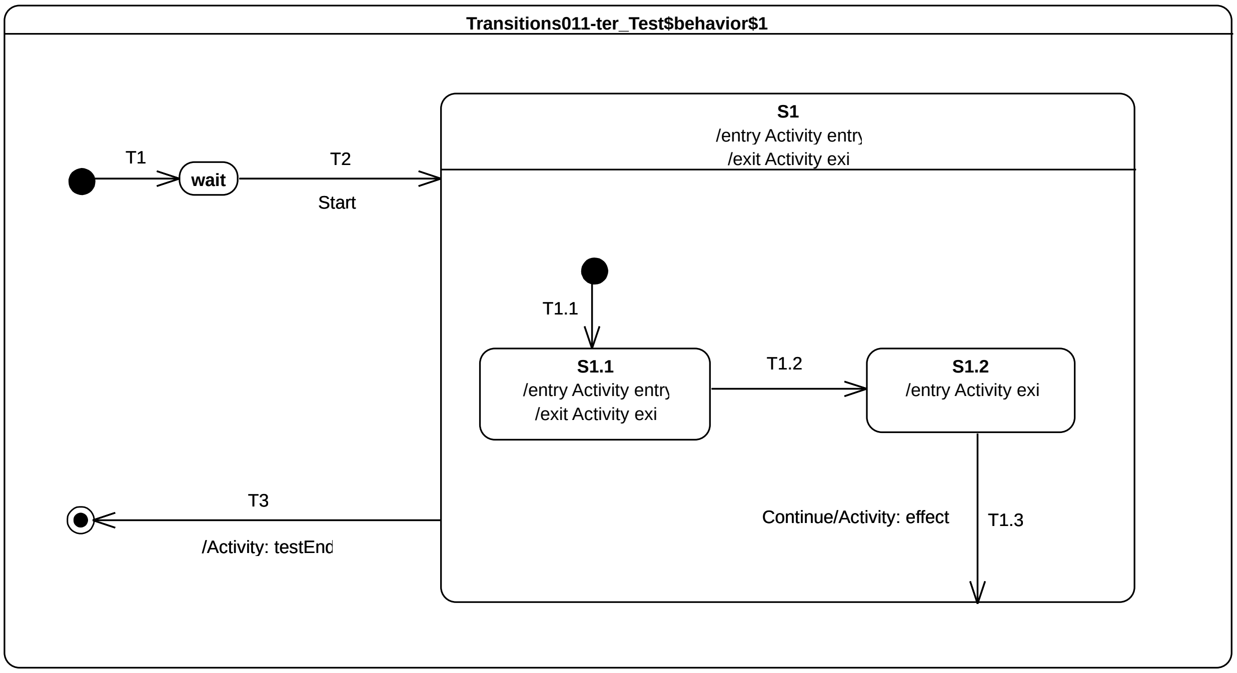


Figure 9.16 - Transition 011 - C Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.

- Continue – received when in configuration *S1[S1.2]*.

Generated trace

- S1(entry)::S1.1(entry)::S1.1(exit)::S1.2(exit)::T1.3(effect)::S1(exit)

Note. The purpose of this test is to demonstrate that, when an external transition is fired, if its source is an internal vertex of the target, then the region of the target that contains (directly or indirectly) the source vertex completes. Consider that the state machine is in configuration *S1[S1.2]*. When *T1.3* is triggered (by the dispatching of Continue), *S1.2* is exited and the effect behavior is executed. However, *S1* is not re-entered (it is already active), but the region that contains the last exited state is completed. The completion of the region implies the generation of a completion event for I. This completion event is used to trigger *T3*.

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)]
4	[**CE(S1.1)**]	[S1[S1.1]]	[T1.2]
5	[Continue, **CE(S1.2)**]	[S1[S1.2]]	[]
6	[**Continue**]	[S1[S1.2]]	[T1.3]
7	[**CE(S1)**]	[S1]	[T3]

9.3.3.7 Transition 011-D

Tested state machine

The state machine that is executed for this test is presented in Figure 9.17.

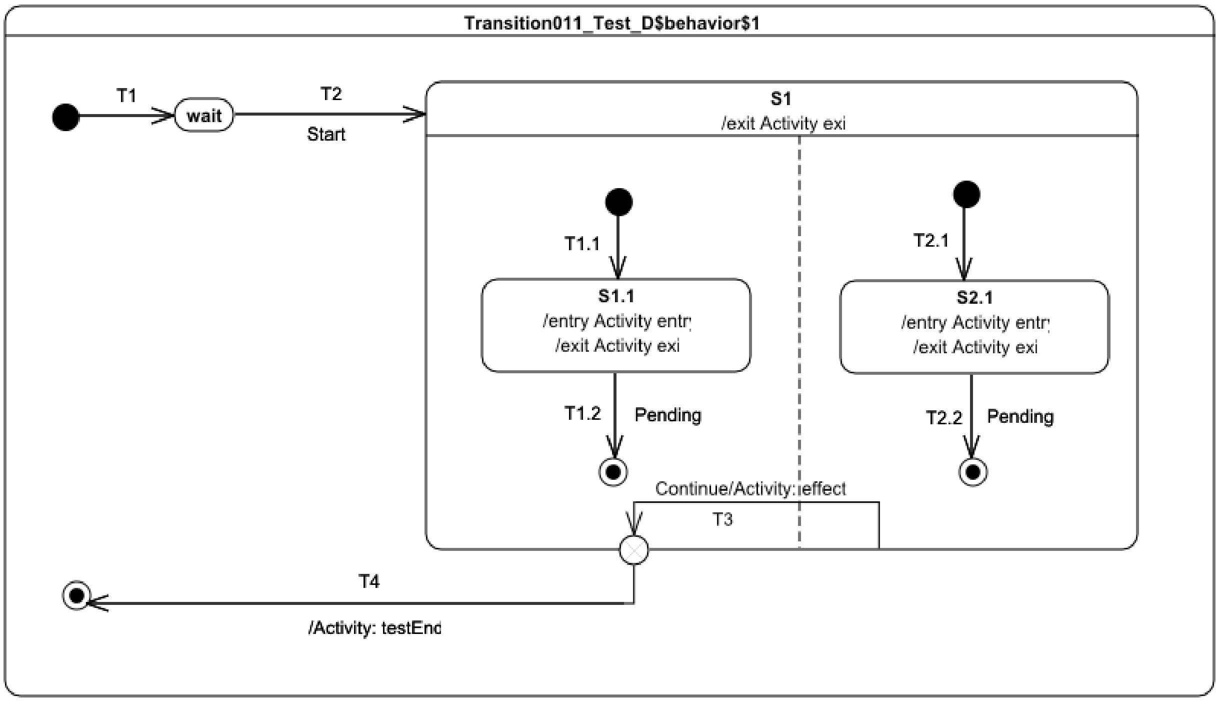


Figure 9.17 - Transition 011- D Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1[S1.1, S2.1].*

Generated trace

- S1.1(entry)::S2.1(entry)::T3(effect)::S1.1(exit)::S2.1(exit)::S1(exit)

Note. The purpose of this test is to demonstrate that, when a local transition leaves the containing state, then this state is not exited and there is no impact on the hierarchy of active states. However, it shows that, if the target of this local transition is an exit point, then the semantics of this pseudostate applies. Hence, all regions owned by the containing state are exited concurrently. When the state machine configuration is *S1[S1.1, S2.1],* the event occurrence *Continue* is dispatched. This leads to the triggering of the local transition *T3*. The source state is not exited, the effect behavior is executed and, finally, the exit point is reached. Hence *S1.1* and *S2.1* are exited immediately, followed by *S1*. The continuation transition fires and leads to the completion of the state machine execution.

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, T2.1)]
4	[Continue, CE(S2.1), **CE(1.1)**]	[S1[S1.1, S2.1]]	[]
5	[Continue, **CE(S2.1)**]	[S1[S1.1, S2.1]]	[]
6	[**Continue**]	[S1[S1.1, S2.1]]	[T3(T4)]

Alternative execution traces

The presence of orthogonal regions in *S1* implies the existence of an alternative execution trace for this test. This trace captures the situation where *S2.1* is exited before *S1.1*.

- S1.1(entry)::S2.1(entry)::T3(effect)::S2.1(exit)::S1.1(exit)::S1(exit)

9.3.3.8 Transition 011-E

Tested state machine

The state machine that is executed for this test is presented in Figure 9.18.

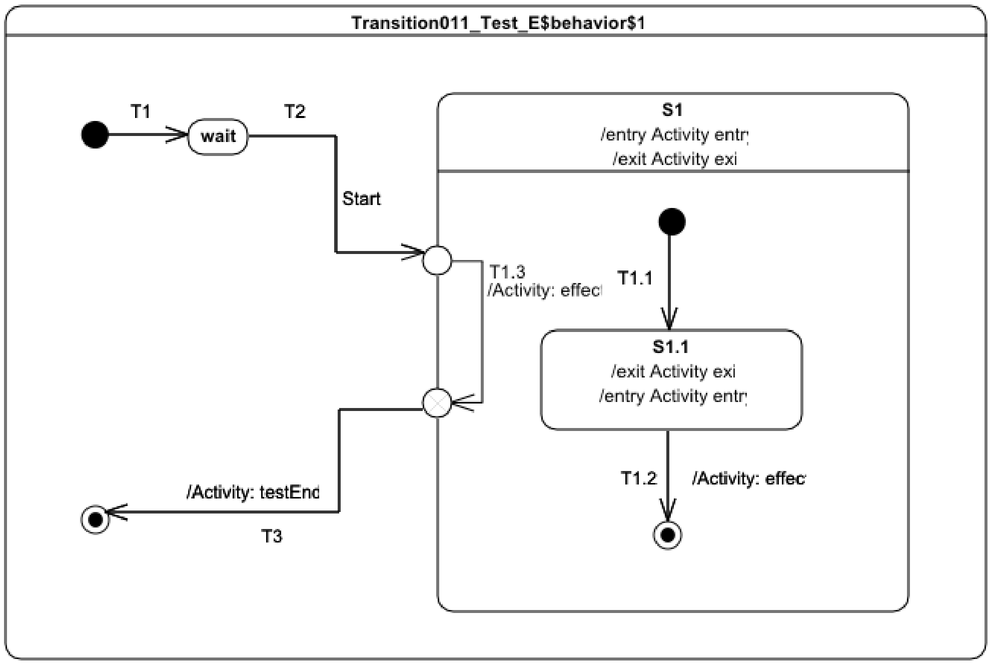


Figure 9.18 - Transition 011 - E Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- S1(entry)::S1.1(entry)::T1.3(effect)::S1.1(exit)::S1(exit)

Note. The purpose of this test is to demonstrate that, when a local transition leaves an entry point and enters an exit point, then exiting the entry point has no effect, there is no impact on the hierarchy of active states however when the exit point is entered the semantics for this pseudostate applies which implies that regions owned by the containing state are exited. Consider the situation where the state machine is in configuration *wait*. When *T2* fires, the entry point is reached, *S1* is entered and the region is entered using the default approach (i.e., an initial transition is sought to start the execution). Hence *S1.1* is entered via the transition *T1.1*. The RTC step initiated by the dispatching of the *Start* event occurrence is not ended. The continuation transition *T1.3* is traversed and its effect behavior is executed. At the point where the exit point is reached, *S1.1* is exited as well as *S1*. The continuation transition *T3* is traversed and leads to the completion of the state machine execution. Transition *T1.2* is never traversed in this test case and, consequently, the *S1* region never 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, T1.3, T3)]

9.3.3.9 Transition 015

Tested state machine

The state machine that is executed for this test is presented in Figure 9.19. The *doActivity* for state S1 state is exactly the same as the one presented in Figure 9.8.

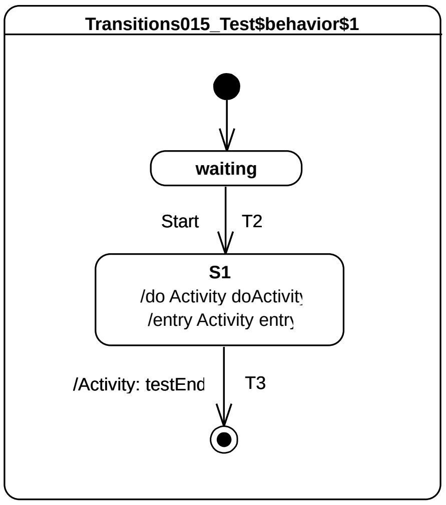


Figure 9.19 - Transition 015 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration waiting.

Generated trace

- S1(entry)::S1(doActivity)

Note. The purpose of this test is to demonstrate that a completion event can be generated for a simple state only if both its *entry* behavior (if any) and its *doActivity* (if any) have completed their executions. Consider the situation where the state machine is in configuration *waiting*. When *T2* is traversed (RTC step started by dispatching *Start*), *S1* is entered and its *entry* behavior is executed. As soon as the *entry behavior* terminates its execution, the *doActivity* behavior is started asynchronously. When this *doActivity* behavior completes its execution, a completion event is generated by *S1* (its *entry* and *doActivity* behaviors are now terminated). This completion event is used in the next RTC step to trigger *T3*.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Start, **CE(waiting)**]	[waiting]	[]
3	[**Start**]	[waiting]	[T2]
4	[**CE(S1)**]	[S1]	[T3]

9.3.3.10 Transition 016

Tested state machine

The state machine that is executed for this test is presented in Figure 9.20.

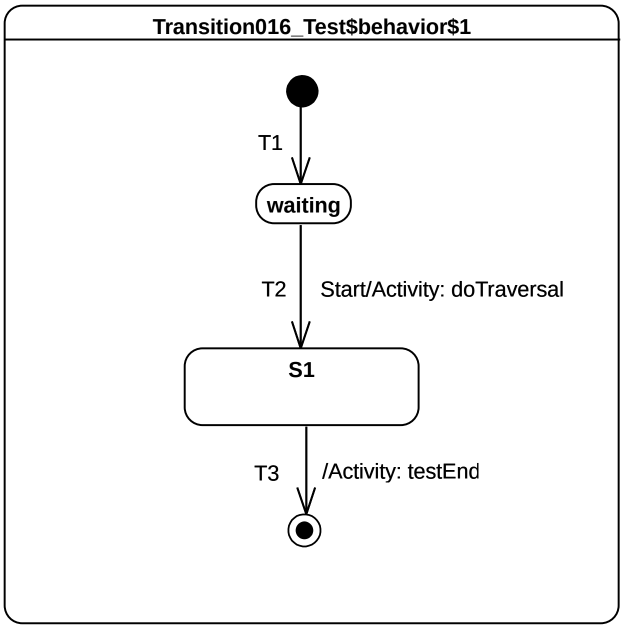


Figure 9.20 - Transition 016 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *waiting*.

Generated trace

- T2(effect)

Note. The purpose of this test is to demonstrate that, if the entered state has no entry or doActivity behaviors, then a completion event is generated upon its entry. Consider the situation where the state machine is in configuration *waiting*. When *Start* is dispatched, *T2* is triggered. Hence, the effect behavior of *T2* is executed and *S1* is entered. In this test case *S1* has no entry or doActivity behaviors, so the completion event is generated when it is entered. The completion event generated for *S1* is used in the next RTC step to trigger *T3,* which is a completion transition.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Start, **CE(waiting)**]	[waiting]	[]
3	[**Start**]	[waiting]	[T2]
4	[**CE(S1.1)**]	[S1]	[T3]

9.3.3.11 Transition 017

Tested state machine

The state machine that is executed for this test is presented in Figure 9.21.

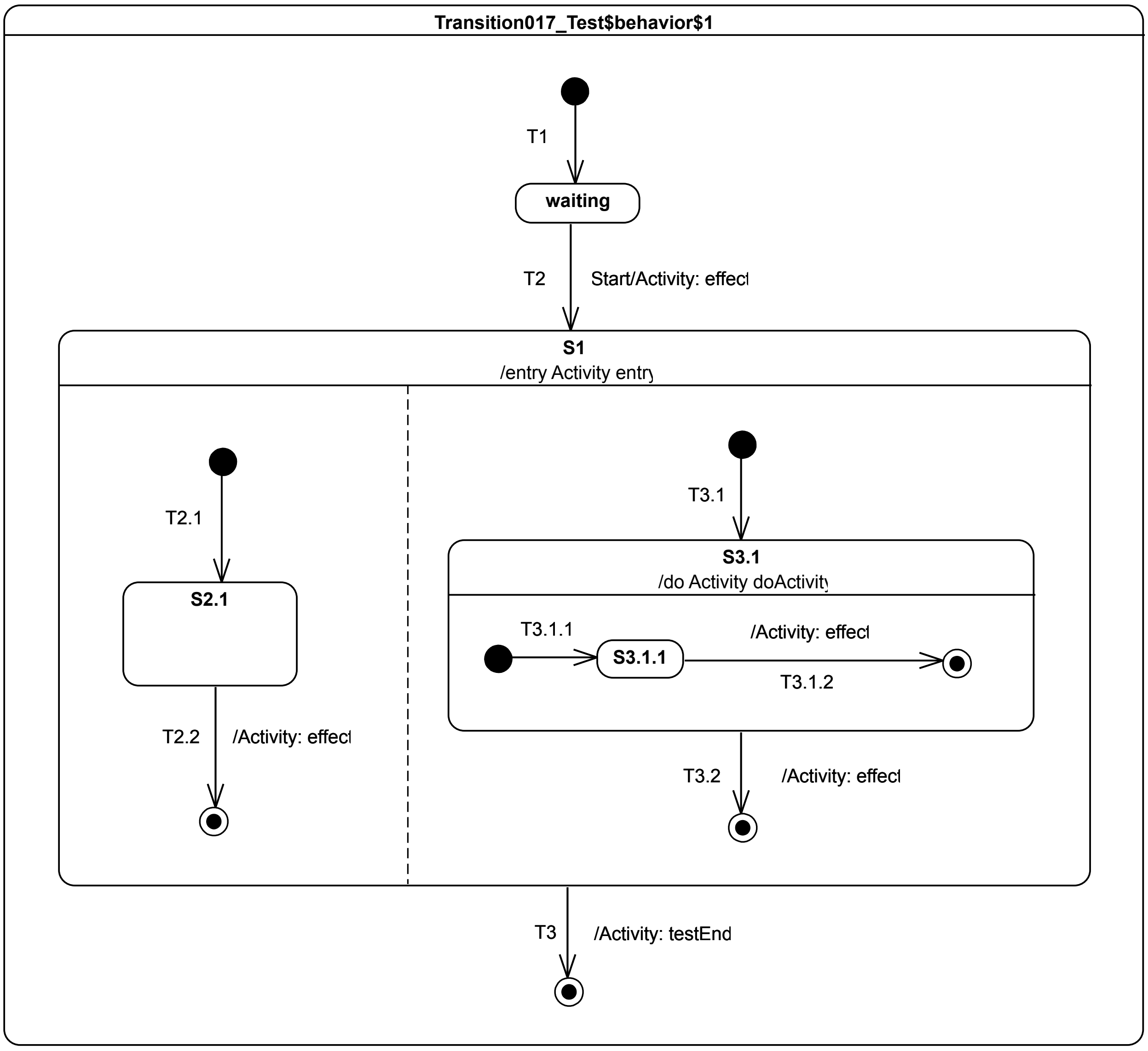


Figure 9.21 - Transition 017 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *waiting*.

Generated trace

- T2(effect)::S1(entry)::T2.2(effect)::T3.1.2(effect)::S3.1(doActivity)::T3.2(effect)

Notes. The purpose of this test is to demonstrate that a composite state can only complete (i.e., generate a completion event) if its entry behavior (if any) has completed, its doActivity behavior has completed and all of its regions have also completed (see Error: Reference source not found8.5.3 for conditions enabling a region to complete). Consider the situation where the state machine is in configuration *waiting*. When Start is dispatched, *T2* fires and implies the entrance of *S1*. Each region of *S1* is entered concurrently using the default entry approach. Assume that for the described execution:

a. The doActivity for *S3.1* starts immediately and includes its message in the execution trace before any other RTC step is performed in the state machine.

b. The completion events generated by *S2.1* and *S3.1* are added in the event pool in the following order: *CE(S1.2)* is first and *CE(S3.1.1)* second. With respect to these assumptions the following execution steps occur: *T2.2* is triggered by the dispatching of *CE(S2.1)*. When the final state is reached no completion event is generated since the right hand side region of the state machine is still running.

c. *T3.1.2* is triggered by the dispatching of *CE(S3.1.2)*. When the final state is reached a completion event is generated for *S3.1* since its doActivity behavior has already terminated its execution.

d. *T3.2* is triggered by the dispatching of the completion event generated by *S3.1*. This leads to the completion of *S1*.

e. The completion event generated by *S1* will be then used to trigger *T3*. The traversal of this transition leads to a final state upon which the state machine execution completes.

RTC steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(waiting)**]	[waiting]	[]
3	[**Start**]	[waiting]	[T2(T2.1, T3.1(T3.1.1))]
4	[CE(S3.1.1), **CE(S2.1)**]	[S1[S2.1, S3.1[S3.1.1]]]	[T2.2]
5	[**CE(S3.1.1)**]	[S1[S3.1[S3.1.1]]]	[T3.1.2]
6	[**CE(3.1)**]	[S1[S3.1]	[T3.2]
7	[**CE(S1)**]	[S1]	[T3]

Alternative execution traces

The state machine specifies parallelism through *S1,* which owns orthogonal regions and *S3.1,* which provides a doActivity that will evolve on its own thread of execution. For this particular state machine, the following set of alternative execution traces (the one presented above is not included in that set) is allowed with regards to the UML state machines semantics:

1. T2(effect)::S1(entry)::T2.2(effect)::S3.1(doActivity)::T3.1.2(effect)::T3.2(effect)

2. T2(effect)::S1(entry)::T2.2(effect)::T3.2(effect)::S3.1(doActivity)::T3.1.2(effect)

3. T2(effect)::S1(entry)::T2.2(effect)::T3.2(effect)::T3.1.2(effect)::S3.1(doActivity)

4. T2(effect)::S1(entry)::S3.1(doActivity)::T2.2(effect)::T3.1.2(effect)::T3.2(effect)

5. T2(effect)::S1(entry)::S3.1(doActivity)::T3.1.2(effect)::T2.2(effect)::T3.2(effect)
6. T2(effect)::S1(entry)::T3.1.2(effect)::T2.2(effect)::S3.1(doActivity)::T3.2(effect)
7. T2(effect)::S1(entry)::T3.1.2(effect)::S3.1(doActivity)::T2.2(effect)::T3.2(effect)

The alternative trace 6 denotes an execution in which the doActivity behavior includes its trace message after that the RTC steps related to the dispatching of completion events CE(S3.1.1) and CE(S2.1) have been performed. The table below describes the different steps leading to the production of this execution trace.

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(waiting)**]	[waiting]	[]
3	[**Start**]	[waiting]	[T2[T2.1, T3.1(T3.1.1)]]
4	[CE(S2.1), **CE(S3.1.1)**]	[S1[S2.1, S3.1[S3.1.1]]]	[T3.1.2]
5	[CE(3.1), **CE(S2.1)**]	[S1[S2.1, S3.1]]	[T2.2]
6	[**CE(3.1)**]	[S1[S2.1]]	[T3.2]
7	[**CE(S1)**]	[S1]	[T3]

9.3.3.12 Transition 019

Tested state machine

The state machine that is executed for this test is presented in Figure 9.22.

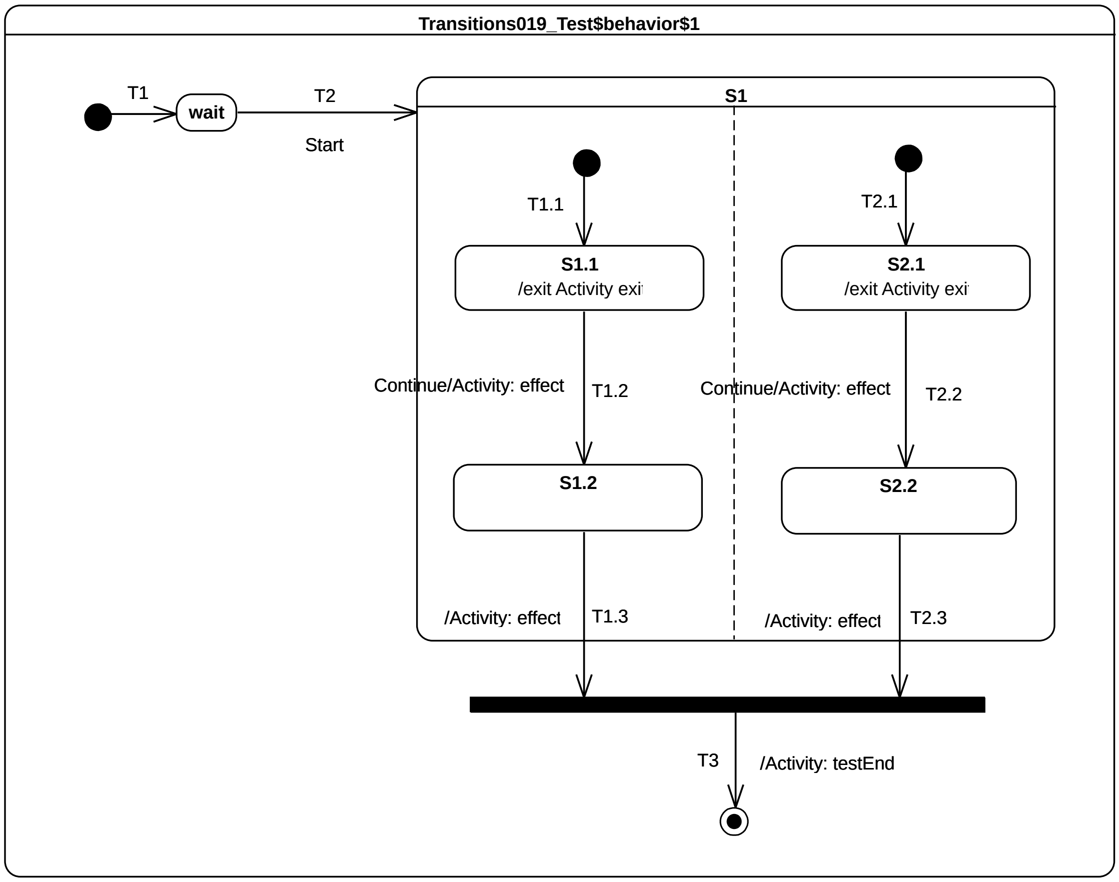


Figure 9.22 - Transition 019 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1[S1.1, S2.1]*.

Generated trace

- S1.1(exit)::T1.2(effect)::S2.1(exit)::T2.2(effect)::T1.3(effect)::T2.3(effect)

Note. The purpose of that test is to demonstrate that the firing order of *T1.3* and *T2.3* is related to the order in which *S1.2* and *S1.3* generate their respective completion events. Consider the situation where the state machine is in configuration *S1[S1.1, S1.2]*. The dispatching of the *Continue* event occurrence implies a simultaneous triggering of both *T1.2* and *T2.2*. Hence, due to the same event occurrence, two completion are generated by *S1.2* and *S2.2* respectively. The order in which these completion events will be dispatched is the order in which they were placed in the event pool. Assuming that *CE(S1.2)* is first, then T1.3 will be fired first. *CE(2.1)* will be triggered next and the join node prerequisite will be satisfied.

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, T2.1)]
4	[Continue, CE(S1.1), **CE(S2.1)**]	[S1[S1.1, S2.1]]	[]
5	[Continue, **CE(S1.1)**]	[S1[S1.1, S2.1]]	[]
6	[**Continue**]	[S1[S1.1, S2.1]]	[T1.2, T2.2]
7	[CE(S2.2), **CE(S1.2)**]	[S1[S1.2, S2.2]]	[T1.3]
8	[**CE(S2.2)**]	[S1[S2.2]]	[T2.3(T3)]

Alternative execution traces

The state machine specifies parallelism, since state S1 owns two orthogonal regions. Hence, different execution traces are possible while still conforming to UML semantics. The set of traces below describe the alternative execution traces that can be observed at runtime.

1. S1.1(exit)::S2.1(exit)::T1.2(effect)::T2.2(effect)::T1.3(effect)::T2.3(effect)
2. S1.1(exit)::S2.1(exit)::T2.2(effect)::T1.2(effect)::T2.3(effect)::T1.3(effect)
3. S2.1(exit)::S1.1(exit)::T2.2(effect)::T1.2(effect)::T2.3(effect)::T1.3(effect)
4. S2.1(exit)::S1.1(exit)::T1.2(effect)::T2.2(effect)::T1.3(effect)::T2.3(effect)
5. S2.1(exit)::T2.2(effect)::S1.1(exit)::T1.2(effect)::T2.3(effect)::T1.3(effect)

The alternative trace 2 denotes an execution where *T2.3* effect behavior is executed before *T1.3* effect behavior. The table below describes the RTC steps leading to the production of this execution trace.

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, T2.1)]
4	[Continue, CE(S2.1), **CE(S1.1)**]	[S1[S1.1, S2.1]]	[]
5	[Continue, **CE(S2.1)**]	[S1[S1.1, S2.1]]	[]

6	[**Continue**]	[S1[S1.1, S2.1]]	[T1.2, T2.2]
7	[CE(S1.2), **CE(S2.2)**]	[S1[S1.2, S2.2]]	[T2.3]
8	[**CE(S1.2)**]	[S1[S1.2]]	[T1.3(T3)]

9.3.3.13 Transition 020

Tested state machine

The state machine that is executed for this test is presented in Figure 9.23.

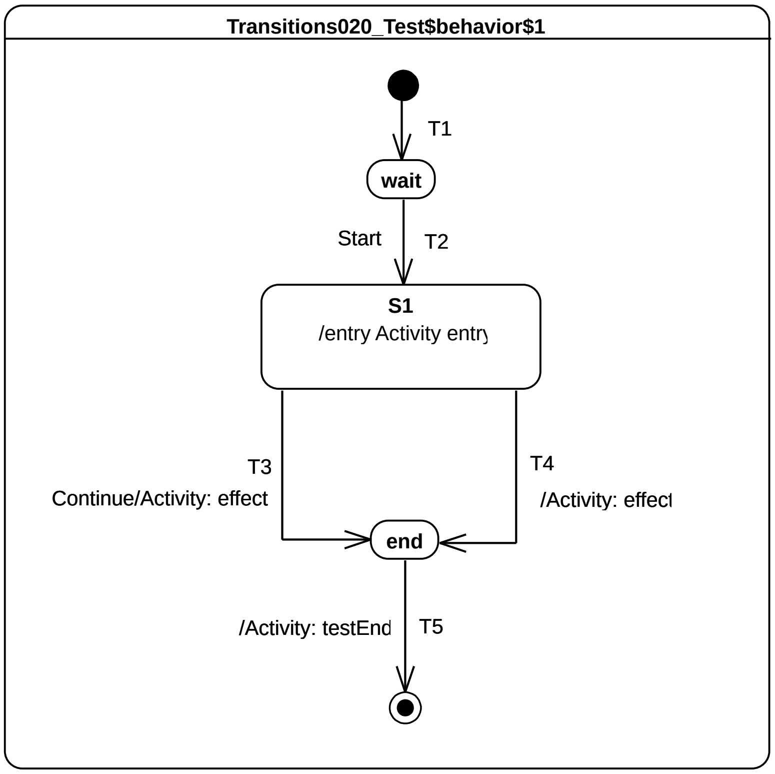


Figure 9.23 - Transition 020 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1*.

Generated trace

- S1(entry)::T4(effect)

Note. The purpose of this test is to demonstrate that completion event occurrences have priority over other event occurrences already present at the pool, except for other completion event occurrences. Consider the situation where the state machine is in configuration *wait*. When Start is dispatched, transition *T2* is triggered. This brings the state machine into configuration *S1*. The entry of *S1* results in the execution of its entry behavior. Upon the termination of the execution of this behavior, *S1* is ready to generate a completion event. The latter is placed at the head of the event pool. Consequently, it is given priority over non-completion event occurrence(s) already present in the pool. Therefore, the next RTC step will begin by the dispatching of the completion event occurrence generated for *S1*. This event occurrence will trigger transition *T4*. The *Continue* event occurrence will never be dispatched.

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]
4	[Continue, **CE(S1.1)**]	[S1]	[T4]
5	[Continue, **CE(end)**]	[end]	[T5]

9.3.3.14 Transition 022

Tested state machine

The state machine that is executed for this test is presented in Figure 9.24.

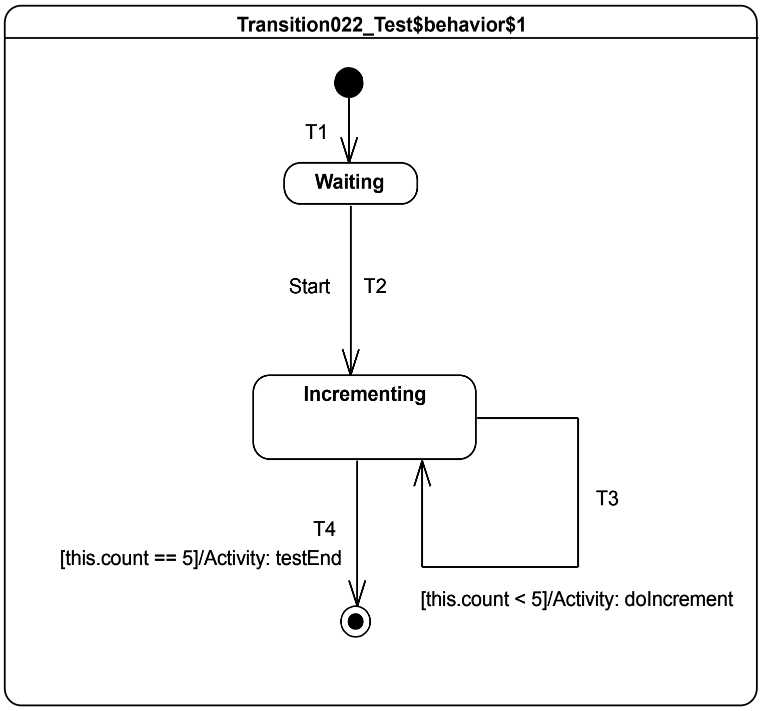

Transition022_Test$behavior$1
T1
Waiting
Start
T2
Incrementing
T3
T4
[this.count == 5]/Activity: testEnd
[this.count < 5]/Activity: doIncrement

Figure 9.24 - Transition 022 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *waiting*.

Generated trace

- T3(effect)::T3(effect)::T3(effect)::T3(effect)::T3(effect)

Note. The purpose of this test is to demonstrate that, if a transition has a guard, then the guard is evaluated. In addition if the result of the evaluation is *false,* then the transition cannot be traversed. Conversely if the result is *true,* then it can be traversed. The intent of the test is to increment the value of a property of the test class that has, as its classifier behavior, the state machine presented in Figure 9.24. The property is initialized to zero when the test object is initialized, and its value is incremented until it reaches the value 5. The state machine that implements this behavior uses guarded transitions (see *T3* and *T4*). When the *Start* event occurrence is dispatched, *T2* fires and *Incrementing* is entered. The completion event generated for that state is used to trigger either *T4* or *T3* based on their guard evaluations.

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]
4	[CE(Incrementing)]	[Incrementing]	[T3]
5	[CE(Incrementing)]	[Incrementing]	[T3]
6	CE(Incrementing)	[Incrementing]	[T3]
7	CE(Incrementing)	[Incrementing]	[T3]
8	CE(Incrementing)	[Incrementing]	[T3]
9	CE(Incrementing)	[Incrementing]	[T4]

9.3.3.15 Transition 023

Tested state machine

The state machine that is executed for this test is presented in Figure 9.25**.**

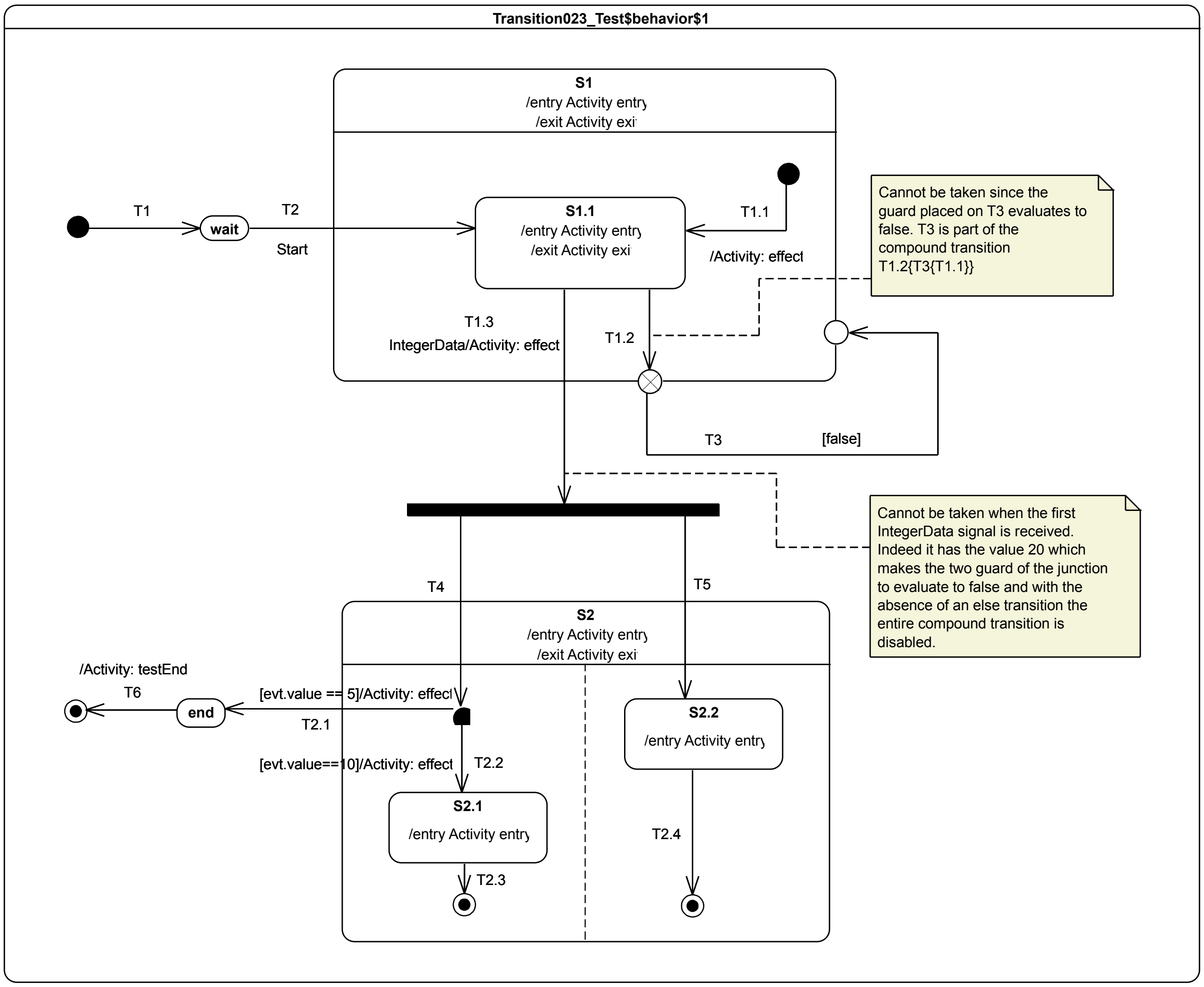

Transition023_Test$behavior$1
S1
/entry Activity entry
/exit Activity exit
T1
wait
T2
Start
S1.1
/entry Activity entry
/exit Activity exit
T1.1
/Activity: effect
Cannot be taken since the guard placed on T3 evaluates to false. T3 is part of the compound transition T1.2{T3{T1.1}}
T1.3
IntegerData/Activity: effect
T1.2
T3
[false]
Cannot be taken when the first IntegerData signal is received. Indeed it has the value 20 which makes the two guard of the junction to evaluate to false and with the absence of an else transition the entire compound transition is disabled.
T4
T5
S2
/entry Activity entry
/exit Activity exit
/Activity: testEnd
T6
end
[evt.value == 5]/Activity: effect
T2.1
[evt.value==10]/Activity: effect
T2.2
S2.1
/entry Activity entry
T2.3
S2.2
/entry Activity entry
T2.4

Figure 9.25 - Transition 023 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.
- IntegerData – received when in configuration *S1[S1.1]*. The property *value* has the value 20.

- IntegerData – received when in configuration *S1[S1.1]*. The property *value* has the value 5.

Generated trace

- S1(entry)::S1.1(entry)::S1.1(exit)::S1(exit)::T1.3(effect)[in=5]::S2(entry)[in=5]::S2.2(entry)[in=5]::S2(exit)[in=5]::T2.1(effect)[in=5]

  Note. The purpose of the test is to assess that, when an event occurrence is about to be dispatched, a static analysis is performed to ensure that from the current state machine configuration we can reach a stable state machine configuration. If no path can be found from the current state machine configuration to the next configuration, then the event occurrence will be lost and no RTC step is initiated.

  In this test case, the state machine receives three event occurrences: *Start, IntegerData(20)* and *IntegerData(5)*. The dispatching of the *Start* event brings the state-machine to the configuration *S1[S1.1]* due to the triggering of T2. The configuration will remain the same until *IntegerData(5)* is dispatched. This can be explained because:

  - The completion event generated for *S1.1* when dispatched cannot bring the state machine to a valid configuration. Indeed event if the completion event could have been used to trigger the compound transition starting with *T1.2, T3* could not have been traversed since its guard evaluated to false.
  - The received I*ntegerData(20)* signal event cannot bring the state machine to a valid configuration. Indeed even if the event occurrence could have been used to trigger the compound transition starting with *T1.3* neither *T2.1* nor *T2.2* could have been traversed since their guards both evaluate to false (because of the value associated with the signal event occurrence). Hence *IntegerData(20)* is lost.

  When *IntegerData(5)* is finally dispatched it triggers the traversal of the compound transition *T1.3[T4[T2.1, T2.2], T5]*.

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]
4	[IntegerData(20), **CE(S1.1)**]	[S1[S1.1]]	[]
5	[**IntegerData(20)**]	[S1[S1.1]]	[]
6	[**IntegerData(5)**]	[S1[S1.1]]	[T1.3[T4[T2.1], T5]]
7	[**CE(end)**]	[end]	[T6]

Alternative execution traces

This state machine specifies parallelism due to the Fork pseudostate. This implies that an alternative execution trace is possible for this state machine. This execution trace is is specified below.

- S1(entry)::S1.1(entry)::S1.1(exit)::S1(exit)::T1.3(effect)[in=5]::S2(entry)[in=5]::S2(exit)[in=5]::T2.1(effect)[in=5]

The difference from the previous trace is that the entry behavior of *S2.2* is not executed. This is possible because, when the compound transition *T1.3[T4[T2.1], T5]* is executed, state *S2* might be exited through *T2.1* before *S2.2* was entered.

9.3.4 Event

9.3.4.1 Overview

Test cases presented in this subclause concern the dispatching and the acceptance of event occurrences in a state machine context.

9.3.4.2 Event 001

Tested state machine

The state machine that is executed for this test is presented in Figure 9.26.

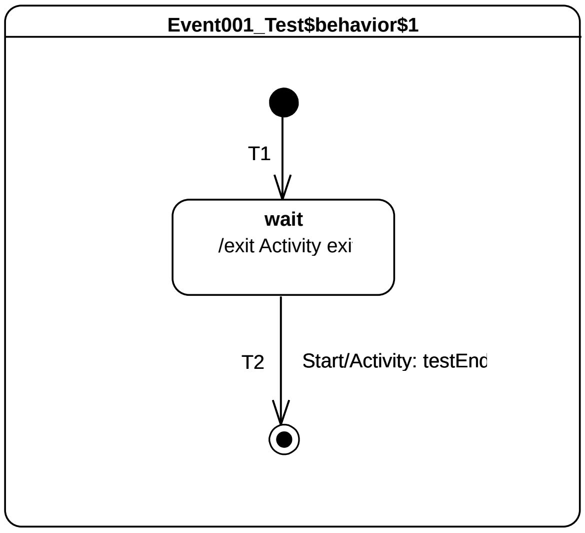


Figure 9.26 - Event 001 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- wait(exit)

Note. The purpose of this test is to show that, upon creation, the tested state machine immediately starts its execution. This execution always starts from the initial pseudostate owned by the region of the state machine. In the context of this test, the initial RTC step implies that transition *T1* is traversed and state *wait* is entered. At the end of this RTC step, the state machine execution enters a wait point (i.e., a stable configuration). As required per UML, the state machine is only able to leave this configuration when a *Start* event occurrence is dispatched from the event pool.

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]

9.3.4.3 Event 002

Tested state machine

The state machine that is executed for this test is presented in Figure 9.27.

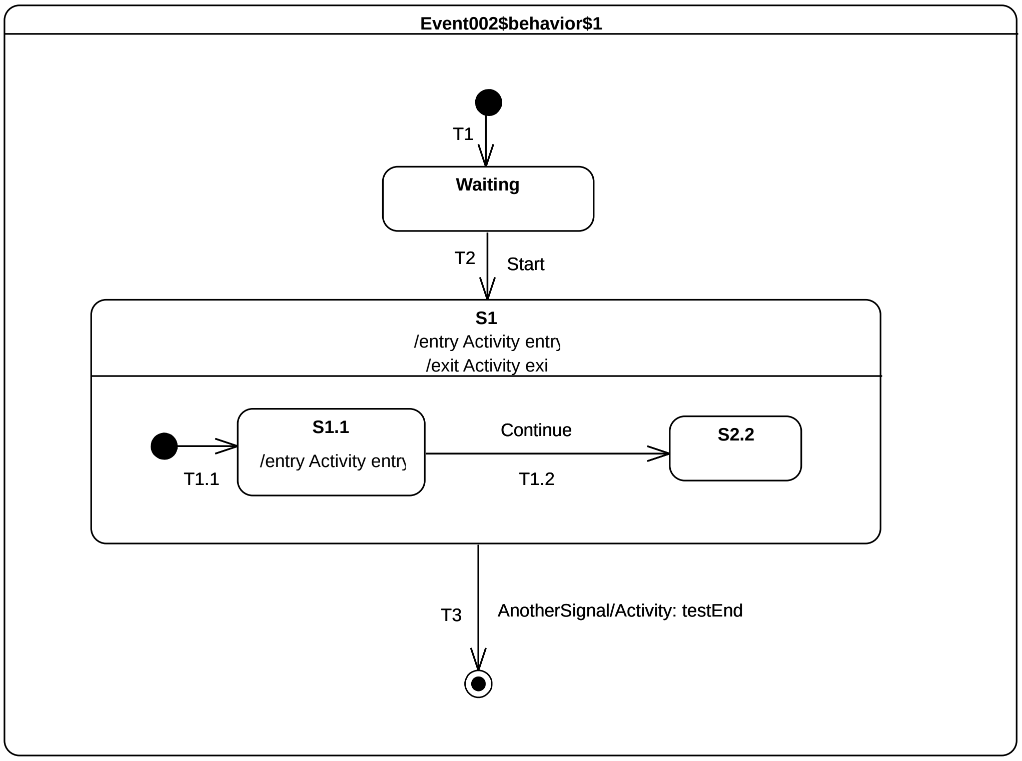


Figure 9.27 - Event 002 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *waiting*.
- AnotherSignal received when in configuration *S1[S1.1]*.

Generated trace

- S1(entry)::S1.1(entry)::S1(exit)

Note. The purpose of this test is to demonstrate that, if a trigger of a transition outgoing an active state matches the dispatched event, then this transition is triggered and a single state machine step is executed. At the point where *AnotherSignal* event occurrence is dispatched, the state machine is in configuration *S1[S1.1]*. This implies that, when *T3* is triggered, state *S1.1* is exited, followed immediately by *S1*. The state machine execution completes by reaching the final state. This illustrates the realization of a RTC step initiated by the dispatching of an event occurrence.

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)]
4	[AnotherSignal, **CE(S1.1)**]	[S1[S1.1]]	[]
5	[AnotherSignal]	[S1[S1.1]]	[T3]

9.3.4.4 Event 008

Tested state machine

The state machine that is executed for this test is presented in Figure 9.28.

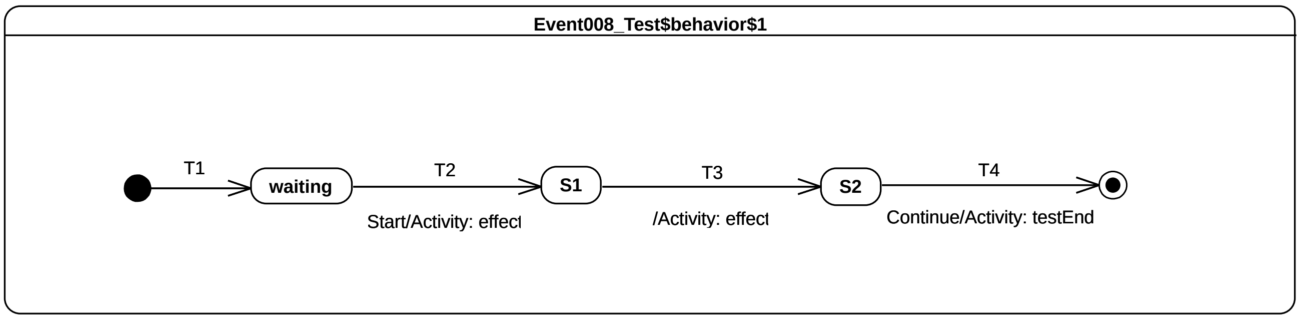


Figure 9.28 - Event 008 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – when received in configuration *waiting*.
- Continue – when received in configuration *S2*.

Generated trace

- T2(effect)::T3(effect)

Note. The purpose of this test is to demonstrate that, if an event occurrence cannot be accepted by a state machine (i.e., cannot be deferred and does not trigger any transition in the current state machine configuration), then this event occurrence is lost. When the *Start* event occurrence is dispatched, the state machine is in configuration *waiting*. The dispatching of this event implies that *T2* is triggered and *S1* is entered. Since *S1* is a simple state with no entry and doActivity behaviors, a completion event occurrence is generated for that state when entered. The dispatching of this completion event occurrence triggers *T3*. Consequently, its effect behavior is executed and *S2* is entered. The completion event occurrence generated by *S2* is dropped. This occurs when there is no possibility to use it for triggering an outgoing transition from that state. The only way to exit *S2* is to receive and dispatch a *Continue* event occurrence.

RTC steps

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

9.3.4.5 Event 009

Tested state machine

The state machine that is executed for this test is presented in Figure 9.29.

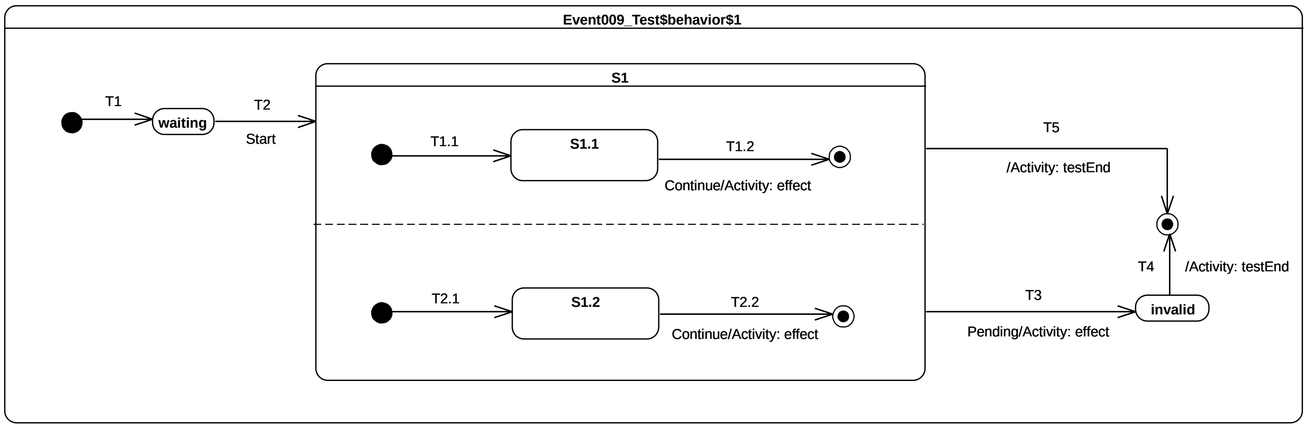


Figure 9.29 - Event 009 Classifier behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *waiting*.
- Continue – received when in configuration *S1[S1.1, S1.2]*.
- Pending – received when in configuration *S1[S1.1, S1.2]*.

Generated trace

- T1.2(effect)::T2.2(effect)

Note. The *Continue* event occurrence is dispatched when the state machine execution is in configuration *S1[S1.1, S1.2]*. The dispatching of this event triggers simultaneously transitions *T1.2* and *T2.2*, which are located in different orthogonal regions. This leads to the completion of the regions of *S1* so that a completion event occurrence is generated for that state. This is used to trigger the completion transition *T5*. Note that transition *T3* is never triggered, because the completion event occurrence that triggers transition *T5* has the priority for dispatching over the *Pending* event occurrence.

RTC steps

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

7	[Pending, **CE(S1)**]	[S1]	[T5]

Alternative execution traces

The presence of two transitions that are placed in orthogonal regions and that can fire on the same event occurrence implies the possibility to observe an alternative execution trace for this test case.

- T2.2(effect)::T1.2(effect)

9.3.4.6 Event 010

Tested state machine

The state machine that is executed for this test is presented in Figure 9.30.

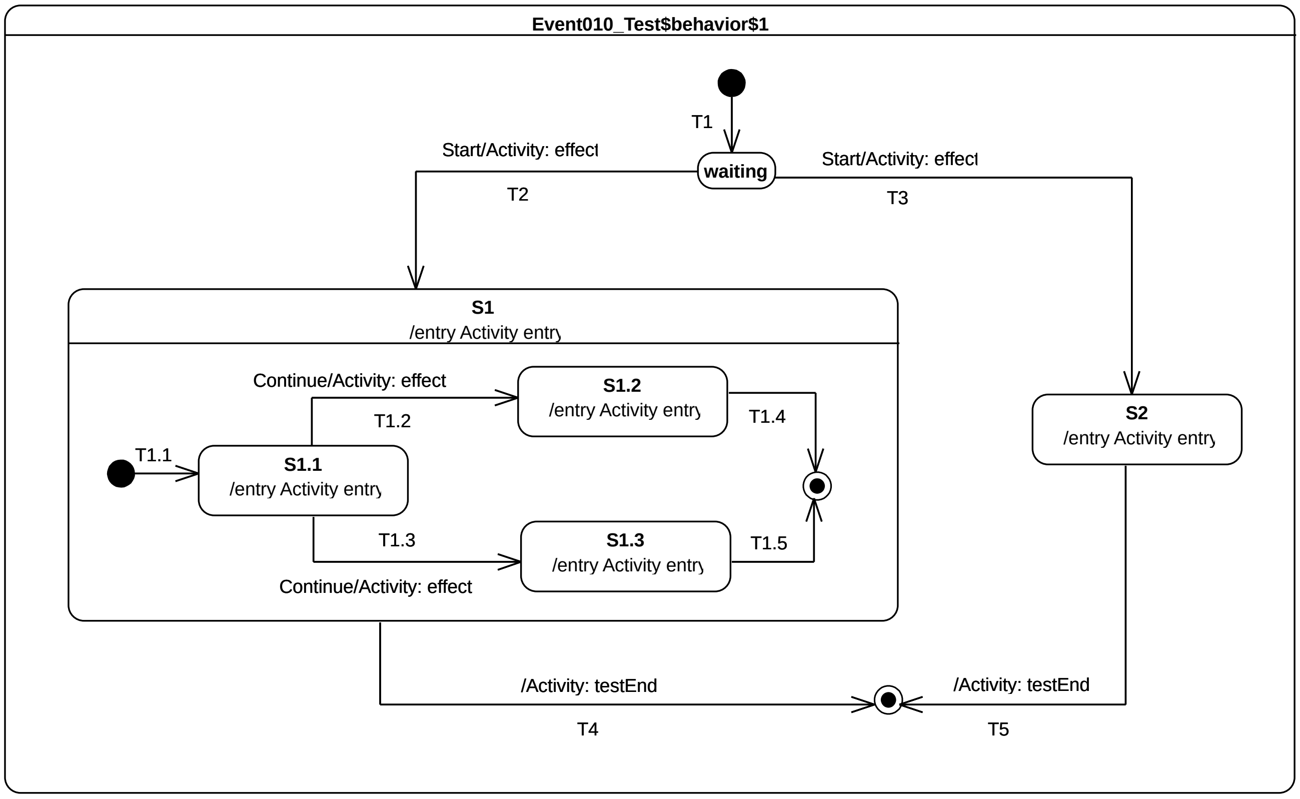


Figure 9.30 - Event 010 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *waiting*.
- Continue – received when in configuration *S1[S1.1]*.

Generated trace

- T2(effect)::S1(entry)::S1.1(entry)::T1.2(effect)::S1.2(entry)

Note. This test case highlights the resolving of transition conflicts at run time. The first conflict that is encountered happens when the event occurrence *Start* is accepted. Many transitions (i.e., *T2* and *T3*) originating from the same state (i.e., *waiting*) can be triggered using this same event occurrence. Only one of them is chosen, nondeterministically. In the case of the above trace, *T2* is chosen. A similar scenario happens when the state machine execution is in configuration *S1[S1.1]*. In the case of the above trace, *T1.2* is triggered by the *Continue* event occurrence.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(waiting)**]	[waiting]	[]
3	[**Start**]	[waiting]	[T2(T1.1)]
4	[Continue, **CE(S1.1)**]	[S1[S1.1]]	[]
5	[**Continue**]	[S1[S1.1]]	[T1.2]
6	[**CE(S1.2)**]	[S1[S1.2]]	[T1.4]
7	[**CE(S1)**]	[S1]	[T4]

Alternative execution trace

The presence of conflicting transitions in this test implies the possibility to observe different executions traces. Indeed, for example, *T2* and *T3* are in conflict, but it is not possible to anticipate which one will be chosen to fire. Hence, the following alternative execution traces can be generated for that test case:

- T2(effect)::S1(entry)::S1.1(entry)::T1.3(effect)::S1.3(entry)
- T3(effect)::S2(entry)

9.3.4.7 Event 015

Tested state machine

The state machine that is executed for this test is presented in Figure 9.31.

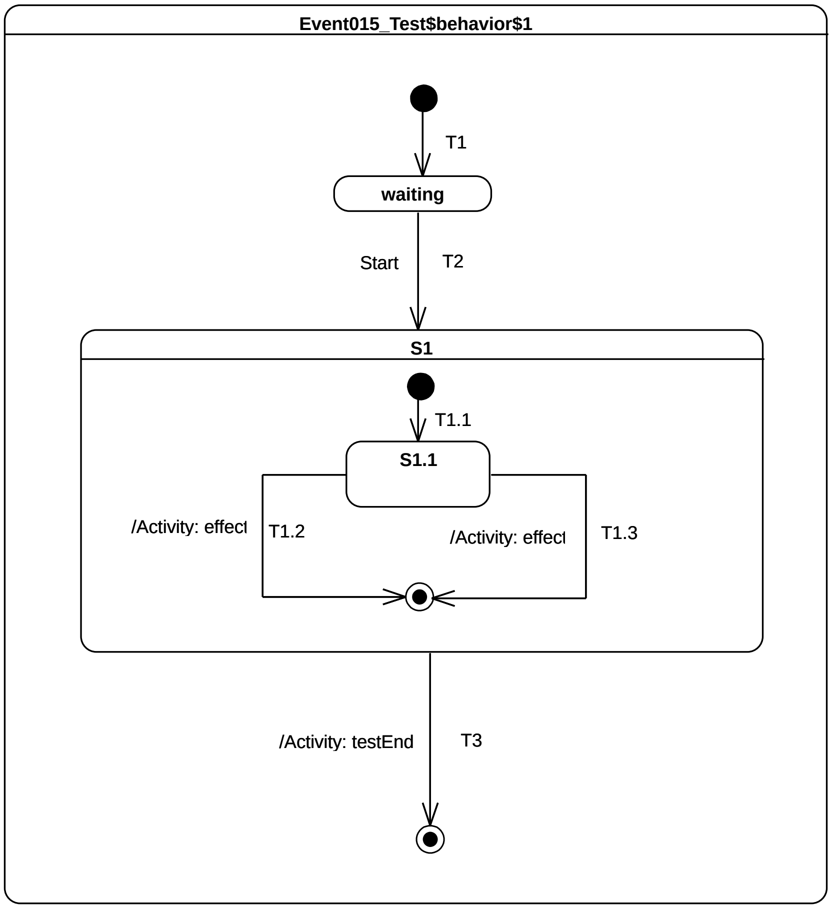


Figure 9.31 - Event 015 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *waiting*.

Generated trace

- T1.2(effect)

Note. The purpose of this test is to demonstrate conflict resolution in the case where more than one completion transition can be fired using the same completion event. This test case is similar to the one presented in 9.3.4.6. Nevertheless, it illustrates the situation in which the conflict occurs when a completion event occurrence for *S1.1* is accepted. The two completion transitions *T1.2* and *T1.3* can both be triggered using the completion event occurrence. Here again, the conflict is resolved nondeterministically.

RTC steps

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

Alternative execution traces

The presence of conflicting transitions in this test implies the possibility of an alternative execution trace. Indeed, *T1.2* and *T1.3* are in conflict, and it is not possible to anticipate which one will be the first chosen to fire.

- T1.3(effect)

9.3.4.8 Event 016-A

Tested state machine

The state machine that is executed for this test is presented in Figure 9.32.

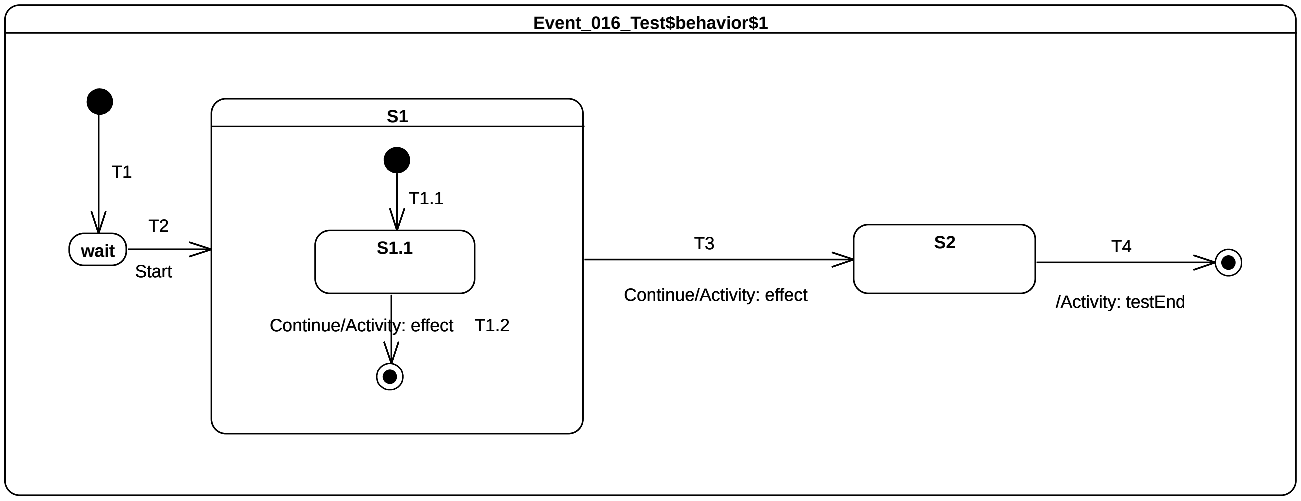

Event_016_Test$behavior$1
T1
wait
T2
Start
S1
T1.1
S1.1
Continue/Activity: effect
T1.2
T3
Continue/Activity: effect
S2
T4
/Activity: testEnd

Figure 9.32 - Event 016 – A Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.

- Continue – received when in configuration *S1[S1.1]*.
- Continue – received when in configuration *S1*.

Generated trace

- T1.2(effect)::T3(effect)

Note. The purpose of this test is to demonstrate that, if multiple transitions are available to fire, but cannot be fired concurrently, then the transition selected to fire is the one with the highest priority. . When the RTC step initiated by accepting the *Continue* event occurrence starts, the state machine is in configuration *S1[S1.1]*. At this point, two transitions can be triggered by the same event occurrence. The resolution of this potential conflict is realized by analyzing transition priorities. Since *S1.1* is the innermost state in the configuration, transitions originating from this state will have the highest priority. Hence *T1.2* is triggered.

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)]
4	[Continue, **CE(S1.1)**]	[S1[S1.1]]	[]
5	[**Continue**]	[S1[S1.1]]	[T1.2]
6	[Continue, **CE(S1)**]	[S1]	[]
7	[**Continue**]	[S1]	[T3]
8	[**CE(S2)**]	[S2]	[T4]

9.3.4.9 Event 016-B

Tested state machine

The state machine that is executed for this test is presented in Figure 9.33.

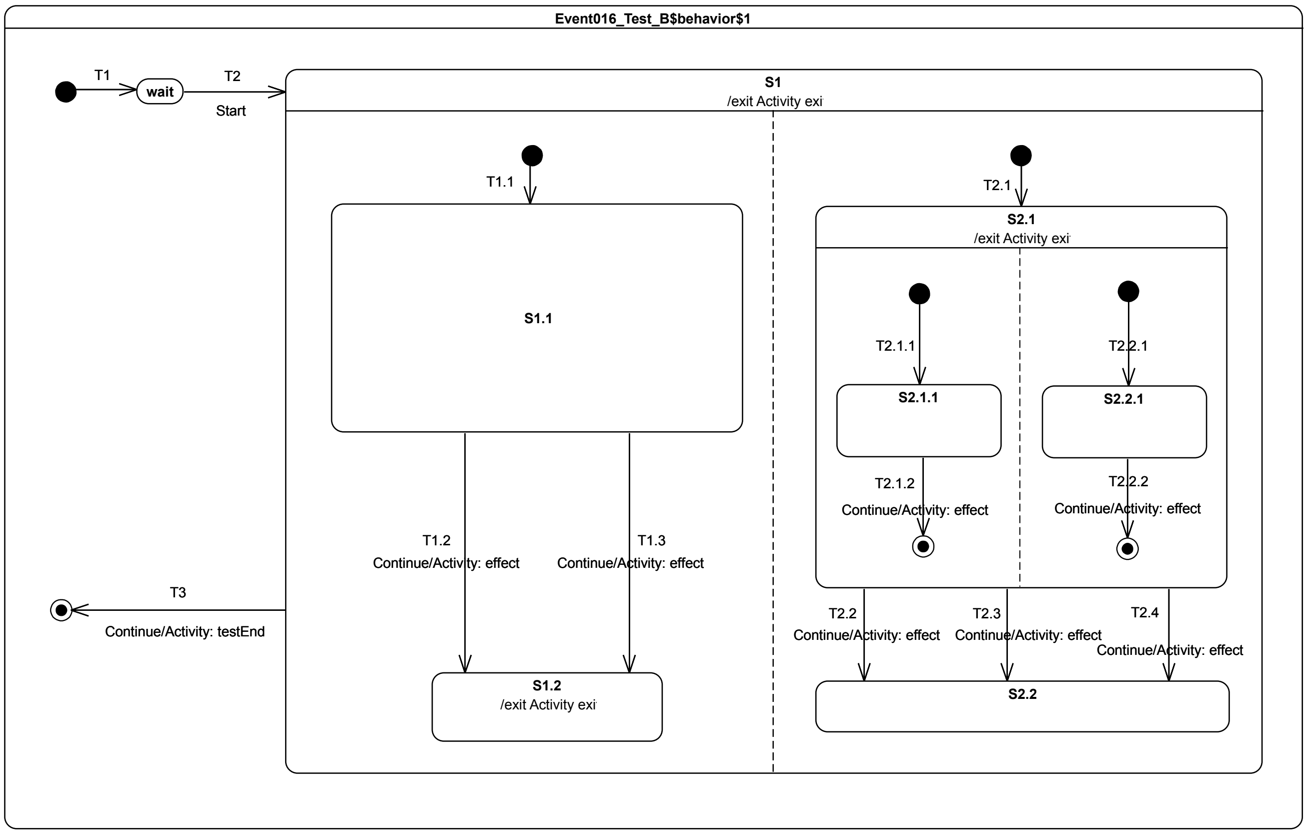

Event016_Test_B$behavior$1
T1
wait
T2
Start
S1
/exit Activity exi
T1.1
S1.1
T1.2
Continue/Activity: effect
T1.3
Continue/Activity: effect
S1.2
/exit Activity exi
T3
Continue/Activity: testEnd
T2.1
S2.1
/exit Activity exi
T2.1.1
S2.1.1
T2.2.1
S2.2.1
T2.1.2
Continue/Activity: effect
T2.2.2
Continue/Activity: effect
T2.2
Continue/Activity: effect
T2.3
Continue/Activity: effect
T2.4
Continue/Activity: effect
S2.2

Figure 9.33 - Event 016 – B Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1[S1.1, S2.1[S2.1.1, S2.2.1]]*.
- Continue – received when in configuration *S1[S1.1, S2.1]*.
- Continue – received when in configuration *S1[S1.2, S2.2]*.

Generated trace

- T1.2(effect)::T2.1.2(effect)::T2.2.2(effect)::S2.1(exit)::T2.2(effect)::S1.2(exit)::S1(exit)

Note. In this test case, hierarchical states, conflicting transitions and orthogonal regions are combined. The purpose is to demonstrate support for conflict resolution, support for transitions priority and firing of multiple transitions on a single execution step. The execution proceeds as follows. The RTC step initiated by the acceptance of the *Start* event occurrence brings the state machine to the configuration *S1[S1.1, S2.1[S2.1.1, S2.2.1]]]*. Completions event occurrencess generated during this RTC step (respectively for states *S1.1*, *S2.1.1* and *S2.2.1*) do not trigger any transition when dispatched.

The next RTC step that actually leads to transition triggering is the one initiated by the acceptance of the *Continue* event occurrence. During this step, *T2.1.2*, *T2.2.2* and *T1.2* are fired using the same event occurrence. This is made possible since all of these transitions are located in different regions and have a trigger for the same event. Note that instead of *T1.2*, the transition *T1.3* could have been fired. This is true since both transitions were in conflict (they leave the same state and have a trigger for the same event type) at the time of the step. Such conflicts are resolved nondeterministically. In the case of the above trace, at the end of the step, the state machine reaches the configuration *S1[S1.2, S2.1]*.

Completion event occurrences generated for states *S1.2* and *S2.1* do not trigger any transition when dispatched. However, when the next *Continue* event occurrence is dispatched, *T2.2* is fired. Here again there is a conflict between *T2.2*, *T2.3* and *T2.4* which is resolved non-deterministically. The last *Continue* event fires *T3* and leads to the completion of the state-machine execution.

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, T2.1(T2.1.1, T2.2.1))]
4	[Continue, CE(2.2.1), CE(S2.1.1), **CE(S1.1)**]	[S1[S1.1, S2.1[S2.1.1, S2.2.1]]]	[]
5	[Continue, CE(2.2.1), **CE(S2.1.1)**]	[S1[S1.1, S2.1[S2.1.1, S2.2.1]]]	[]
6	[Continue, **CE(2.2.1)**]	[S1[S1.1, S2.1[S2.1.1, S2.2.1]]]	[]
7	[**Continue**]	[S1[S1.1, S2.1[S2.1.1, S2.2.1]]]	[T1.2, T2.1.2, T2.2.2]
8	[Continue, CE(S2.1), **CE(S1.2)**]	[S1[S1.2, S2.1]]	[]
9	[Continue, **CE(S2.1)**]	[S1[S1.2, S2.1]]	[]
10	[**Continue**]	[S1[S1.2, S2.1]]	[T2.2]
11	[Continue, **CE(S2.2)**]	[S1[S1.2, S2.2]]	[]
12	[Continue]	[S1[S1.2, S2.2]]	[T3]

Alternative execution traces

The test case specifies concurrency with the orthogonal regions owned by *S1* and *S2.1*. This implies that when the test case is executed it can produce execution traces that are different from the one initially described, while remaining correct regarding the semantics specified for UML state machines. These alternative execution traces are presented below:

1. T1.3(effect)::T2.1.2(effect)::T2.2.2(effect)::S2.1(exit)::T2.2(effect)::S1.2(exit)::S1(exit)
2. T1.3(effect)::T2.1.2(effect)::T2.2.2(effect)::S2.1(exit)::T2.3(effect)::S1.2(exit)::S1(exit)
3. T1.3(effect)::T2.1.2(effect)::T2.2.2(effect)::S2.1(exit)::T2.4(effect)::S1.2(exit)::S1(exit)
4. T1.3(effect)::T2.2.2(effect)::T2.1.2(effect)::S2.1(exit)::T2.2(effect)::S1.2(exit)::S1(exit)

5. T1.3(effect)::T2.2.2(effect)::T2.1.2(effect)::S2.1(exit)::T2.3(effect)::S1.2(exit)::S1(exit)
6. T1.3(effect)::T2.2.2(effect)::T2.1.2(effect)::S2.1(exit)::T2.4(effect)::S1.2(exit)::S1(exit)
7. T2.1.2(effect)::T1.3(effect)::T2.2.2(effect)::S2.1(exit)::T2.2(effect)::S1.2(exit)::S1(exit)
8. T2.1.2(effect)::T1.3(effect)::T2.2.2(effect)::S2.1(exit)::T2.3(effect)::S1.2(exit)::S1(exit)
9. T2.1.2(effect)::T1.3(effect)::T2.2.2(effect)::S2.1(exit)::T2.4(effect)::S1.2(exit)::S1(exit)
10. T2.1.2(effect)::T2.2.2(effect)::T1.3(effect)::S2.1(exit)::T2.2(effect)::S1.2(exit)::S1(exit)
11. T2.1.2(effect)::T2.2.2(effect)::T1.3(effect)::S2.1(exit)::T2.3(effect)::S1.2(exit)::S1(exit)
12. T2.1.2(effect)::T2.2.2(effect)::T1.3(effect)::S2.1(exit)::T2.4(effect)::S1.2(exit)::S1(exit)
13. T2.2.2(effect)::T1.3(effect)::T2.1.2(effect)::S2.1(exit)::T2.2(effect)::S1.2(exit)::S1(exit)
14. T2.2.2(effect)::T1.3(effect)::T2.1.2(effect)::S2.1(exit)::T2.3(effect)::S1.2(exit)::S1(exit)
15. T2.2.2(effect)::T1.3(effect)::T2.1.2(effect)::S2.1(exit)::T2.4(effect)::S1.2(exit)::S1(exit)
16. T2.2.2(effect)::T2.1.2(effect)::T1.3(effect)::S2.1(exit)::T2.2(effect)::S1.2(exit)::S1(exit)
17. T2.2.2(effect)::T2.1.2(effect)::T1.3(effect)::S2.1(exit)::T2.3(effect)::S1.2(exit)::S1(exit)
18. T2.2.2(effect)::T2.1.2(effect)::T1.3(effect)::S2.1(exit)::T2.4(effect)::S1.2(exit)::S1(exit)
19. T1.2(effect)::T2.1.2(effect)::T2.2.2(effect)::S2.1(exit)::T2.3(effect)::S1.2(exit)::S1(exit)
20. T1.2(effect)::T2.1.2(effect)::T2.2.2(effect)::S2.1(exit)::T2.4(effect)::S1.2(exit)::S1(exit)
21. T1.2(effect)::T2.2.2(effect)::T2.1.2(effect)::S2.1(exit)::T2.2(effect)::S1.2(exit)::S1(exit)
22. T1.2(effect)::T2.2.2(effect)::T2.1.2(effect)::S2.1(exit)::T2.3(effect)::S1.2(exit)::S1(exit)
23. T1.2(effect)::T2.2.2(effect)::T2.1.2(effect)::S2.1(exit)::T2.4(effect)::S1.2(exit)::S1(exit)
24. T2.1.2(effect)::T1.2(effect)::T2.2.2(effect)::S2.1(exit)::T2.2(effect)::S1.2(exit)::S1(exit)
25. T2.1.2(effect)::T1.2(effect)::T2.2.2(effect)::S2.1(exit)::T2.3(effect)::S1.2(exit)::S1(exit)
26. T2.1.2(effect)::T1.2(effect)::T2.2.2(effect)::S2.1(exit)::T2.4(effect)::S1.2(exit)::S1(exit)
27. T2.1.2(effect)::T2.2.2(effect)::T1.2(effect)::S2.1(exit)::T2.2(effect)::S1.2(exit)::S1(exit)
28. T2.1.2(effect)::T2.2.2(effect)::T1.2(effect)::S2.1(exit)::T2.3(effect)::S1.2(exit)::S1(exit)
29. T2.1.2(effect)::T2.2.2(effect)::T1.2(effect)::S2.1(exit)::T2.4(effect)::S1.2(exit)::S1(exit)
30. T2.2.2(effect)::T1.2(effect)::T2.1.2(effect)::S2.1(exit)::T2.2(effect)::S1.2(exit)::S1(exit)
31. T2.2.2(effect)::T1.2(effect)::T2.1.2(effect)::S2.1(exit)::T2.3(effect)::S1.2(exit)::S1(exit)
32. T2.2.2(effect)::T1.2(effect)::T2.1.2(effect)::S2.1(exit)::T2.4(effect)::S1.2(exit)::S1(exit)

33. T2.2.2(effect)::T2.1.2(effect)::T1.2(effect)::S2.1(exit)::T2.2(effect)::S1.2(exit)::S1(exit)
34. T2.2.2(effect)::T2.1.2(effect)::T1.2(effect)::S2.1(exit)::T2.3(effect)::S1.2(exit)::S1(exit)
35. T2.2.2(effect)::T2.1.2(effect)::T1.2(effect)::S2.1(exit)::T2.4(effect)::S1.2(exit)::S1(exit)

Alternative trace 33 describes an execution where *T2.2.2* effect behavior is executed first and followed by the ones of *T2.1.2* and *T1.2*. The RTC steps that leads to the production of this trace are described in the table below.

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, T2.1(T2.1.1, T2.2.1))]
4	[Continue, CE(2.2.1), CE(S2.1.1), **CE(S1.1)**]	[S1[S1.1, S2.1[S2.1.1, S2.2.1]]]	[]
5	[Continue, CE(2.2.1), **CE(S2.1.1)**]	[S1[S1.1, S2.1[S2.1.1, S2.2.1]]]	[]
6	[Continue, **CE(2.2.1)**]	[S1[S1.1, S2.1[S2.1.1, S2.2.1]]]	[]
7	[**Continue**]	[S1[S1.1, S2.1[S2.1.1, S2.2.1]]]	[T2.2.2, T2.1.2, T1.2]
8	[CE(S1.2), **CE(S2.1)**]	[S1[S1.2, S2.1]	[]
9	[Continue, **CE(S1.2**)]	[S1[S1.2, S2.1]]	[]
10	[**Continue**]	[S1[S2.2]]	[T2.2]
11	[Continue**, CE(S2.2)**]	[S1[S2.2]]	[]
12	[**Continue**]	[S1[S2.2]]	[T3]

9.3.4.10 Event 017-A

Tested state machine

The state machine that is executed for this test is presented in Figure 9.34.

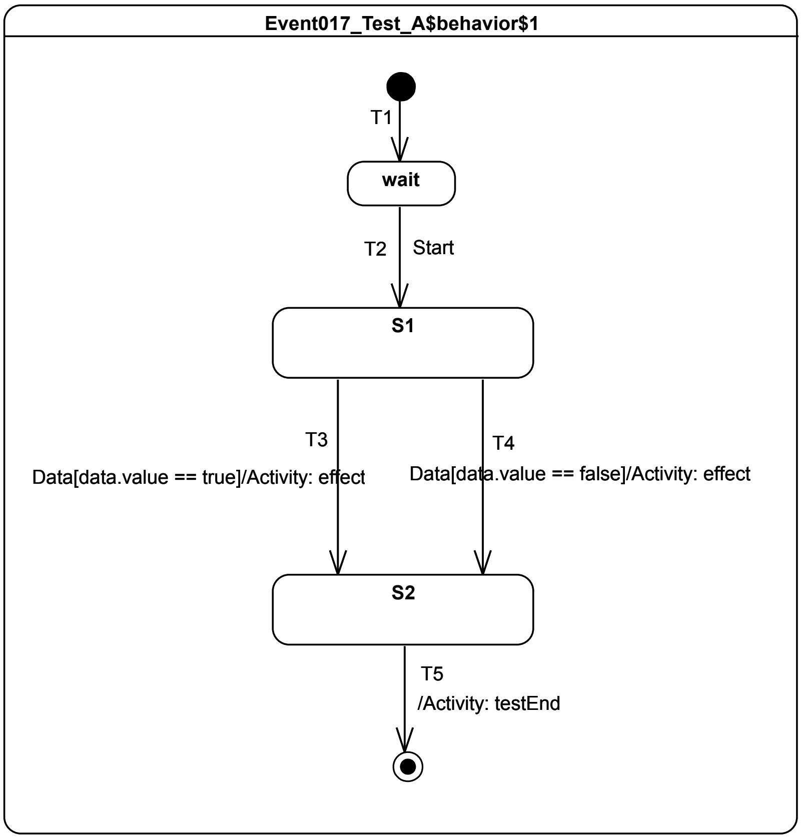

Event017_Test_A$behavior$1
T1
wait
T2
Start
S1
T3
Data[data.value == true]/Activity: effect
T4
Data[data.value == false]/Activity: effect
S2
T5
/Activity: testEnd

Figure 9.34 - Event 017 - A Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Data(true) – received when in configuration *S1*.

Generated trace

- T3(effect)

  Note. The purpose of this test is to check that during a RTC step initiated by the dispatching of a signal event occurrence, guard specifications and effect behaviors that are executed can access to properties values of the signal. In that test case, when the state machine is in configuration *S1* and *Data(true)* is received, *T3* fires because its guard comparing the value of the property *value* to true evaluates to true. Hence the effect behavior is executed and the trace fragment *T3(effect)* is added. The execution of the state machine completes when the RTC step initiated by the dispatching of *S2* completion event occurrence ends.

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]
4	[Data(true), **CE(S1)**]	[S1]	[]
5	[**Data(true)**]	[S1]	[T3]
6	[**CE(S2)**]	[S2]	[T5]

9.3.4.11 Event 017-B

Tested state machine

The state machine that is executed for this test is presented in Figure 9.35.

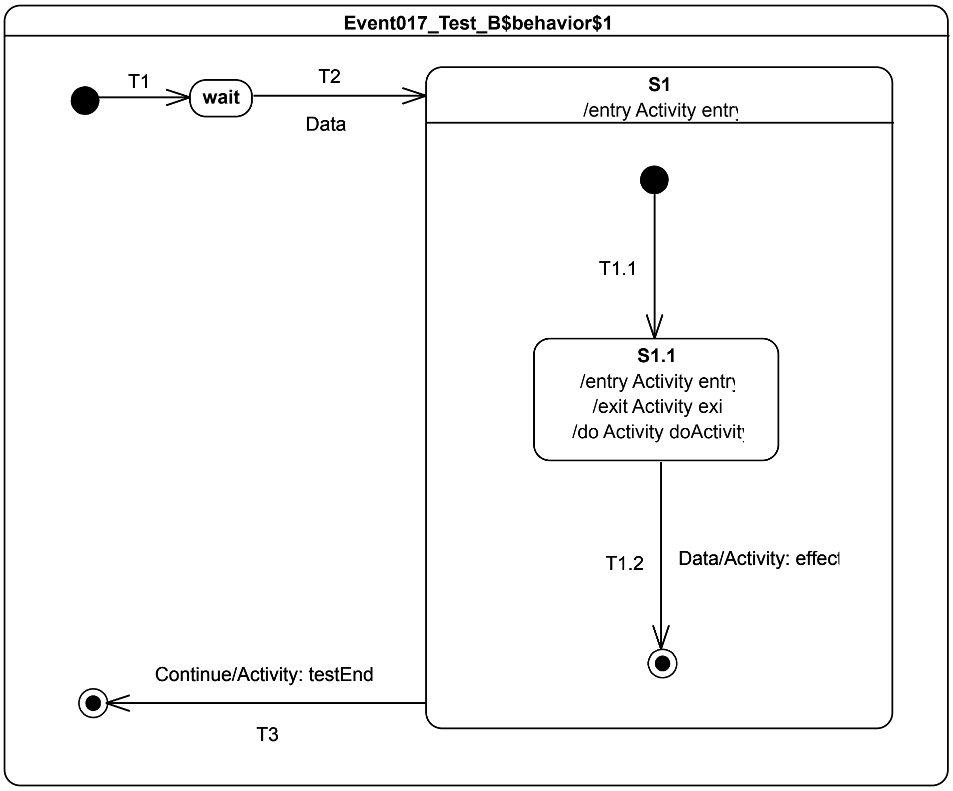


Figure 9.35 - Event 017 - B Test Classifier Behavior

Test executions

Received event occurrence(s)

- Data(true) – received when in configuration *wait*.
- Data(false) – received when in configuration *S1[S1.1]*.
- Continue – received when in configuration *S1*.

Generated trace

- S1(effect)[in=true]::S1.1(entry)[in=true]::S1.1(doActivity)[in=true]::S1.1(exit)[in=false]::T1.2(effect)[in=false]

Note. The purpose of this test is to check that state behaviors (i.e., entry, doActivity and exit behaviors) have access to the signal instance that initiated the RTC step in which they are executed. In the test case, when the state machine is in configuration *wait*, the dispatching of *Data(true)* implies the execution of the compound transition *T2(T1.1)*. Hence, during that step, the *S1* entry behavior and the *S1.1* entry behavior are executed, and the *S1.1* doActivity behavior is invoked. Considering that they are all executed, the following trace fragment is added to the trace: *S1(effect)[in=true]::S1.1(entry)[in=true]::S1.1(doActivity)[in=true]*. When the second occurrence of *Data(false)* is dispatched, *S1.1* is exited and *T1.2* is traversed. The execution of the S1.1 exit behavior and the *T2* effect behavior add the following fragment to the trace: *S1.1(exit)[in=false]::T1.2(effect)[in=false]*. The state-machine execution completes at the end of the RTC step in which *T3* is fired by the dispatching of the *Continue* event occurrence.

RTC steps

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

Alternative execution trace(s)

In this test case, *S1.1* has a doActivity behavior. The doActivity starts execution asynchronously from the state machine when the state *S1.1* is entered. Hence, the doActivity may or may not have the time to contribute to the trace before the state *S1.1* is exited. Considering this aspect, the following alternative execution trace can be generated:

- S1(effect)[in=true]::S1.1(entry)[in=true]::S1.1(exit)[in=false]::T1.2(effect)[in=false]

9.3.4.12 Event 018

Tested state machine

The state machine that is executed for this test is presented in Figure 9.36.

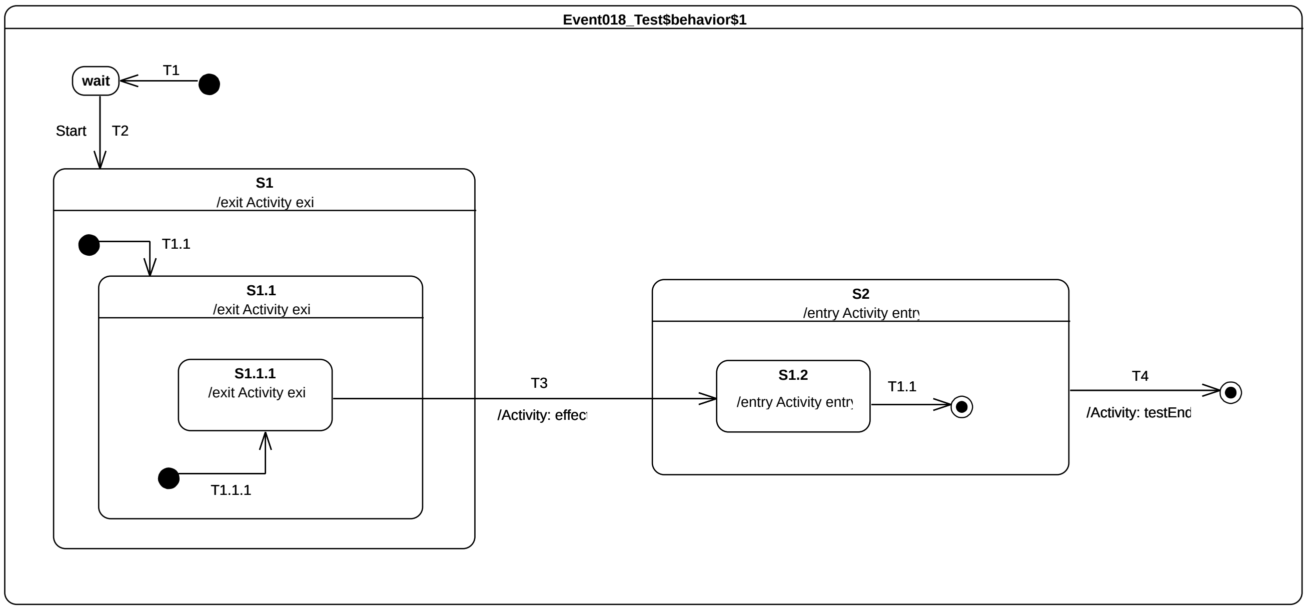


Figure 9.36 - Event 018 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- S1.1.1(exit)::S1.1(exit)::S1(exit)::T3(effect)::S2(entry)::S1.2(entry)

Note. The focus of this test case is to cover the situation in which transition *T3* is triggered by the acceptance of the completion event generated for *S1.1.1*. It shows support for exiting explicitly a nested state and entering explicitly a nested state. The exiting sequence implies that parent states are exited after the source state was exited. The entering sequence implies that parent states are entered before the target state is entered. Consider the situation where the state machine is in configuration. *S1[S1.1[S1.1.1]]*. When *T3* is traversed, it results in the following set of actions:

1. *S1.1.1, S1.1* and *S1* are exited (in this order).
2. The effect behavior of the transition is executed.
3. *S2* and *S2.1* are entered (in this order).

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))]
4	[**CE(S1.1.1)**]	[S1[S1.1[S1.1.1]]]	[T3]
5	[**CE(S1.2)**]	[S2[S1.2]]	[T1.1]
6	[**CE(S2)**]	[S2]	[T4]

9.3.4.13 Event 019-A

Tested state machine

The state machine that is executed for this test is presented in Figure 9.37.

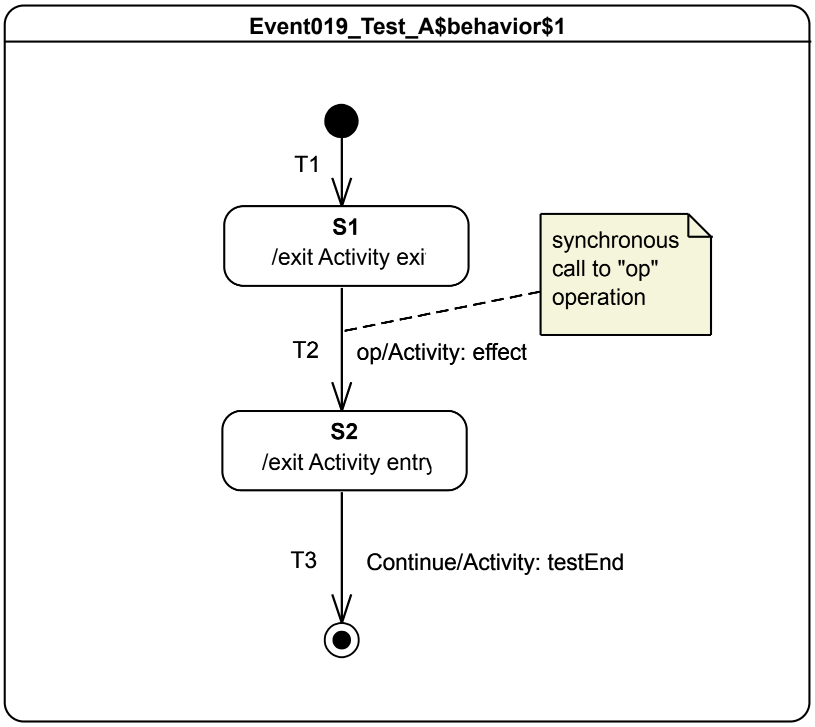

Event019_Test_A$behavior$1
T1
S1
/exit Activity exi
synchronous call to "op" operation
T2
op/Activity: effect
S2
/exit Activity entry
T3
Continue/Activity: testEnd

Figure 9.37 - Event 019 – A Test Classifier Behavior

Test executions

Received event occurrence(s)

- Call to operation *op()* – received when in configuration *S1*. This operation has no parameters.
- Continue – received when in configuration *S2*.

Generated trace

- S1(exit)::Call(op)::End::S2(entry)

Note. The purpose of this test is ensure that a call event is generated when an operation with no method is called on an active object and that this call event is dispatched in a RTC step. In the test case, the initial step brings the state machine to the configuration *S1*. When the call event is dispatched, the transition *T2,* which has a trigger for call event referencing the operation *op,* is fired. The computations corresponding to the realization of the operation are specified in the effect behavior, which is executed during the traversal of the transition. For this test, the computations are straightforward and lead to the inclusion of a new trace fragment: *Call(op)*. The state machine execution completes when the *Continue* event occurrence is received and triggers *T3,* which enables the state machine to reach the final state.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[T1]
2	[Call(op[]), **CE(S1)**]	[S1]	[]
3	[**Call(op[])**]	[S1]	[T2]
4	[Continue, **CE(S2)**]	[S2]	[]
5	[**Continue**]	[S2]	[T3]

9.3.4.14 Event 019-B

Tested state machine

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

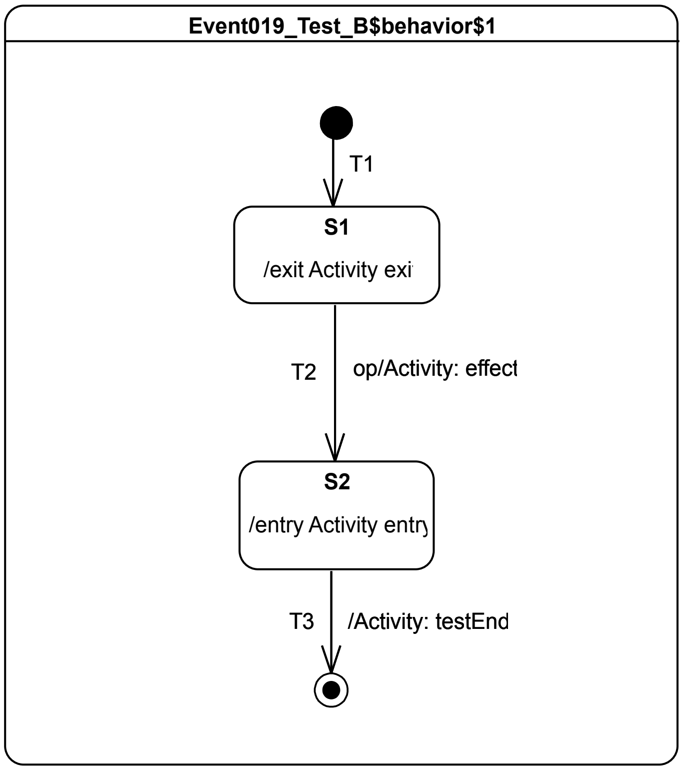


Figure 9.38 - Event 019 - B Test Classifier Behavior

Test executions

Received event occurrence(s)

- Call to operation *op(in p1: Integer, in p2: String)* is received when in configuration *S1*. Values associated to the parameters are (in order): 42 and "input".

Generated trace

- S1(exit)[in=42][in=input]::T2(effect)[in=42][in=input]::S2(entry)[in=42][in=input]

Note. The purpose of this test is to check that, during the dispatching of a call event for an operation with input parameters, behaviors have access to the input parameter values. In the test case, the initial step brings the state machine to the configuration *S1*. The step triggered by the call event implies the execution of the exit behavior of *S1*. This behavior has input parameters conforming to those of the operation. When executed, the behavior has access to the input parameter values 42 and "input" and adds the trace fragment: *S1(exit)[in=42][in=input]*. The *T2* effect behavior and *S2* entry behavior similarly have access to the input parameter values. The state machine execution completes when the completion event occurrence generated by *S2* triggers *T3,* whose traversal lead to the final state.

RTC steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Call(op), **CE(S1)**]	[S1]	[]
3	[**Call(op)**]	[S1]	[T2]
4	[**CE(S2)**]	[S2]	[T3]

9.3.4.15 Event 019-C

Tested state machine

The state machine that is executed for this test is presented in Figure 9.39.

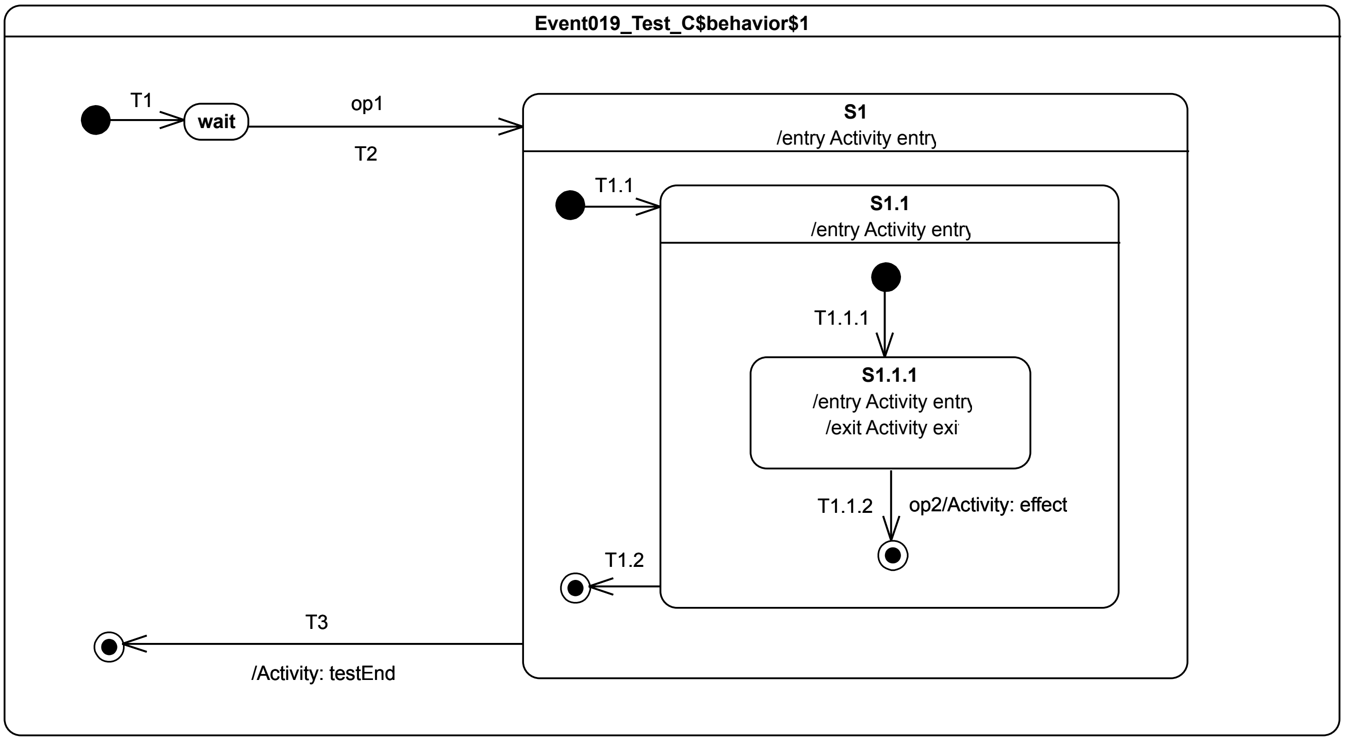


Figure 9.39 - Event 019 - C Test Classifier Behavior

Test executions

Received event occurrence(s)

- Call to operation *op1(in p1: Integer, in p2: Integer)* – received when in configuration *wait*. Values associated to parameters are: 42 and “input”.

- Call to *operation op2(in p1: Boolean)* – received when in configuration *S1[S1.1[S1.1.1]]*. Value associated to parameter *p1* is: true.

Generated trace

- S1(entry)[in=42][in=input]::S1.1(entry)[in=42][in=input]::S1.1.1(entry)[in=42][in=input]::S1.1.1(exit)[in=true]::T1.1.2(effect)[in=true]

Note. The purpose of this test is to check that input parameter values of an operation are made available to behaviors executed during a step, even if they are located at different level of nesting. In the test case, the execution proceeds as follows. When the state machine is in the configuration *wait,* the call event for operation *op1* is received. The step initiated by the dispatching of *op1* implies the execution of the compound transition *T2[T1.1[T1.1.1]]*. During the step, the *S1* entry behavior, *S1.1* entry behavior and *S1.1.1* entry behavior are executed. Their executions add the following trace fragments: *S1(entry)[in=42][in=input], S1.1(entry)[in=42][in=input]* and *S1.1.1(entry)[in=42][in=input]*.The second call event that is dispatched triggers the transition *T1.1.2*, which implies the execution of the *S1.1.1* exit behavior and *T1.1.2* effect behavior. They add the following fragments to the trace: *S1.1.1(exit)[in=true] and T1.1.2(effect)[in=true].*

RTC steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Call(op1[42, input]), **CE(wait)**]	[wait]	[]
3	[**Call(op1[42, input])**]	[wait]	[T2(T1.1(T1.1.1))]
4	[Call(op2[true]), **CE(S1.1.1)**]	[S1[S1.1[S1.1.1]]]	[]
5	[**Call(op2[true])**]	[S1[S1.1[S1.1.1]]]	[T1.1.2]
6	[**CE(S1.1)**]	[S1[S1.1]]	[T1.2]
7	[**CE(S1)**]	[S1]	[T3]

9.3.4.16 Event 019-D

Tested state machine

The state machine that is executed for this test is presented in Figure 9.40.

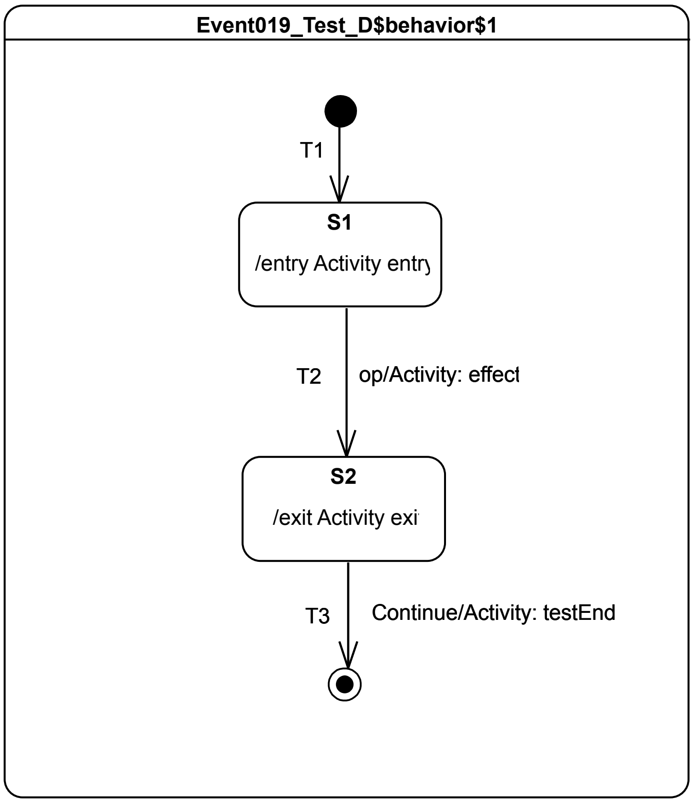


Figure 9.40 - Event 019 - D Test Classifier Behavior

Test executions

Received event occurrence(s)

- Call to operation *op():String* – received when in configuration *S1*.
- Continue – received when in configuration *S2*.

Generated trace

- S1(entry)::T2(effect)[out=output]::[out=output]::S2(exit)

Note. The purpose of this test is to check that, when an operation with output parameters is called on an active class, output values might be produced during the step of execution in which the call event is dispatched. In such a situation, these output values are made available as output values of the operation. In this test case, the call event corresponding to the call of the operation *op* is dispatched when the state-machine is in configuration *S1*. Hence the transition *T2* is fired and the effect behavior as well as the exit behavior of *S2* are executed. The values they produce are made available as operation outputs. However, in that context the exit behavior is the last behavior to be executed before the RTC completes. Consequently, the output value produced by this behavior is the output value of the operation.

RTC steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Call(op(), **CE(S1)**]	[S1]	[]
3	[**Call(op)**]	[S1]	[T2]
4	[Continue, **CE(S2)**]	[S2]	[]
5	[**Continue**]	[S2]	[T3]

9.3.4.17 Event 019-E

Tested state machine

The state machine that is executed for this test is presented in Figure 9.41.

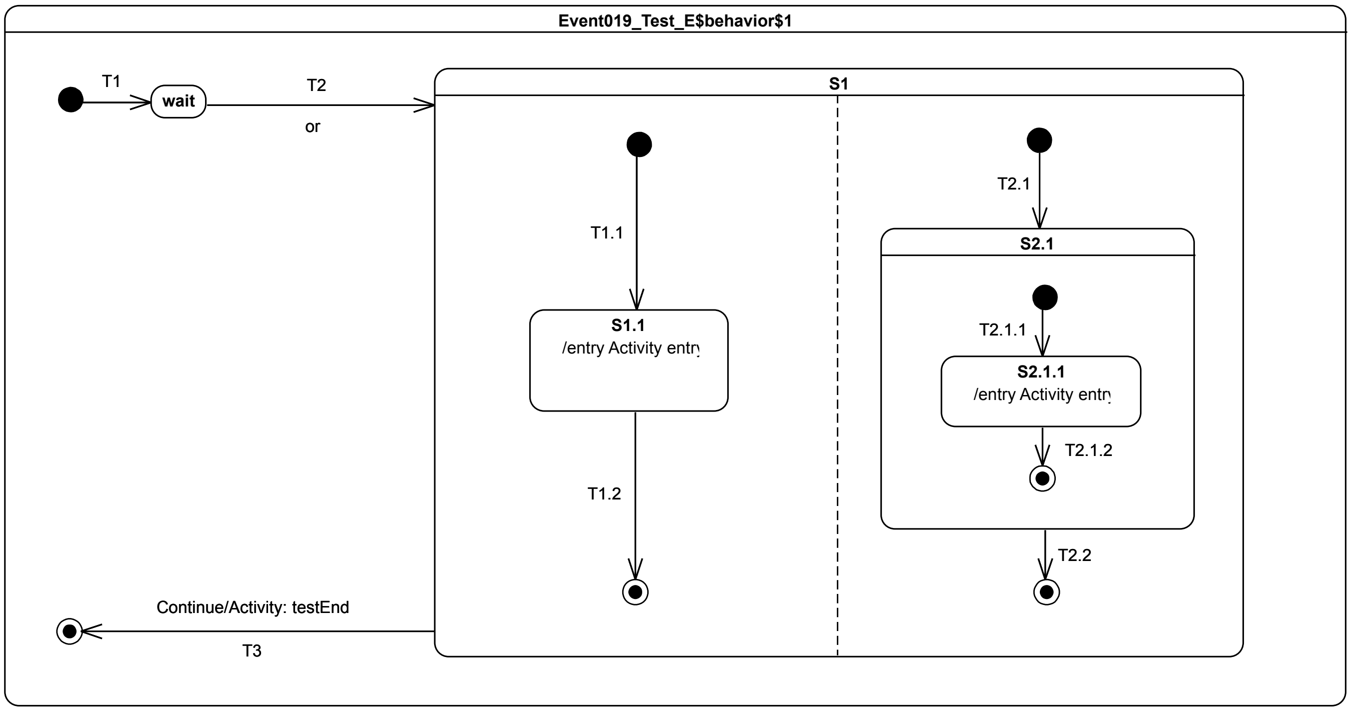


Figure 9.41 - Event 019 - E Test Classifier Behavior

Test executions

Received event occurrence(s)

- Call to operation *or(in left: Boolean, in right: Boolean, out result: Boolean): Boolean* – received when in configuration *wait*. Values associated to input parameters are true and true.

- Continue – received when in configuration *S1*.

Generated trace

- S1.1(entry)[in=true][in=true][out=true][out=true]::S2.1.1(entry)[in=true][in=true][out=false][out=false]::[out=false][out=false]

Note. The purpose of this test is to check that output values for the operation call *or* are also produced even if the executed behaviors are located at different levels of nesting. In addition, it shows that two possible output values can be produced depending on how the orthogonal regions of *S1* are interleaved at run time. In that case, the realization of the operation call is encoded in the entry behaviors that are executed during the step initiated by the call-event occurrence. The output values produced by the *S1.1* entry behavior and the *S2.1.1* entry behavior are different. Indeed, the *S1.1* behavior realizes the operation by performing a logical OR between the two inputs while *S2.1.1* behavior performs a logical XOR. Hence if the *S1.1* behavior is the last executed behavior and the two inputs have the value true, then the output values for the operation will be true. Conversely, if the *S2.1.1* behavior is the last executed behavior and the two inputs have the value true then the output values for the operation will be false.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[T1]
2	[Call(or(true, true), **CE(wait)**]	[wait]	[]
3	[**Call(or(true, true))**]	[wait]	[T2(T1.1, T2.1(T2.1.1))]
4	[CE(S2.1.1), **CE(S1.1)**]	[S1[S1.1, S2.1[S2.1.1]]]	[T1.2]
5	[**CE(S2.1.1)**]	[S1[S2.1[S2.1.1]]]	[T2.1.2]
6	[**CE(S2.1**)]	[S1[S2.1]]	[T2.2]
7	[Continue, **CE(S1)**]	[S1]	[]
8	[**Continue**]	[S1]	[T3]

Alternative execution traces

In the test case, there is concurrency specified in state S1. This means an alternative execution is possible for the state machine under test. This trace is described below and shows the case where S1.1 entry behavior is executed after the S2.1.1 entry behavior.

- S2.1.1(entry)[in=true][in=true][out=false][out=false]::S1.1(entry)[in=true][in=true][out=true][out=true]::[out=true][out=true]

9.3.5 Entering

9.3.5.1 Overview

Test cases presented in this subclause deal with entry semantics of composite states.

9.3.5.2 Entering 004

Tested state machine

The state machine that is executed for this test is presented in Figure 9.42.

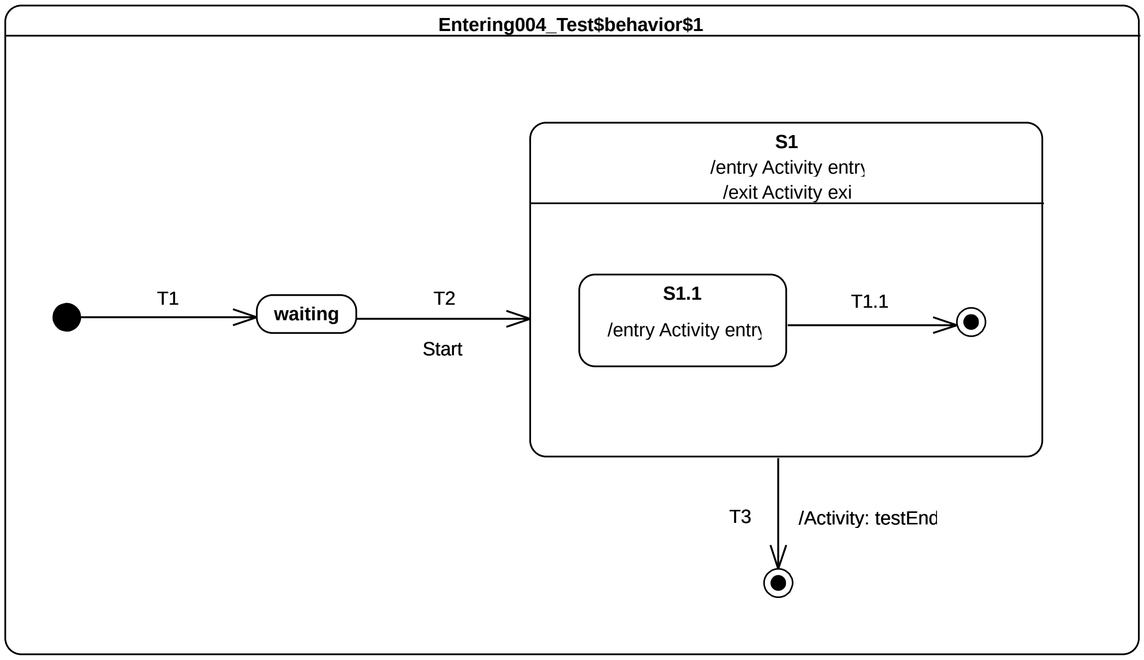


Figure 9.42 - Entering 004 Test Classifier behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *waiting*.

Generated trace

- S1(entry)::S1(exit)

Note. The purpose of this test is to demonstrate that, if a region has no initial pseudostate but is entered implicitly, then this region is ignored by the execution. The test case model intentionally omits an initial pseudostate and transition for the composite state *S1* – a situation that is syntactically valid but not recommended. Consequently, *S1* is treated as if it is a simple state upon the completion of transition *T2*. This means that, when *S1* is entered, it completes right after the termination of its entry behavior. This completion event is used to trigger the transition *T3*, which leads to the completion of the state machine execution.

RTC steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(waiting)**]	[waiting]	[]
3	[**Start**]	[waiting]	[T2]
4	[**CE(S1)**]	[S1]	[T3]

9.3.5.3 Entering 005

Tested state machine

The state machine that is executed for this test is presented in Figure 9.43.

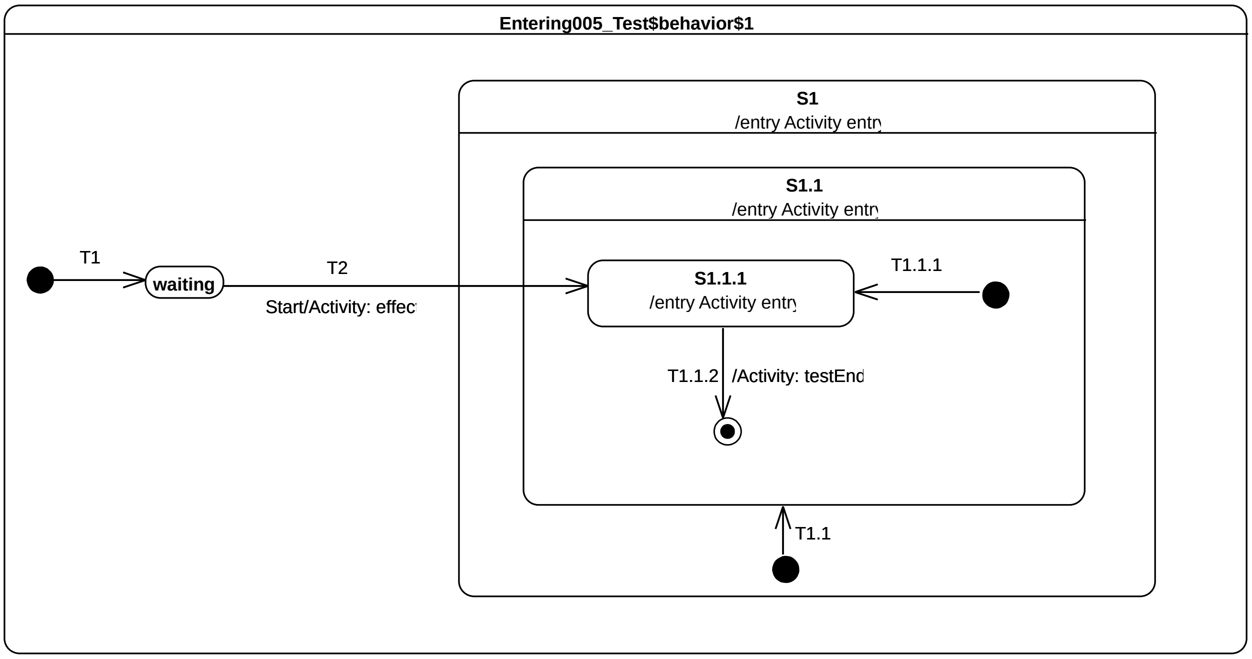


Figure 9.43 - Entering 005 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *waiting*.

Generated trace

- T2(effect)::S1(entry)::S1.1(entry)::S1.1.1(entry)

Note. The purpose of this test is to demonstrate that, when a nested state is entered directly while its parent states have not yet been entered, then the entry behavior of the entered state is always executed after the entry behavior of its parent and that rule applies recursively. This test case illustrates direct entry to the deeply nested state *S1.1.1*. In this situation, when *T2* is triggered, its effect behavior is executed and leads to the entering of *S1*, *S1.1*, and *S1.1.1* respectively. *S1* and *S1.1* are composite states whose unique region is entered directly (i.e., even if an initial pseudostate and transition exist, they will not be taken).

RTC steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(waiting)**]	[waiting]	[]
3	[**Start**]	[waiting]	[T2]
4	[**CE(S1.1.1)**]	[S1[S1.1[S1.1.1]]]	[T1.1.2]
5	[**CE(S1.1)**]	[S1[S1.1]]	[]

9.3.5.4 Entering 009

Tested state machine

The state machine that is executed for this test is presented in Figure 9.44.

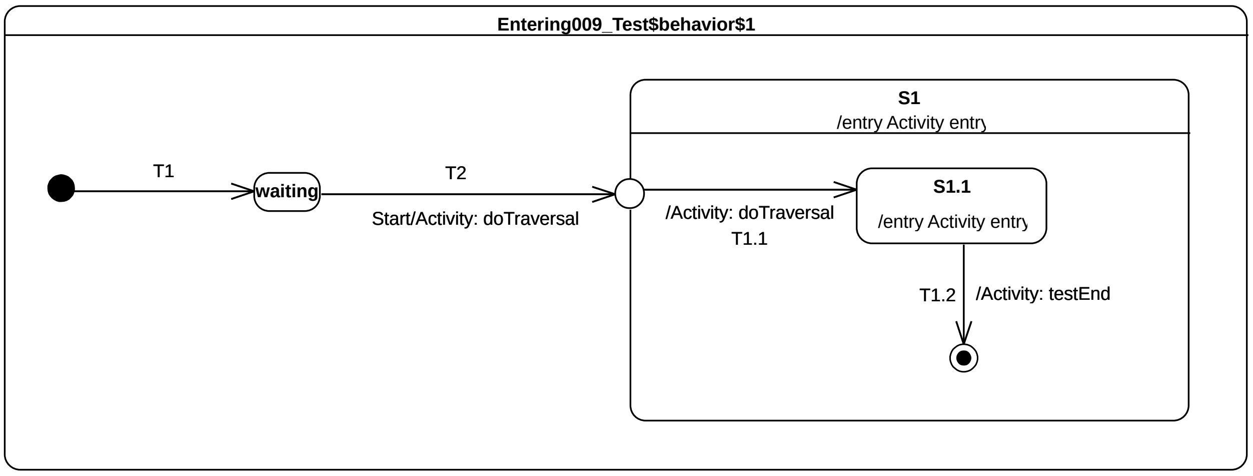


Figure 9.44 - Entering 009 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *waiting*.

Generated trace

- T2(effect)::S1(entry)::T1.1(effect)::S1.1(entry)

Note. This test case illustrates entering of a composite state through an entry point. It shows that the entry behavior of the composite state owning the entry point is executed before the effect behavior of the transition leaving the entry point. Consider the situation where the state machine is in configuration *waiting*. When the *Start* event is dispatched, *T2* is traversed and the entry point is reached. This leads to the entering of *S1* and the execution of its entry behavior. The region of *S1* is then entered immediately after the entry point is reached and transition *T1.1* is traversed. This leads to the entering of *S1.1*. The latter completes when its entry behavior has executed, after which the completion event is used to trigger the transition *T1.2*.

RTC steps

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

9.3.5.5 Entering 010

Tested state machine

The state machine that is executed for this test is presented in Figure 9.45.

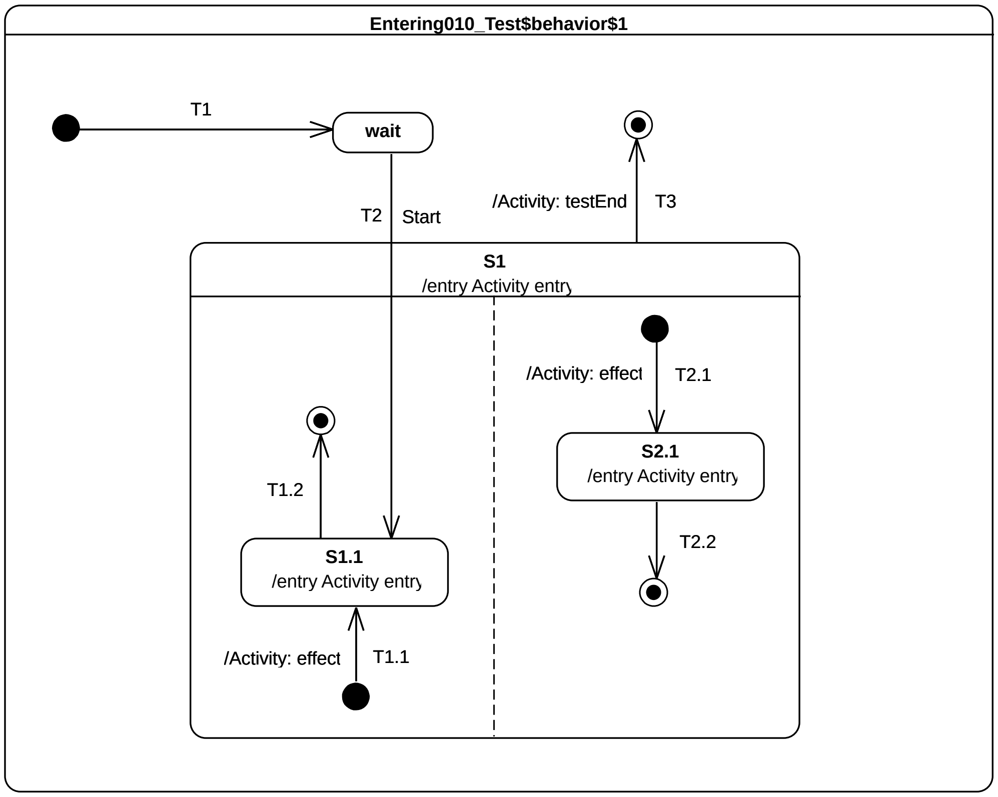


Figure 9.45 - Entering 010 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- S1(entry)::T2.1(effect)::S2.1(entry)::S1.1(entry)

Note. This test case presents the entry into a composite state with multiple regions. In this case, one region is entered directly whereas the other is entered by default. This occurs in the RTC step initiated by acceptance of the *Start* event occurrence. First *S1* is entered, which leads to the default entry of the right-hand side region. Next, the left hand side region is entered directly (i.e., the initial pseudo state and its outgoing transition are not traversed). This means that state *S1.1* is entered. Note that this describes one possible execution, since the concurrency implied by the orthogonal regions of *S1* can lead to other valid execution traces. The rules for default and direct entry of regions remain the same as stated in previous tests.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[waiting]	[T2(T1.1, T2.1)]
4	[CE(S1.1), **CE(2.1)**]	[S1[S1.1, S2.1]]	[T2.2]
5	[**CE(S1.1)**]	[S1[S1.1]]	[T1.2]
6	[**CE(S1)**]	[S1]	[T3]

Alternative execution traces(s)

In this test case, *S1* specifies concurrency due to its two orthogonal regions. Hence, there are also traces that might be generated by the test that are different from the one presented above. These traces are:

1. S1(entry)::S1.1(entry)::T2.1(effect)::S2.1(entry)
2. S1(entry)::T2.1(effect)::S1.1(entry)::S2.1(entry)

Consider trace 2. It shows that *S1.1* entry behavior is executed first, followed by the effect behavior of *T2.1* and finally the execution of *S2.1* entry behavior. The RTC steps lead to the production of this trace are described below:

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T2.1)]
4	[CE(S2.1), **CE(S1.1)**]	[S1[S1.1, S2.1]]	[T1.2]
5	[**CE(S2.1)**]	[S1[S2.1]]	[T2.2]
6	[**CE(S1)**]	[S1]	[T3]

9.3.5.6 Entering 011

Tested state machine

The state machine that is executed for this test is presented in Figure 9.46.

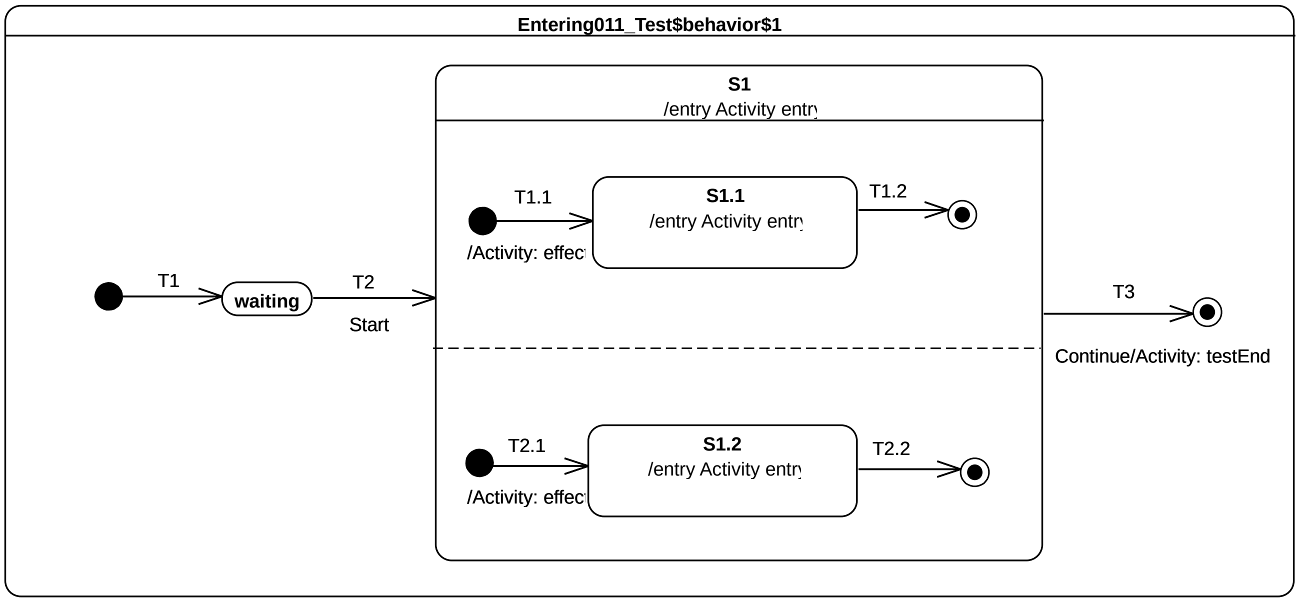


Figure 9.46 - Entering 011 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *waiting*.
- Continue – received when in configuration *S1*.

Generated trace

- S1(entry)::T2.1(effect)::S1.2(entry)::T1.1(effect)::S1.1(entry)

Note. This test case covers the situation where all regions of a composite state are entered using the default entry rule. The RTC step that is initiated by the acceptance of the *Start* event occurrence leads to the entering of *S1*, which means that, after its entry behavior is executed, all regions will be started concurrently. The execution of each region starts from its initial pseudostate. *S1* completes when both of its region have completed. This occurs when completion events generated by *S1.1* and *S1.2* have been dispatched. The *S1* completion event is then used to trigger transition *T3*.

RTC steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(waiting)**]	[waiting]	[]
3	[**Start**]	[waiting]	[T2(T1.1, T2.1)]
4	[CE(1.2), **CE(S1.1)**]	[S1[S1.1, S2.1]]	[T1.2]

5	[**CE(1.2)**]	[S1[S2.1]]	[T2.2]
6	[Continue, **CE(S1)**]	[S1]	[]
7	[**Continue**]	[S1]	[T3]

Alternative execution traces

In this test, concurrency is specified due to orthogonal regions owned by the composite state *S1*. The presence of these regions implies that different execution traces can be observed for the tested state machine while conforming to UML semantics. These execution traces are:

1. S1(entry)::T2.1(effect)::T1.1(effect)::S1.2(entry)::S1.1(entry)
2. S1(entry)::T2.1(effect)::T1.1(effect)::S1.1(entry)::S1.2(entry
3. S1(entry)::T1.1(effect)::T2.1(effect)::S1.1(entry)::S1.2(entry)
4. S1(entry)::T1.1(effect)::T2.1(effect)::S1.2(entry)::S1.1(entry)
5. S1(entry)::T1.1(effect)::S1.1(entry)::T2.1(effect)::S1.2(entry)

Consider trace 4. The RTC steps leading to the production of this trace are described in the table below:

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

9.3.6 Exiting

9.3.6.1 Overview

Tests presented in this subclause assess that semantics associated with state exiting rules conform to what is specified in UML.

9.3.6.2 Exiting 001

Tested state machine

The state machine that is executed for this test is presented in Figure 9.47.

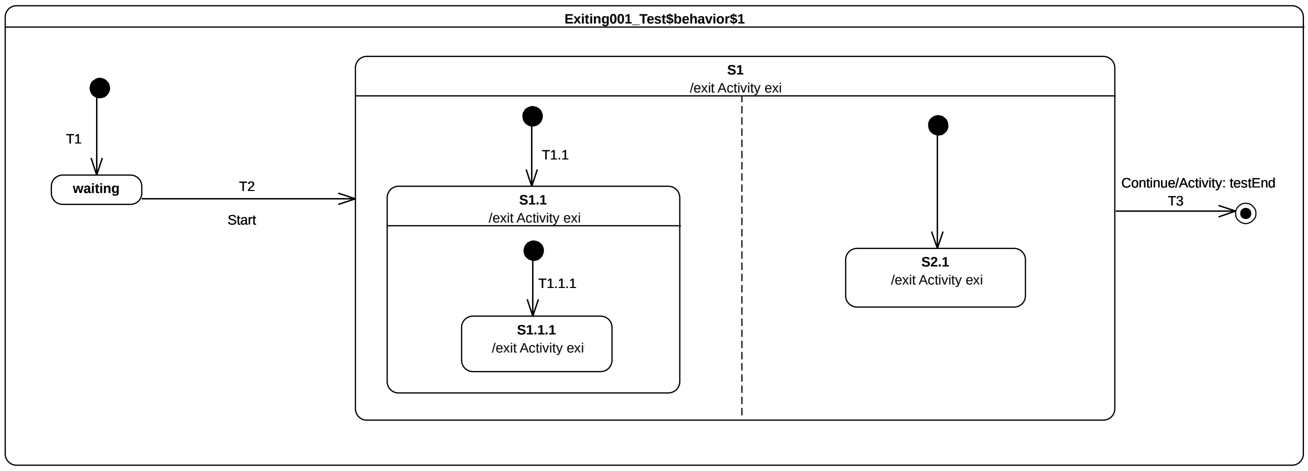


Figure 9.47 - Exiting 001 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *waiting*.
- Continue – received when in configuration *S1[S1.1[S1.1.1], S2.1]*.

Generated trace

- S1.1.1(exit)::S1.1(exit)::S2.1(exit)::S1(exit)

Note. This test illustrates the exit sequence of a composite state with orthogonal regions. It shows that the exit sequence starts with the innermost active state of each region and is propagated up to the source state (i.e., the composite state). Consider the situation where the state machine is in configuration *S1[S1.1[S1.1.1], S2.1]*. When the *Continue* event occurrence is accepted, it triggers transition *T3*. The first action encountered by the traversal of *T3* is the exiting of *S1*. This requires first that all active states in all regions controlled by this state are exited. The exit sequence starts for each region with the innermost active state. Hence, assuming that the left-hand side *S1.1.1* is exited first, it will be immediately followed by *S1.1*. Concurrently, *S2.1* is exited in the right-hand region. The exit sequence is concluded by the execution of the exit behavior of *S1*. Finally, the effect behavior of *T3* is executed and the state machine execution completes.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(waiting)**]	[waiting]	[]
3	[**Start**]	[waiting]	[T2(T1.1(T1.1.1), T2.1)]
4	[Continue, CE(S2.1), **CE(S1.1.1)**]	[S1[S1.1[S1.1.1], S2.1]]	[]
5	[Continue, CE(S2.1)]	[S1[S1.1[S1.1.1], S2.1]]	[]

6	[Continue]	[S1[S1.1[S1.1.1], S2.1]]	[T3]

Alternative execution traces

In this test, concurrency is specified due to orthogonal regions owned by the composite state *S1*. The presence of these regions implies that a different execution trace can be observed for the tested state machine while conforming to UML semantics. This execution trace is:

1. S1.1.1(exit)::S2.1(exit)::S1.1(exit)::S1(exit)
2. S2.1(exit)::S1.1.1(exit)::S1.1(exit)::S1(exit)

Consider trace 2. This trace is generated when the *Continue* signal event occurrence is dispatched, initiating the process of concurrently exiting the *S1* orthogonal regions. In this situation, the *S2.1* exit behavior is executed before *S1.1.1* exit behavior.

9.3.6.3 Exiting 002

Tested state machine

The state machine that is executed for this test is presented in Figure 9.8. The *doActivity* behavior of *S1* has exactly the same behavior as the one presented in Figure 9.37 except that, instead of waiting for a *Continue* event occurrence, it waits for an *AnotherSignal* event occurrence.

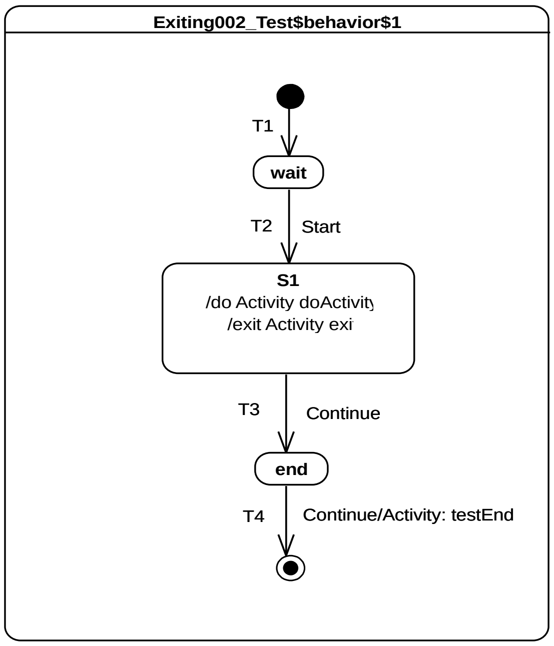


Figure 9.48 - Exiting 002 Test Classifier Behavior

The exit behavior is specified as shown in Figure 9.49. It contributes to the trace production by adding the trace fragment *S1(exit)* and sends a signal *AnotherSignal* to the current context object executing this behavior.

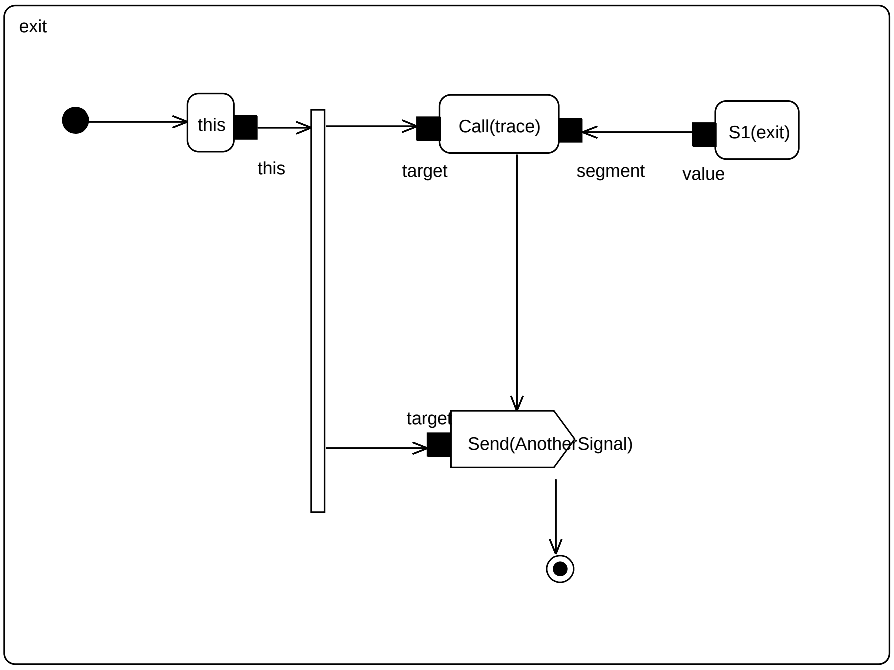


Figure 9.49 - S1 exit behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1*.
- AnotherSignal – received when in configuration *S1*.
- Continue – received when in configuration *end*.

Generated trace

- S1(doActivityPartI)::S1(exit)

Note. The purpose of this test is to demonstrate that the *doActivity* behavior (if it is still running) is aborted before the exit behavior is actually executed. In this test case, the *doActivity* behavior is started asynchronously after *S1* is entered. It is the very last action that takes place during the RTC step initiated by the acceptance of the *Start* event occurrence.

In this case, when the *Continue* event occurrence is dispatched, the *doActivity* behavior is still running. Indeed the latter waits for an *AnotherSignal* occurrence. However, *S1* is now forced to be exited using the transition *T3* (due to the acceptance of the *Continue* event occurrence). Hence, its *doActivity* behavior is aborted and its exit behavior is executed. To verify that the *doActivity* was effectively aborted before execution of the exit, the exit behavior of *S1* sends an *AnotherSignal* occurrence to the context object. If the *doActivity* was not aborted, then it would have used this event occurrence to continue its execution and it would have completed the execution trace with the message *S1(doActivityPartII)*.

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]
4	[AnotherSignal, **Continue**]	[S1]	[T3]
5	[Continue, AnotherSignal, **CE(end)**]	[end]	[]
6	[Continue, **AnotherSignal**]	[end]	[]
7	[**Continue**]	[end]	[T4]

9.3.6.4 Exiting 003

Tested state machine

The state machine that is executed for this test is presented in Figure 9.50.

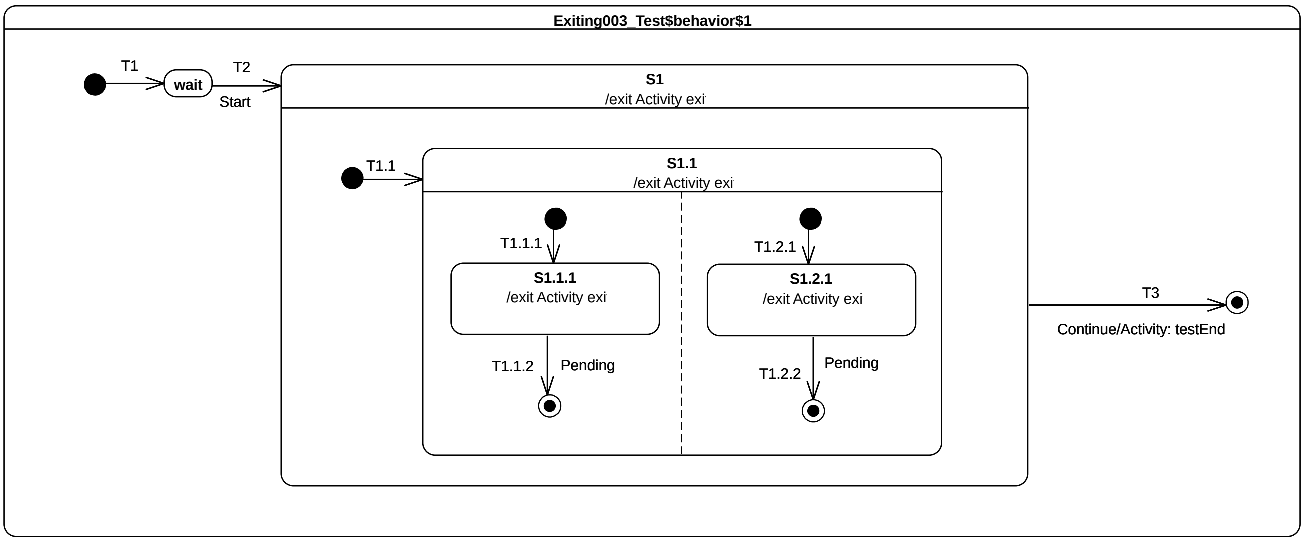


Figure 9.50 - Exiting 003 Test Classifier Behavior

Test executions

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 a composite state is exited, the exit behaviors that are executed first are those owned by the innermost active state(s). When the *Continue* event occurrence is accepted, the state machine is in configuration *S1[S1.1[S1.1.1, S1.2.1]]*. This means that to conform to UML state machine semantics, the exit sequence of S1 must start by first exiting *S1.1.1* and *S1.2.1*. Next, *S1.1* is exited and, finally, *S1* is exited.

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 traces

In this test, concurrency is specified due to orthogonal regions owned by the composite state *S1.1*. The presence of these regions implies that a different execution trace can be observed for the tested state machine while conforming to UML semantics. This alternative execution trace is:

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

In this case, when the *Continue* signal event occurrence is dispatched, *S1.2.1* exit behavior is executed before *S1.1.1* exit behavior.

9.3.6.5 Exiting 004

Tested state machine

The state machine that is executed for this test is presented in Figure 9.51.

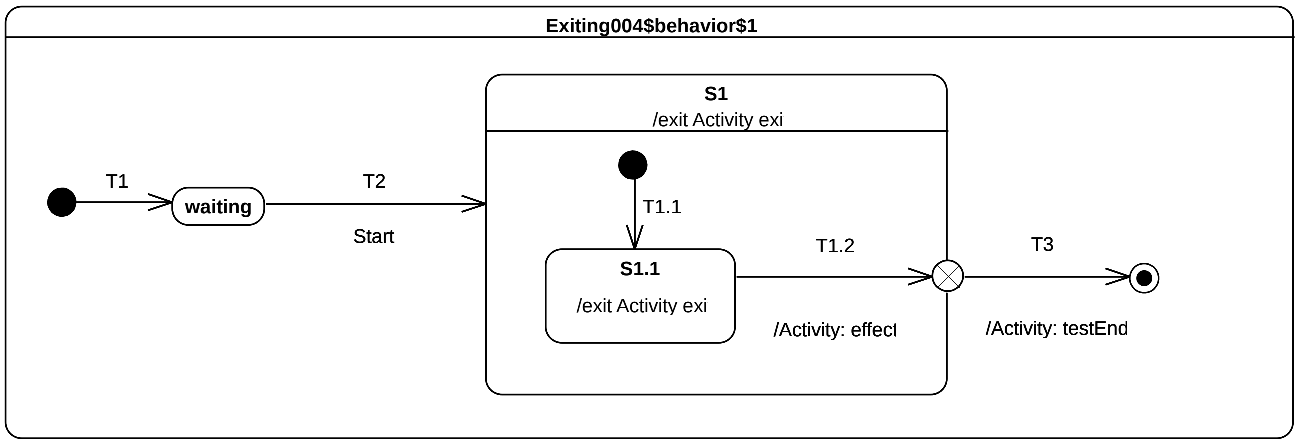


Figure 9.51 - Exiting 004 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- S1.1(exit)::T1.2(effect)::S1(exit)

Note. The purpose of this test is to validate that, when a composite state is exited using an exit point, then the effect behavior of the transition entering this pseudostate is executed before the exit behavior of the state. At the point where the state machine is in configuration *S1[S1.1],* the completion event generated for *S1.1* is dispatched and accepted. This initiates an RTC step during which *S1.1* is exited, the effect behavior is of *T1.2* is executed, *S1* is exited and finally the continuation transition *T3* is traversed.

RTC steps

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

9.3.6.6 Exiting 005

Tested state machine

The state machine that is executed for this test is presented in Figure 9.52.

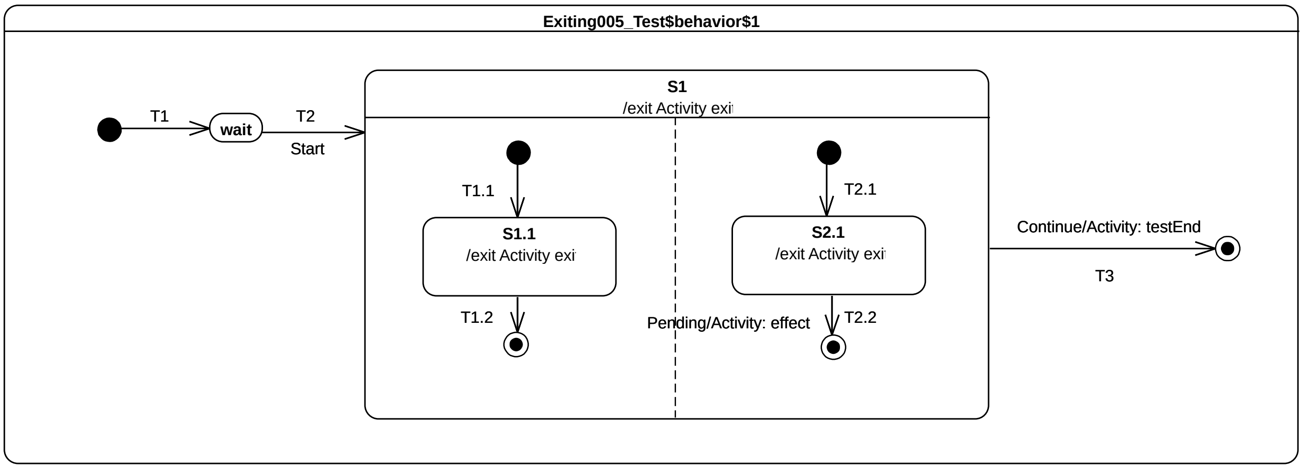


Figure 9.52 - Exiting 005 Test Classifier behavior

Test executions

Received event occurrence(s)

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

Generated trace

- S1.1(exit)::S2.1(exit)::S1(exit)

Note. The purpose of the test is to ensure that, when exiting a composite state with orthogonal regions, all regions that have not yet completed are exited and active states in these regions have their exit behaviors executed before the one owned by the composite state,.. When the Continue event occurrence is dispatched, the state machine is in the configuration *S1[S2.1]* (the left-hand region has already completed due to the acceptance of the *S1.1* completion event). Transition *T3* is triggered next, and the exit sequence starts with the execution of the *S2.1* exit behavior followed by the exit behavior of *S1*.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(waiting)**]	[waiting]	[]
3	[**Start**]	[waiting]	[T2(T1.1, T2.1)]
4	[Continue, CE(S2.1), **CE(S1.1)**]	[S1[S1.1, S2.1]]	[T1.2]
5	[Continue, **CE(S2.1)**]	[S1[S2.1]]	[]
6	[Continue]	[S1]	[T3]

9.3.7 Entry

9.3.7.1 Overview

Tests presented in this subclause assess that semantics associated with entry points conform to what is specified in UML.

9.3.7.2 Entry 002-A

Tested state machine

The state machine that is executed for this test is presented in Figure 9.53.

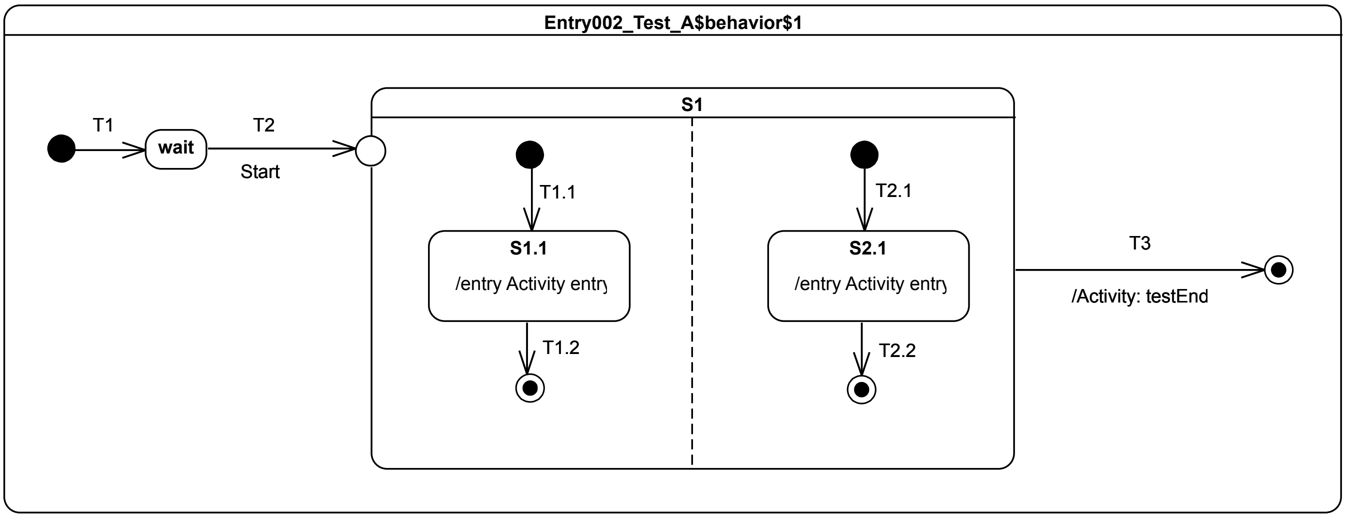


Figure 9.53 - Entry 002 - A Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- S1.1(entry)::S2.1(entry)

Note. The fact that the entry behavior of the state owning the entry point is executed before the effect behavior of the transition outgoing the exit point was demonstrated in the test case presented in 9.3.5.4. The purpose of the present test is to demonstrate that, if an entry point is placed on a composite state with orthogonal regions, then this entry point behaves as a fork. When the *Start* event occurrence is accepted by the state machine, *T2* is triggered. At the end of the *T2* traversal, the entry point is reached, which implies the entry of *S1*. Since there are no transitions originating from the pseudostate and penetrating into the state, all regions of *S1* are entered using the default entry rule (i.e, each region starts is execution using the its initial pseudostate). Hence, both transitions *T1.1* and *T2.1* are traversed resulting in states *S1.1* and *S2.1* executing their entry behaviors. At the end of each entry behavior execution, a completion event is generated. This is the end of the RTC step initiated by the dispatching of the *Start* event occurrence. The two following RTC steps are related to the dispatching and the acceptance of these completion events. As soon as both regions have completed, a completion event is generated for *S1*. The last RTC step consist of accepting this completion event to trigger *T3* thereby completing the state machine execution when the final state is reached.

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, T2.1)]
4	[CE(S2.1), **CE(S1.1)**]	[S1[S1.1, S2.1]]	[T1.2]
5	[**CE(S2.1)**]	[S1[S2.1]]	[T2.2]
6	[**CE(S1)**]	[S1]	[T3]

Alternative execution traces

In the test, there is concurrency specified in state *S1*. This means an alternative execution is possible for the state machine under test. This trace is described below and shows the case where the *S1.1* entry behavior is executed after the *S2.1* entry behavior.

- S2.1(entry)::S1.1(entry)

The RTC steps leading to the production of this trace are described in the table below.

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, T2.1)]
4	[CE(S1.1), **CE(S2.1)**]	[S1[S1.1, S2.1]]	[T2.2]
5	[**CE(S1.1)**]	[S1[S2.1]]	[T1.2]
6	[**CE(S1)**]	[S1]	[T3]

9.3.7.3 Entry 002-B

Tested state machine

The state machine that is executed for this test is presented in Figure 9.54.

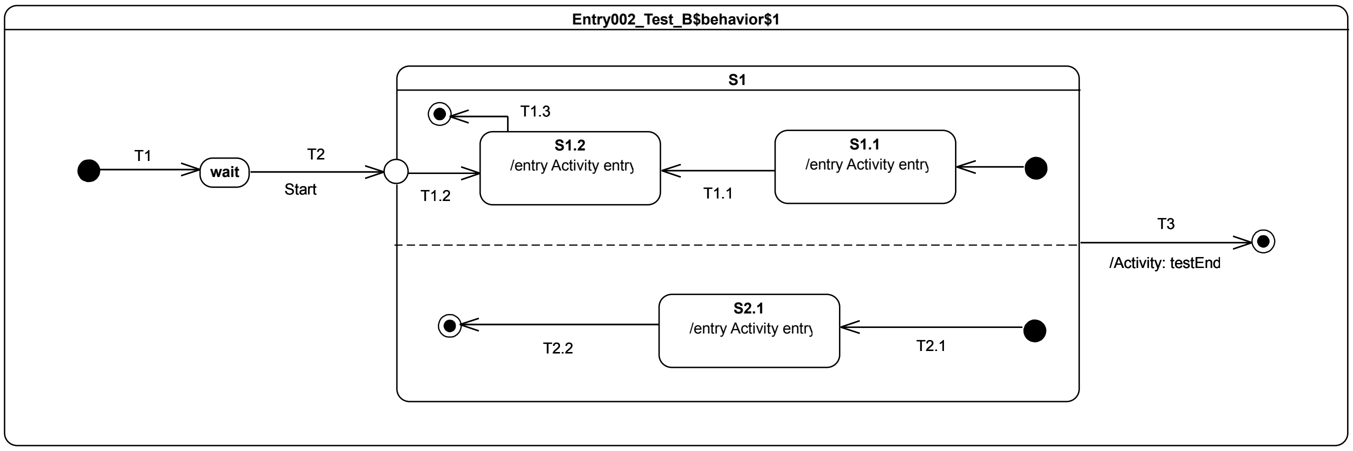


Figure 9.54 - Entry 002 - B Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- S2.1(entry)::S1.2(entry)

Note. The test case presented in 9.3.7.4 demonstrates that, when a transition outgoing from entry point and penetrating a composite state is traversed, then the region containing the targeted state is entered explicitly (i.e., without using the initial pseudo-state). The purpose of the present test is to demonstrate that, if orthogonal regions exist in that composite state, then these are entered using the default approach, whereas the first one (i.e., the one containing the target vertex) is entered explicitly. As we can see from the generated execution trace, the *S1.1(entry)* message does not appear, which indicates the entrance of the upper region of *S1* was realized explicitly. In addition, *S2.1*(*entry*) is part of the trace, indicating that the other region of *S1* was entered using the default entry approach.

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(T2.1, T1.2)]
4	[CE(S1.2), **CE(S2.1)**]	[S1[S1.2, S2.1]]	[T2.2]
5	[**CE(S1.2)**]	[S1[S1.2]]	[T1.3]
6	[**CE(S1)**]	[S1]	T3

Alternative execution traces

In the test, there is concurrency specified in state *S1*. This means an alternative execution is possible for the state machine under test. This trace is described below and shows the case where *S1.2* entry behavior is executed before the *S2.1* entry behavior.

- S1.2(entry)::S2.1(entry)

The RTC steps leading to the production of this trace are described in the table below.

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T1.2, T2.1)]
4	[CE(S2.1)**, CE(S1.2)**]	[S1[S1.2, S2.1]]	[T1.3]
5	[**CE(S2.1)**]	[S1[S1.2]]	[T2.2]
6	[**CE(S1)**]	[S1]	T3

9.3.7.4 Entry 002-C

Tested state machine

The state machine that is executed for this test is presented in Figure 9.55.

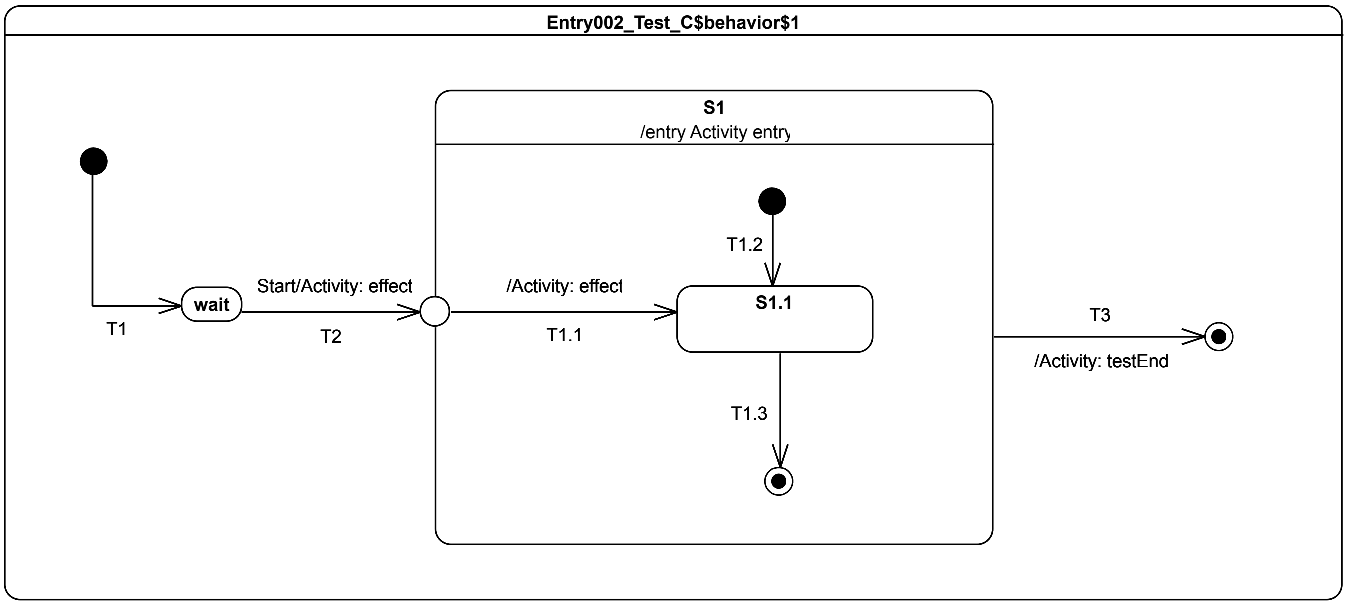


Figure 9.55 - Entry 002 - C Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- T2(effect)::S1(entry)::T1.1(entry)

Note. The purpose of this test is to demonstrate that the entry behavior of the state owning the entry point is always executed before the effect behavior(s) of the transition(s) originating from this entry point. When the Start event occurrence is dispatched and accepted by the state machine, *T2* is triggered and traversed. This traversal implies the execution of the effect behavior as well as the entrance of the entry point pseudostate. When the entry point is entered, *S1* is entered and its entry behavior is executed. As soon as the previous actions have completed, the continuation transition *T1.1* can be traversed. Hence its effect behavior is executed and *S1.1* is entered.

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)]
4	[**CE(S1.1)**]	[S1[S1.1]]	[T1.3]
5	[**CE(S1)**]	[S1]	[T3]

9.3.7.5 Entry 002-D

Tested state machine

The state machine that is executed for this test is presented in Figure 9.56.

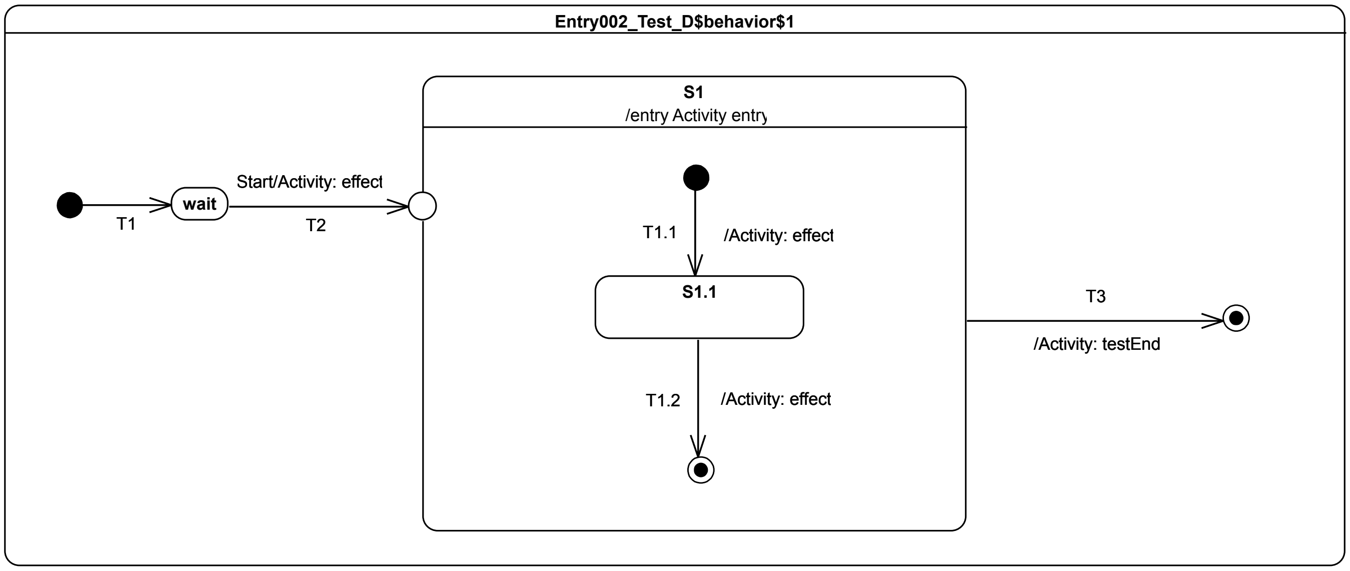


Figure 9.56 - Entry 002 - D Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- T2(effect)::S1(entry)::T1.1(effect)::T1.2(effect)

Note. The purpose of this test is to consolidate what was shown in previous test-cases presented in 9.3.7. It demonstrates that, if the composite has a single region and is entered through an entry with no outgoing transitions, then the region is entered using the default entry approach. When the Start event occurrence is dispatched and accepted by the state machine, *T2* is triggered and traversed. This is manifested in the trace by the message *T2(effect)*. Next, S1 is entered and its entry behavior is executed (see message *S1(entry)* in the trace). Finally, we see the region is entered using the default approach, since the execution of the *T1.1* effect behavior adds message *T1.1(effect)* to the trace.

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)]
4	[**CE(S1.1)**]	[S1[S1.1]]	[T1.2]
5	[**CE(S1)**]	[S1]	[T3]

9.3.7.6 Entry 002-E

Tested state machine

The state machine that is executed for this test is presented in Figure 9.57.

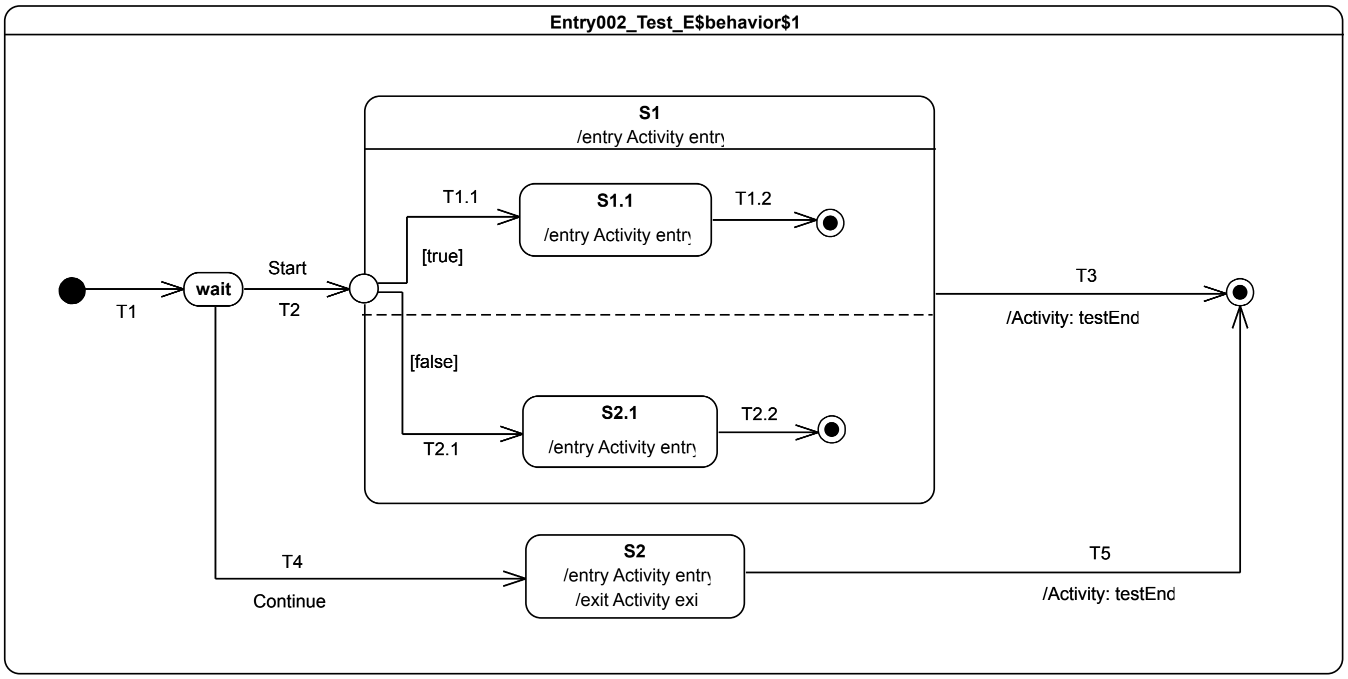


Figure 9.57: Entry002 - E Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *wait*.

Generated trace

- S2(entry)::S2(exit)

Note. The purpose of this test is to demonstrate the impact of the static analysis performed before an event occurrence is actually dispatched on the execution. In that test case, the *Start* event occurrence is lost. This loss is due to the impossibility of finding a path to a valid state machine configuration from the configuration *wait*. Indeed, the compound transition *T2(T1.1, T2.1)* cannot be traversed since the transition *T2.1* has a guard evaluating to false. The reason for this choice is that, as the entry point plays the same role as fork pseudostate (because all of its outgoing transitions target states located in orthogonal regions), hence it must be possible to propagate the execution to all of its outgoing transitions. When the *Continue* event occurrence is dispatched the state machine is in configuration *wait*. Transition *T4* is fired using this event occurrence and state *S2* is entered. The completion event for *S2* when dispatched triggers *T5,* which enables the state machine execution to reach the final state and to complete.

RTC Steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[]
4	[**Continue**]	[wait]	[T4]
5	[**CE(S2)**]	[S2]	[T5]

9.3.7.7 Entry 002-F

Tested state machine

The state machine that is executed for this test is presented in Figure 9.58.

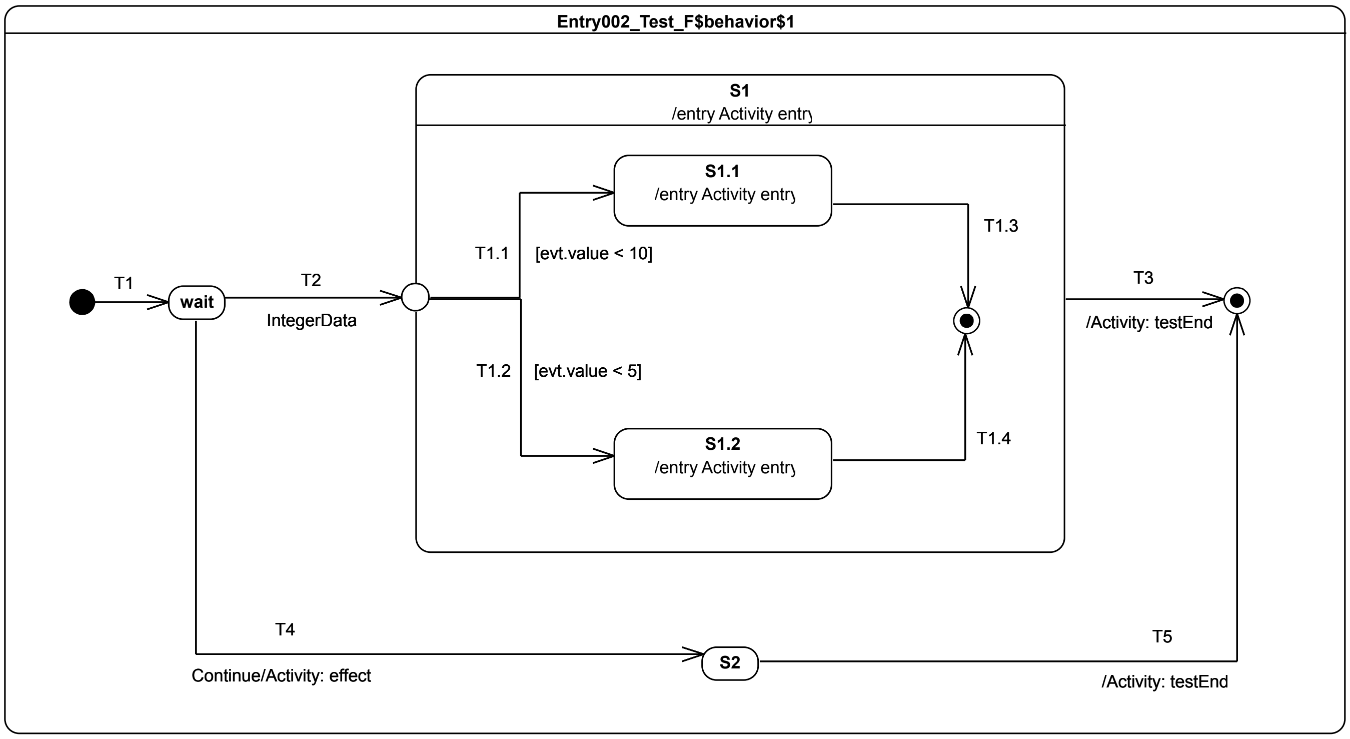


Figure 9.58: Entry 002 - F Test Classifier Behavior

Test executions

Received event occurrence(s)

- IntegerData(8) – received when in configuration *wait*.

- Continue – received either on configuration *S1* or *S2*.

Generated trace

- S1(entry)[in=8]::S1.1(entry)[in=8]

Note. In this test, the entry point plays the role of a junction pseudostate (it has outgoing transitions to different states located in the same region). The test case shows that a valid path can be found during the static analysis phase from the wait configuration to the *S1[S1.1]* configuration. Indeed when *IntegerData(8)* is dispatched, the compound transition *T2(T1.1)* is traversed because the guard on *T1.1* evaluates to true, while the guard on *T1.2* evaluates to false. The completion event generated during that step for *S1.1* is used in the next step to trigger *T1.3*. Hence *S1* completes and *T3* is fired when the completion event is dispatched.

RTC steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[**IntegerData(8)**]	[wait]	[T2(T1.1)]
3	[**CE(S1.1)**]	[S1[S1.1]]	[T1.3]
4	[Continue, **CE(S1)**]	[S1]	[T3]

9.3.8 Exit

9.3.8.1 Overview

Tests presented in this subclause assess that semantics associated with exit points conform to what is specified in UML.

9.3.8.2 Exit 001

Tested state machine

The state machine that is executed for this test is presented in Figure 9.59.

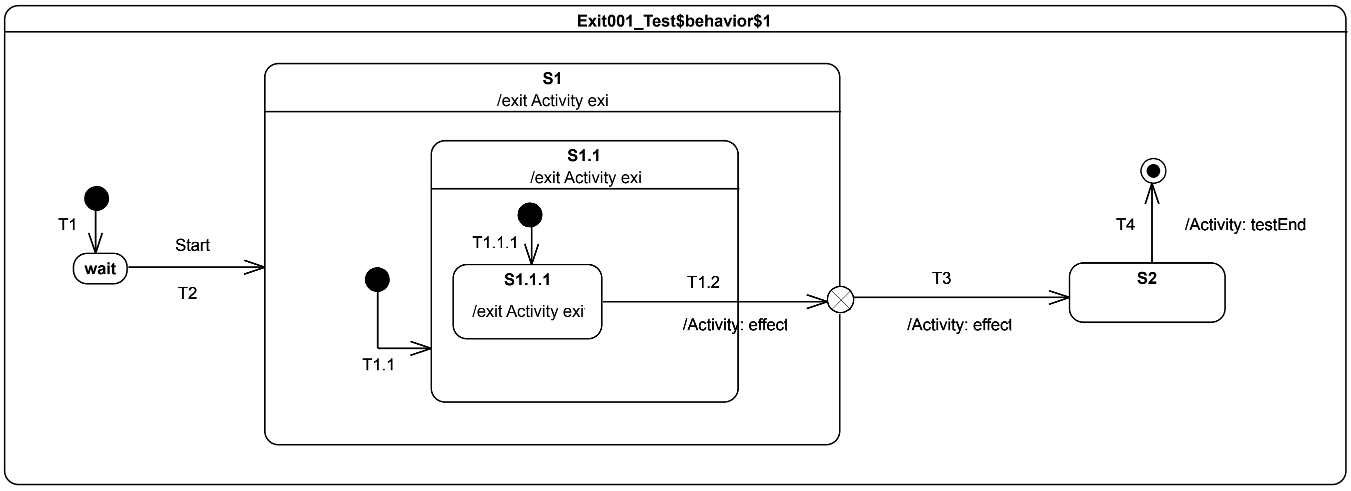


Figure 9.59 - Exit 001 Test Classifier behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- S1.1.1(exit)::S1.1(exit)::T1.2(effect)::S1(exit)::T3(effect)

Note. The purpose of this test case is to demonstrate support of the exit point pseudostate for exiting a composite state. The completion event generated by *S1.1.1* is dispatched and accepted when the state machine is in configuration *S1[S1.1[S1.1.1]]*. At this point, *T1.2* is triggered. When traversed, this transition implies first that *S1.1.1* is exited as well as *S1.1*. Next, its effect behavior is executed and, finally, the exit point placed on *S1* is reached. The semantics of the exit point requires *S1* to be exited and transition *T3* to be traversed.

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))]
4	[**CE(S1.1.1)**]	[S1[S1.1[S1.1.1]]]	[T1.2(T3)]
5	[**CE(S2)**]	[S2]	[T4]

9.3.8.3 Exit 002

Tested state machine

The state machine that is executed for this test is presented in Figure 9.60.

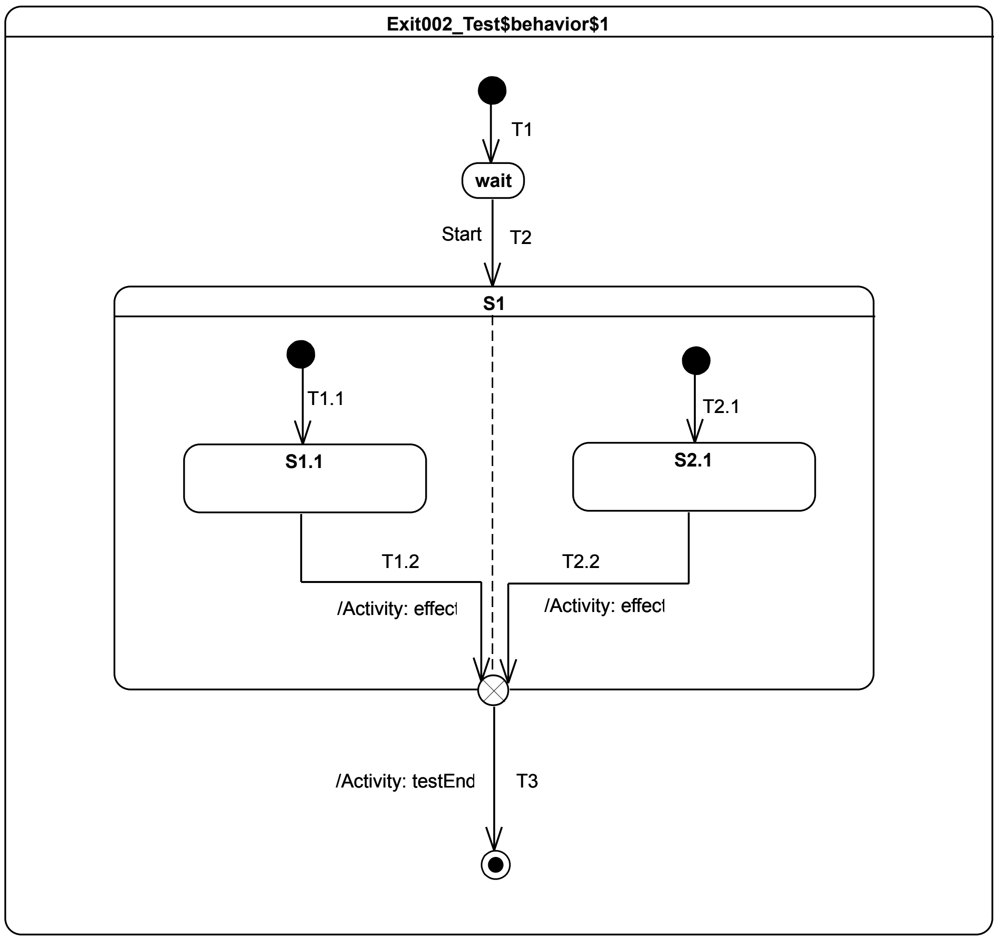


Figure 9.60 - Exit 002 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- T1.2(effect)::T2.2(effect)

Note. The purpose of this test is to demonstrate that, if multiple transitions originating from states located in different orthogonal regions terminate on an exit pseudostate, then it acts (in addition to its original semantics) as a join pseudostate.

When the completion event generated by *S1.1* is dispatched and accepted by the state machine, then *T1.2* is triggered and traversed. This is the first time that the exit point is reached. It cannot be traversed since its prerequisites are not satisfied (i.e., all of its incoming transition have not already been traversed). The next RTC step is initiated by the acceptance of the *S2.1* completion event. *T2.2* is triggered, after which the execution reaches the exit point for the second time. The latter is traversed and its outgoing transition is taken. The state machine execution completes when the final state is reached.

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, T2.1)]
4	[CE(S2.1), **CE(S1.1)**]	[S1[S1.1, S2.1]]	[T1.2]
5	[**CE(S2.1)**]	[S1[S2.1]]	[T2.2(T3)]

Alternative execution traces

In the test, there is concurrency specified in state *S1*. This means an alternative execution is possible for the state machine under test. This trace is described below and shows the case where *T2.2* effect behavior is executed after the *T1.2* effect behavior.

- T2.2(effect)::T1.2(effect)

9.3.8.4 Exit 003

Tested state machine

The state machine that is executed for this test is presented in Figure 9.61.

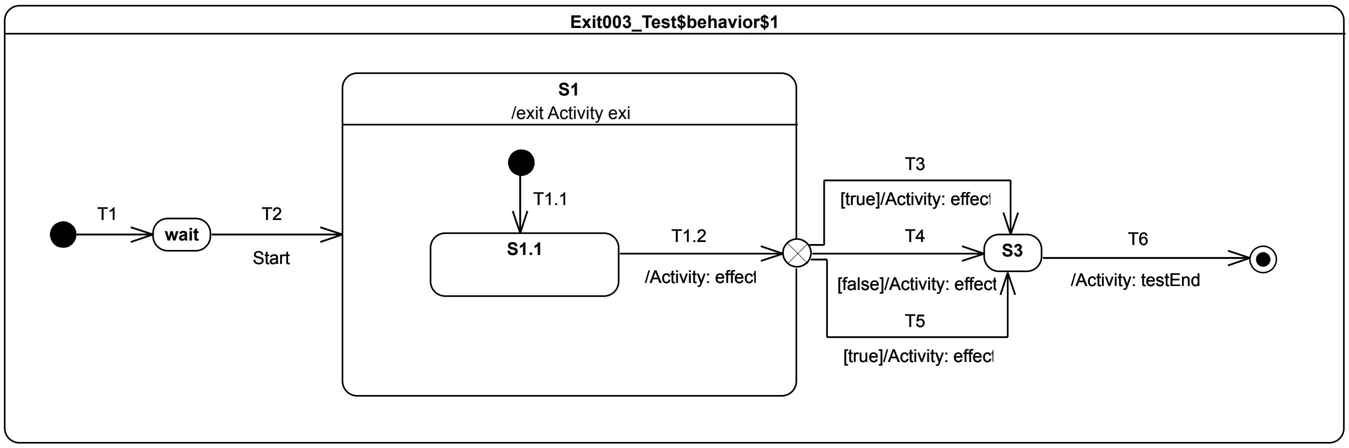


Figure 9.61 - Exit 003 Test Classifier Behavior

Test executions

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- T1.2(effect)::S1(exit)::T3(effect)

Note. The purpose of this test case is to ensure that, in a situation where multiple transitions outgoing an exit point are ready to be traversed, only one of them will actually be selected for firing. At the point of the execution when the *S1.1* completion event is accepted, the exit point that is placed on *S1* is reached. The guards placed on transitions originating from this exit point are evaluated. The set of enabled transitions is now composed of *T3* and *T5*. The transition to be traversed is chosen nondeterministically. In the case of the above trace, *T3* is chosen and *S3* is entered. The completion event generated by *S3* will be dispatched in the next RTC step and the state machine will complete its execution.

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)]
4	[**CE(S1.1)**]	[S1[S1.1]]	[T1.2(T3)]
5	[**CE(S3)**]	[S3]	[T6]

Alternative execution traces

In this test, the exit point has two outgoing transitions (*T3* and *T5*) that can be fired at the same time. This results in a conflicting situation, where only one transition will actually be chosen to fire. As this choice is nondeterministic, one alternative execution trace can be observed for this test case.

- T1.2(effect)::S1(exit)::T5(effect)

9.3.9 Choice

9.3.9.1 Overview

Tests presented in this subclause assess that choice semantics conform to what is specified in UML.

9.3.9.2 Choice 001

Tested state machine

The state machine that is executed for this test is presented in Figure 9.62.

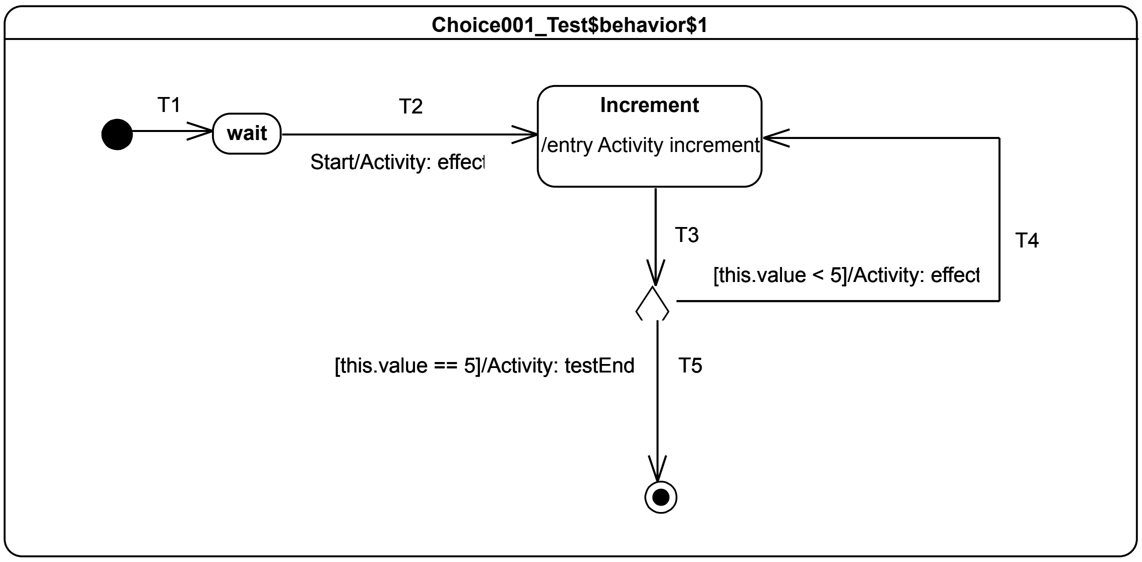

Choice001_Test$behavior$1
T1
wait
T2
Start/Activity: effect
Increment
/entry Activity increment
T3
T4
[this.value < 5]/Activity: effect
[this.value == 5]/Activity: testEnd
T5

Figure 9.62 - Choice 001 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- T4(effect)::T4(effect)::T4(effect)::T4(effect)

Note. The purpose of this test is to demonstrate how evaluation of transition guards of transitions originating from a choice pseudostate has an impact on the execution flow. When the *Start* event occurrence is dispatched and accepted by the state machine, *T2* is triggered. The execution of its associated effect behavior implies the initialization of the *value* property of the class for which the state machine plays the role of a classifier behavior. The *Increment* state is entered and its entry behavior increments the value of property *value*. Right after the termination of the entry behavior, a completion event is generated for the state *Increment*. This is the end of the RTC step initiated by the acceptance of the *Start* event occurrence. The next RTC step is initiated by the acceptance of the completion event generated by the *Increment* state. This triggers *T3*, which is the incoming transition of the choice pseudostate. When it is reached, all guards placed on outgoing transitions are evaluated. Only the guard placed on *T4* evaluates to true, so that this continuation transition is taken. This leads to re-entering of the *Increment* state. The next four RTC steps repeat this execution path. The fifth consists in traversing *T5*, whose guard now evaluates to true. When the final state is reached the state machine completes its execution.

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]
4	[**CE(Increment)**]	[Increment]	[T3(T4)]
5	[**CE(Increment)**]	[Increment]	[T3(T4)]
6	[**CE(Increment)**]	[Increment]	[T3(T4)]
7	[**CE(Increment)**]	[Increment]	[T3(T4)]
8	[**CE(Increment)**]	[Increment]	[T3(T5)]

9.3.9.3 Choice 002

Tested state machine

The state machine that is executed for this test is presented in Figure 9.63.

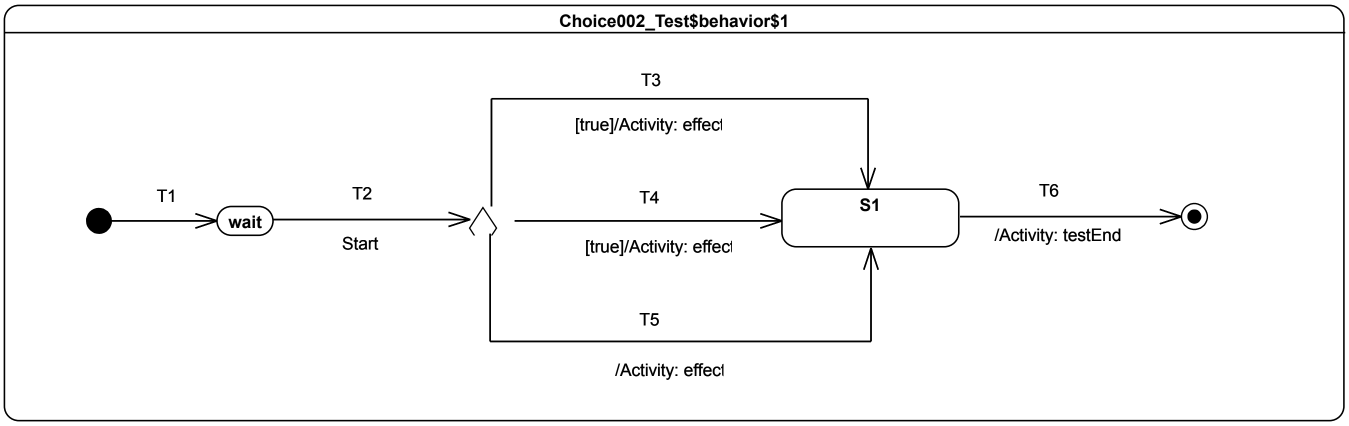


Figure 9.63 - Choice 002 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- T4(effect)

Note. The purpose of this test is to demonstrate that, if many transitions originating from a choice pseudostate are enabled, then at most one of them is chosen to be traversed. When the *Start* event occurrence is dispatched and accepted by the state machine, *T2* is triggered. The choice pseudostate reached at this point has outgoing transitions. Each guard of each transition is evaluated. It happens at this point that *T3*, *T4* and *T5* are all ready to be traversed (a transition with no explicit guard is considered to have a guard that always evaluates to true). The transition that will be fired is chosen nondeterministically. In the case of the above trace, *T4* is chosen to be traversed.

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(T4)]
4	[**CE(S1)**]	[S1]	[T6]

Alternative execution traces

In this test, the choice has three outgoing transitions (*T3, T4* and *T5*) that can be fired at the same time. This result is a conflicting situation where only one transition will actually be chosen to fire. As this choice is nondeterministic, two alternative execution traces can be observed for this test case.

- T3(effect)
- T5(effect)

9.3.9.4 Choice 003

Tested state machine

The state machine that is executed for this test is presented in Figure 9.64.

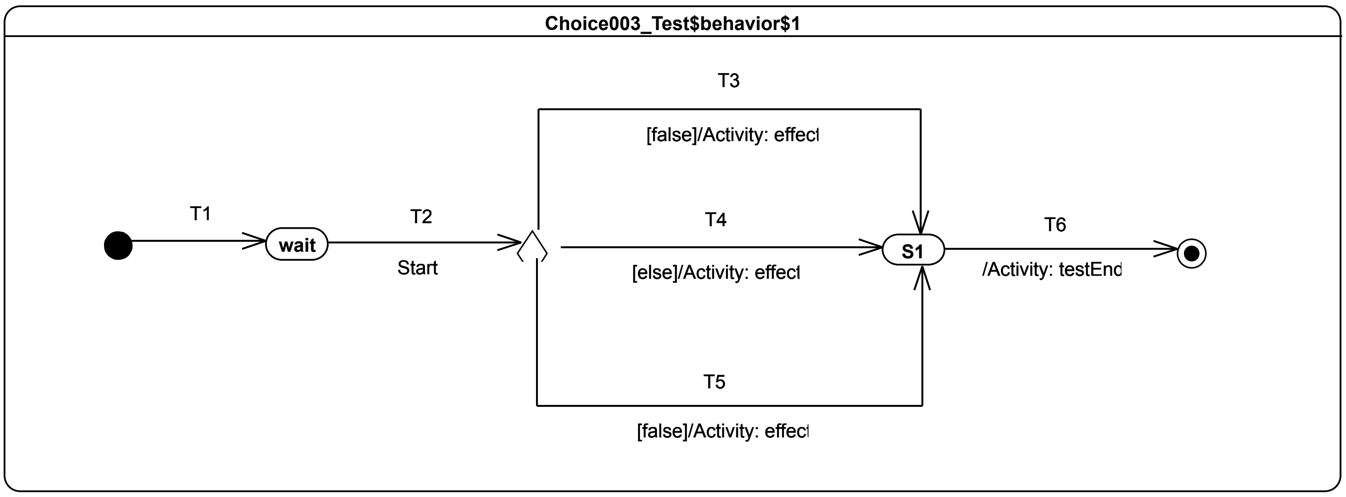


Figure 9.64 - Choice 003 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- T4(effect)

Note. The purpose of this test is to demonstrate that, if a choice point has an *else* outgoing transition and all of other outgoing transitions have guards that evaluate to false, then the *else* transition is chosen and traversed. When the *Start* event occurrence is dispatched and accepted, *T2* is triggered, which enables the state machine to reach the choice point. At this point, the guards of transitions *T3* and *T5* are evaluated. Neither of them evaluates to true, but there also exists an *else* transition *T4*. This transition (i.e., *T4*) is traversed and *S1* is entered. The completion event generated by *S1* is used to trigger *T6,* which leads to the completion of the state machine execution.

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(T4)]
4	[**CE(S1)**]	[S1]	[T6]

9.3.9.5 Choice 004

Tested state machine

The state machine that is executed for this test is presented in Figure 9.65.

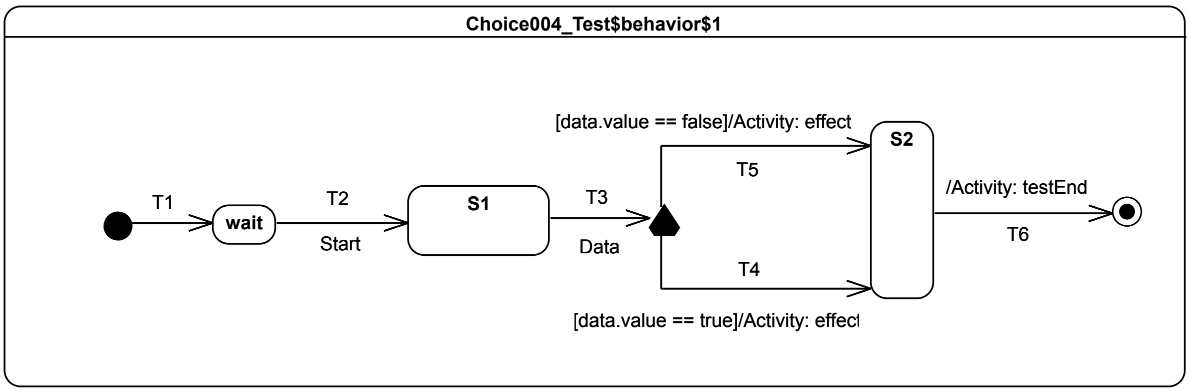

Choice004_Test$behavior$1
T1
wait
T2
Start
S1
T3
Data
[data.value == false]/Activity: effect
T5
T4
[data.value == true]/Activity: effect
S2
/Activity: testEnd
T6

Figure 9.65 - Choice 004 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Data(true) – received when in configuration *S1*.

Generated trace

- T4(effect)

Note. The purpose of this test case is to demonstrate that data available in an event occurrence can be made available to guards placed on outgoing transitions of a choice pseudostate. When the *Data* event occurrence is dispatched, the compound transition *T3(T4)* is traversed. *T4* is chosen because its guard evaluates to true. Note that the guards on the outgoing transitions of a choice pseudostate are not evaluated during the static analysis phase, but they are evaluated at the time the choice is reached by the execution flow. This means that, even in the case where the step is accepted by the state machine, there is no guarantee that the execution will be able to reach a valid state machine configuration. The dispatching of the *S2* completion event enables the transition *T6* to be fired.

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]
4	[Data(true), **CE(S1)**]	[S1]	[]
5	[**Data(true)**]	[S1]	[T3(T4)]
6	[**CE(S2)**]	[S2]	[T6]

9.3.9.6 Choice 005

Tested state machine

The state machine that is executed for this test is presented in Figure 9.66.

- T1.2 guard evaluates to true.
- T1.3 guard evaluates to false.
- T1.4 guard evaluates to true.
- T1.5 guard evaluates to false.

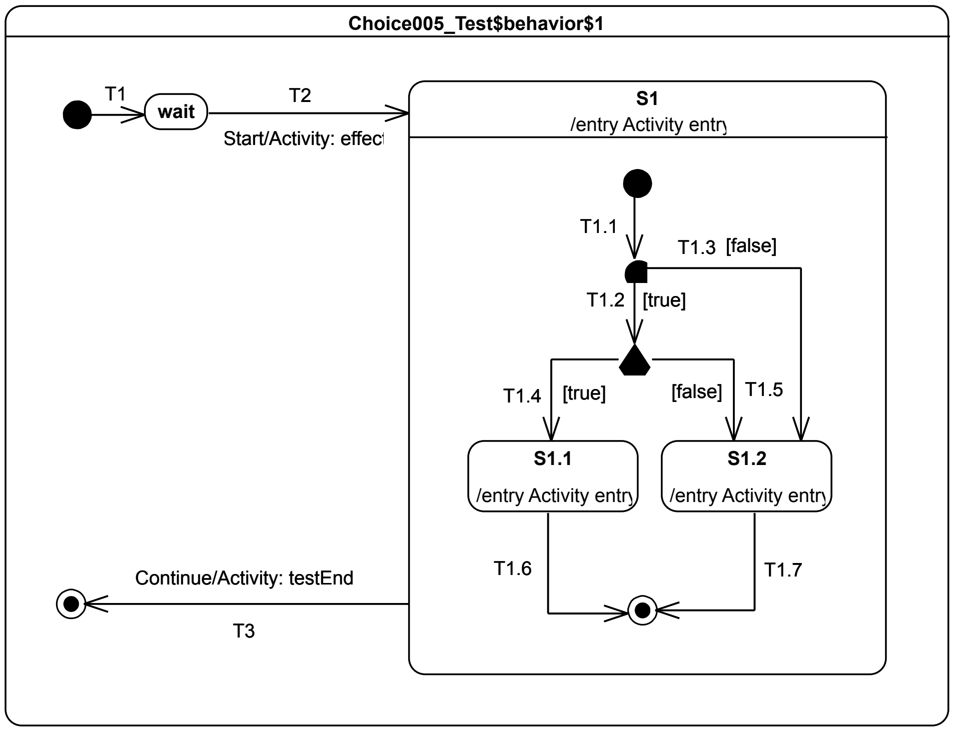


Figure 9.66 - Choice 005 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1*.

Generated trace

- T1.2(guard)::T1.3(guard)::T2(effect)::S1(entry)::T1.4(guard)::T1.5(guard)::S1.1(entry)

Note. The purpose of this test is to demonstrate that guards of transitions outgoing a choice pseudostate are not evaluated prior to an RTC step but during the RTC step. To demonstrate this, guards placed on *T1.2*, *T2.3*, *T1.4* and *T1.5* contribute to the trace generated by this test case. Consider the situation where the state machine is in configuration *wait,* and the *Start* event is about to be dispatched. Before realizing the RTC step, the static analysis phase takes place. During this static analysis, the compound transitions *T2(T1.1, T1.2)* and *T2(T1.1, T1.3)* are evaluated in order to determine if a valid state machine configuration can be reached. Guards on transitions are evaluated during this phase, which explains that the trace fragments *T1.2*(guard) and *T1.3*(guard) appear at the beginning of the trace. In this test it is only possible for that step to reach a valid state machine configuration using the compound *T2(T1.1, T1.2)* (*T1.3* guard evaluates to false). One can notice that, during the static analysis, guards of outgoing transitions of the choice pseudo state have not been evaluated. These latter are clearly evaluated when the choice pseudostate is reached since the trace shows that *T2(effect)* and *S1(entry)* fragments appear before *T1.4(guard)::T1.5(guard)*. When *S1.1* is entered a completion event is generated, which enables *T6* to be fired in the next step and *S1* to complete. The last RTC step is realized when the *Continue* event is dispatched.

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, T1.2, T1.4)]
4	[**CE(S1.1)**]	[S1[S1.1]]	[T1.6]
5	[Continue, **CE(S1)**]	[S1]	[]
6	[**Continue**]	[S1]	[T3]

9.3.10 Junction

9.3.10.1 Overview

Test presented in this subclause assess that junction pseudostate semantics conform to what is specified in UML.

9.3.10.2 Junction 001

Tested state machine

The state machine that is executed for this test is presented in Figure 9.67.

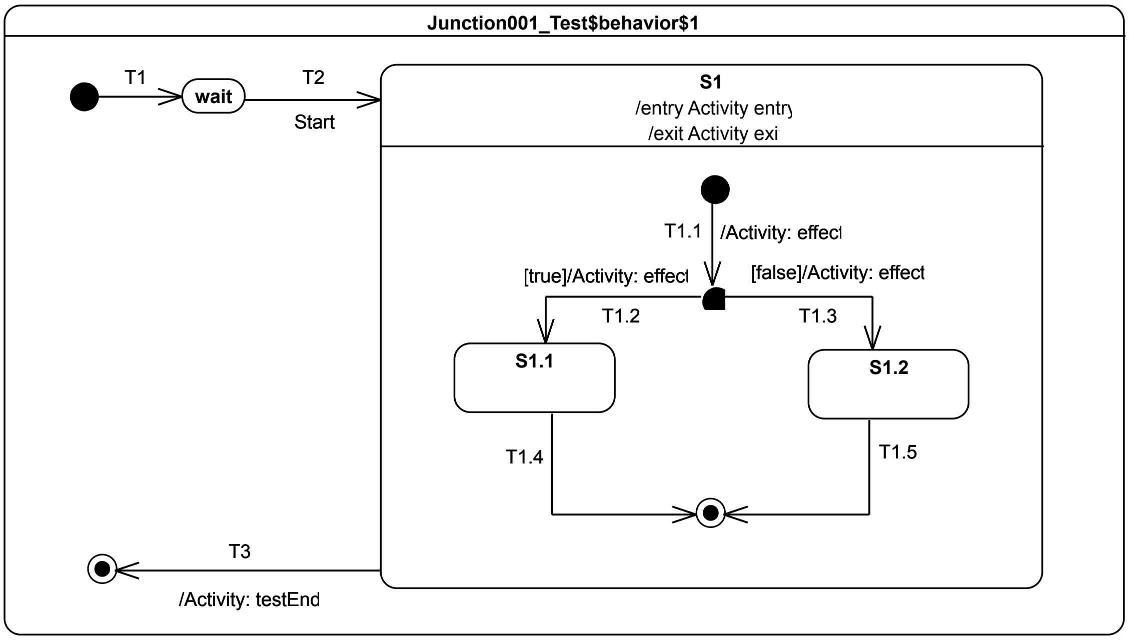


Figure 9.67 - Junction 001 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- S1(entry)::T1.1(effect)::T1.2(effect)::S1(exit)

Note. The purpose of this test is to demonstrate that junction pseudostate guards are evaluated before the RTC step that would traverse the junction is executed. Consider the situation where the state machine is in configuration *wait*. Prior to dispatching of the *Start* event occurrence, a static analysis is performed on the state machine. The purpose of this static analysis is to determine whether, from the current configuration, at least one valid path can be found to the next configuration. In this case, the paths that are evaluated are represented by the compound transitions *T2(T1.1, T1.2)* and *T2(T1.1, T1.3)*. During the evaluation, the guards on these transitions are evaluated. This leads to the conclusion that, in the current situation, only *T2(T1.1, T1.2)* can be traversed. Specifically, following this transition will reach the configuration *S1[S1.1]*. It is not possible to reach *S1[S1.2]* due to the *T1.3* guard, which evaluates to false. Following the traversal of *T2(T1.1, T1.2)*, *S1.1* is entered and a completion event occurrence is generated for that state. This completion event occurrence triggers *T1.4*, which leads to the completion of *S1*. Its completion event occurrence is then used in the next step to fire *T3*.

RTC steps

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

9.3.10.3 Junction 002

Tested state machine

The state machine that is executed for this test is presented in Figure 9.68.

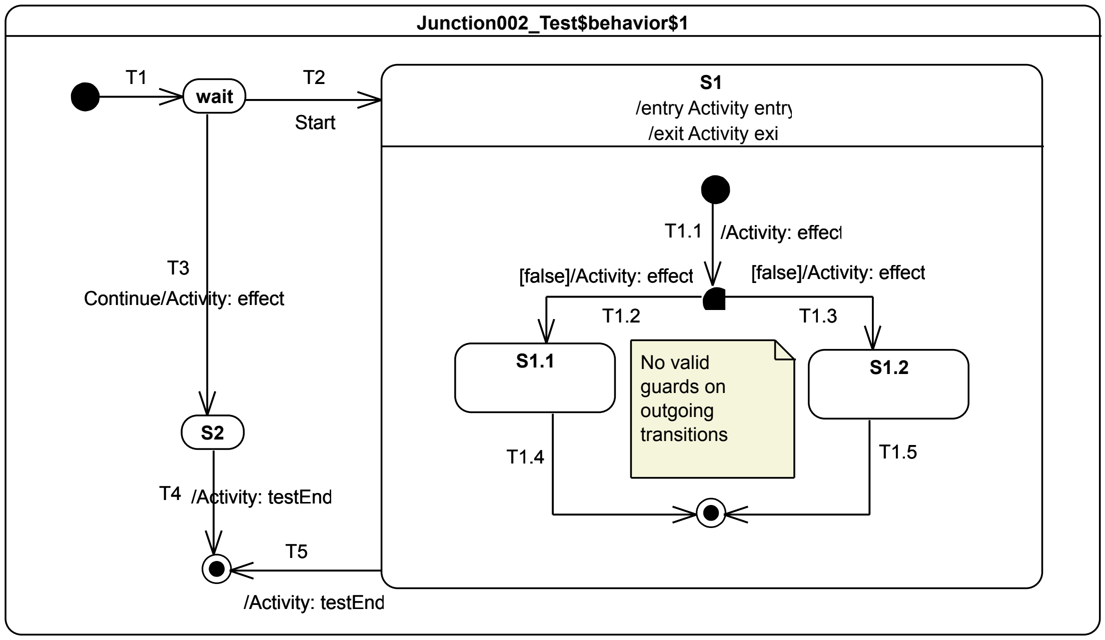


Figure 9.68 - Junction 002 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *wait*.

Generated trace

- T3(effect)

Note. The purpose of this test is to demonstrate that, in the situation where a path does not lead to a valid state machine configuration, this path is disabled. If all paths are disabled, then the event occurrence that should have been used to trigger one path is dispatched and lost. Consider the situation where the state machine is in the configuration *wait* and the *Start* event occurrence is about to be dispatched. Possible paths to be triggered by the *Start* event occurrence are *T2(T1.1, T1.2)* and *T2(T1.1, T1.3)*. Nevertheless, none of them can be traversed to reach a valid state machine configuration. Indeed, both *T1.2* and *T1.3,* which are parts of the paths, have a guard evaluating to false. Hence the *Start* event occurrence is dispatched and lost and the state machine remains in configuration *wait*. When the *Continue* event occurrence is dispatched, *T3* is triggered and *S2* is entered. The completion event occurrnce of *S2* is used to fire *T4* in the next step.

RTC steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[]
4	[**Continue**]	[wait]	[T3]
5	[**CE(S2)**]	[S2]	[T4]

9.3.10.4 Junction 003

Tested state machine

The state machine that is executed for this test is presented in Figure 9.69.

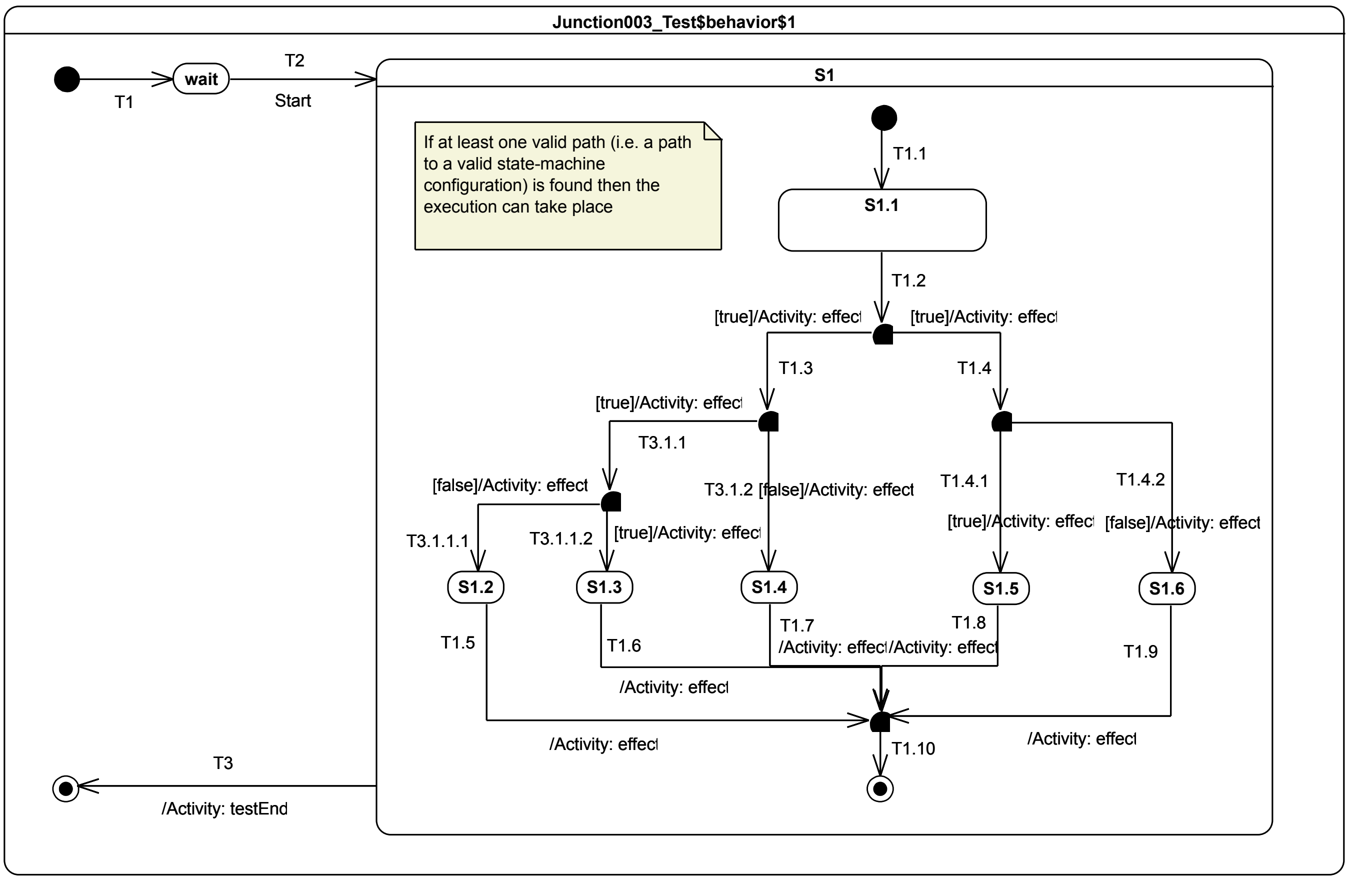


Figure 9.69 - Junction 003 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- T1.3(effect)::T3.1.1(effect)::T3.1.1.2(effect)::T1.6(effect)

Note. The purpose of this test is twofold. First, it demonstrates the flow of the static analysis along nested paths. Second, it demonstrates the capability of the semantics to resolve ambiguity if multiple transitions outgoing a junction have a guard evaluating to true. Consider the situation where the state machine is in configuration *S1[S1.1]*. When the *S1.1* completion event occurrence is dispatched, multiple paths are evaluated:

- ○ *T1.2(T1.3, T3.1.1, T3.1.1.1)*
- ○ *T1.2(T1.3, T3.1.1, T3.1.1.2)*
- ○ *T2(T1.3, T3.1.2)*

- *T2(T1.4, T1.4.1)*
- *T2(T1.4, T1.4.2)*

Among these paths only two offer the possibility to reach a valid state machine configuration:

- *T1.2(T1.3, T3.1.1, T3.1.1.2)*
- *T2(T1.4, T1.4.1)*

Assume that *T1.2(T1.3, T3.1.1, T3.1.1.2)* is chosen to be traversed. This brings the state machine to the configuration *S1[S1.3]*. In the next RTC step, the completion event occurrence generated for *S1.3* is used to trigger the compound transition *T1.6(T1.10)*. When the final state is reached, *S1* completes. The last RTC step is initiated by the dispatching of its completion event occurrence.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2]
4	[**CE(S1.1)**]	[S1[S1.1]]	T1.2(T1.3, T3.1.1, T3.1.1.2)
5	[**CE(S1.3)**]	[S1[S1.3]]	[T1.6(T1.10)]
6	[**CE(S1)**]	[S1]	[T3]

Alternative execution traces

In this test, the junction has two outgoing transitions (T1.3 and T1.4) that can be fired at the same time. This result is a conflicting situation where only one transition will actually be chosen to fire. As this choice is nondeterministic, one alternative execution trace can be observed for this test case.

- T1.4(effect)::T1.4.1(effect)::T1.8(effect)

9.3.10.5 Junction 004

Tested state machine

The state machine that is executed for this test is presented in Figure 9.70.

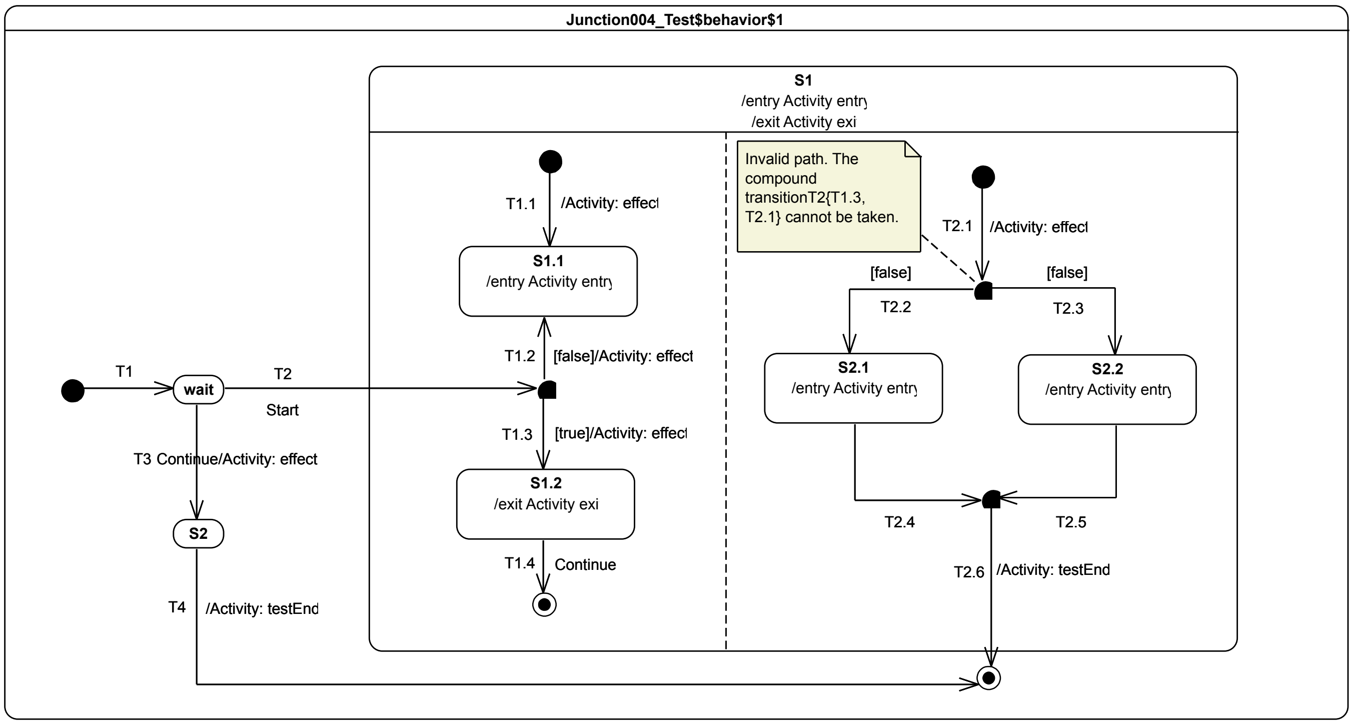


Figure 9.70 - Junction 004 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *wait*.

Generated trace

- T3(effect)

Note. The purpose of this test is to demonstrate the flow of the static analysis through orthogonal regions. Consider the situation where the state machine is in configuration *wait* and the *Start* event occurrence is about to be dispatched. In this case, none of the four paths starting from *T2* that are evaluated allow the state machine to reach a valid configuration. Indeed, even if it were possible to find one path (*T2(T1.3)*) in the left region, it is not possible to find any valid path in the right region. Both *T2(T2.1, T2.2)* and *T2(T2.1, 2.3)* contain a transition with a guard evaluating to false. Hence, when the *Start* event occurrence is dispatched, it is lost. The next event occurrence to be dispatched is for the signal *Continue*, which triggers *T3*. This enables the state machine to reach the configuration S2. The completion event occurrence generated by that state triggers the last RTC step.

RTC steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[]
4	[**Continue**]	[wait]	[T3]
5	[**CE(S2)**]	[S2]	[T4]

9.3.10.6 Junction 005

Tested state machine

The state machine that is executed for this test is presented in Figure 9.71.

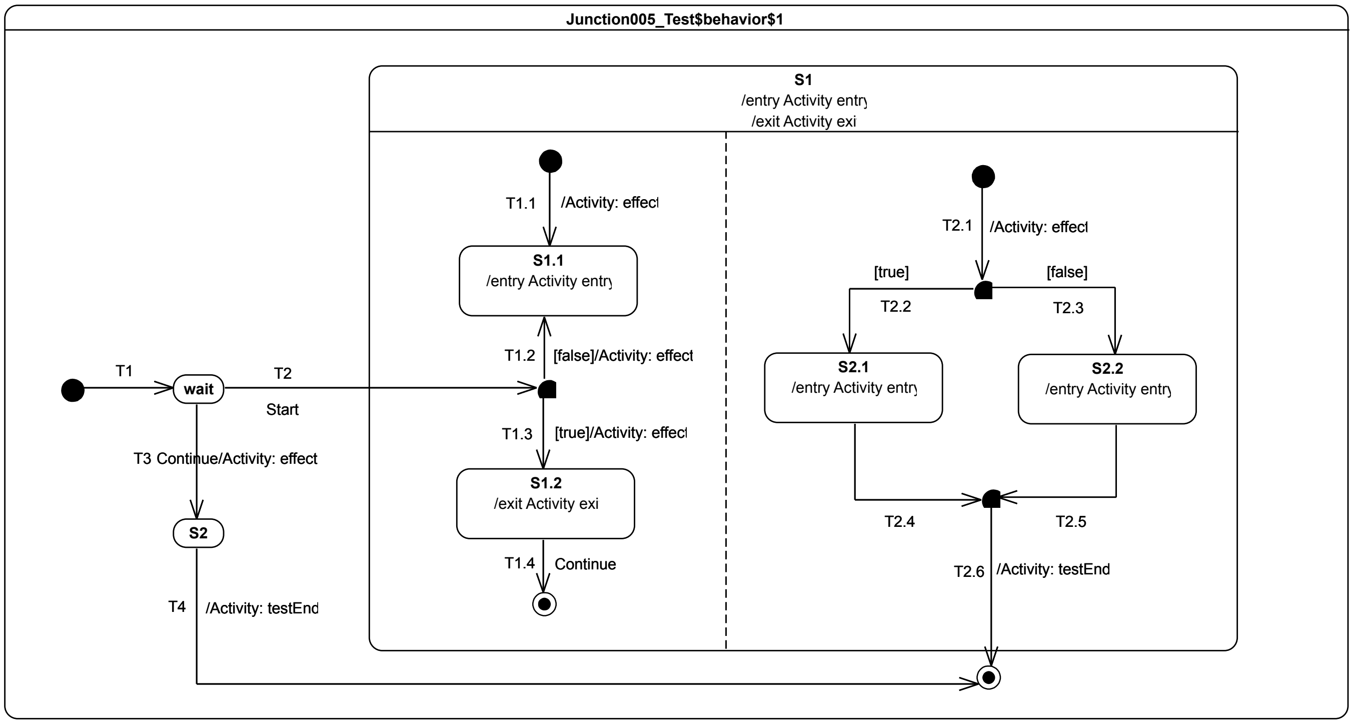


Figure 9.71 - Junction 005 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

- Continue – received in configuration *wait*.

Generated trace

- S1(entry)::T2.1(effect)::S2.1(entry)::T1.3(effect)::S1.2(exit)::S1(exit)

  Note. The purpose of this test is to demonstrate the flow of the static analysis through orthogonal regions. This test is the exact opposite of the Junction 004 test in the sense that, when the *Start* event occurrence is dispatched, it is possible to find a valid path to the next valid state machine configuration. Consider the situation where the state machine is in configuration *wait*. When the *Start* event occurrence is about to be dispatched, four paths can be evaluated. Among these paths, the one corresponding to the compound transition *T2(T1.3, T2.1, T2.2)* is the only one that leads to a valid state machine configuration: *S1[S1.2, S2.1]*. Consequently, this compound transition is traversed. The completion event occurrence generated for *S1.2* is lost, since the state has no completion transition available. This is not the case for the *S2.1* completion event occurrence, which, when dispatched, triggers the compound transition *T2.4(T2.6)*. During the traversal of this transition *S1.2* and *S1* are exited. 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	[InitialTransition]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T1.3, T2.1, T2.2)]
4	[CE(S1.2)**, CE(S2.1)**]	[S1[S1.2, S2.1]]	[T2.4(T2.6)]

Alternative execution steps

The presence of orthogonal regions in *S1* implies the possibility of valid alternative execution traces. These alternative execution traces are listed below.

- S1(entry)::T2.1(effect)::T1.3(effect)::S2.1(entry)::S1.2(exit)::S1(exit)
- S1(entry)::T1.3(effect)::T2.1(effect)::S2.1(entry)::S1.2(exit)::S1(exit)

9.3.10.7 Junction 006

Tested state machine

The state machine that is executed for this test is presented in Figure 9.72.

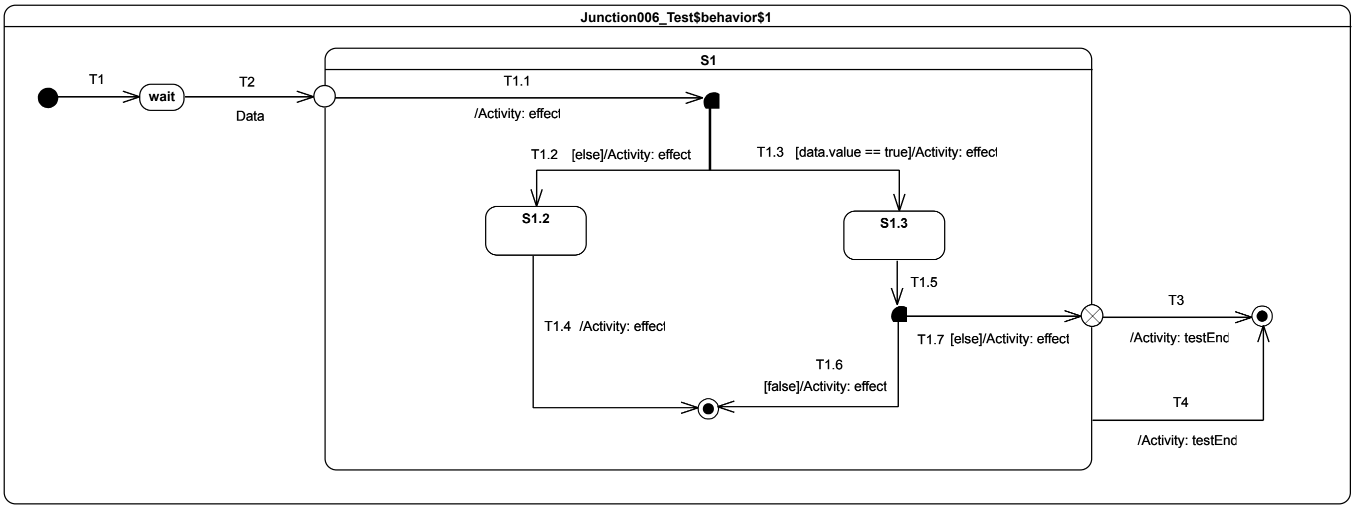


Figure 9.72 - Junction 006 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Data(true) – received when in configuration *wait*.

Generated trace

- T1.1(effect)[in=true]::T1.3(effect)[in=true]::T1.7(effect)

Note. The purpose of this test is twofold. First, it demonstrates the capability of accessing event data during static analysis. Second, it demonstrates usage of the *else* transition in cases where none of the other transitions outgoing a junction has a guard evaluating to true. Consider the situation where the state machine is in configuration *wait* and a *Data(true)* event occurrence is about to be dispatched. A valid path (*T2(T1.1, T1.3)*) allows the state machine to move from the current configuration to configuration *S1[S1.3]*. There is no need to use the *else* transition in this case, since the *T1.3* guard evaluates to true. Therefore, after the dispatching of *Data(true)*, the state machine enters configuration *S1[S1.3].* The completion event occurrence generated by *S1.3* is used during the following RTC step to trigger the compound transition *T1.5(T1.7, T3)*, which leads the state machine execution to reach the final state. Note that the *else* transition *T1.7* is used for this step, since this is the only way to traverse through the junction pseudostate.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Data(true), **CE(wait)**]	[wait]	[]
3	[**Data(true)**]	[wait]	[T2(T1.1, T1.3)]
4	[**CE(S1.3)**]	[S1[S1.3]]	[T1.5(T1.7, T3)]

9.3.11 Fork

9.3.11.1 Overview

Tests presented in this subclause assess that fork semantics conform to what is specified in UML.

9.3.11.2 Fork 001

Tested state machine

The state machine that is executed for this test is presented in Figure 9.73.

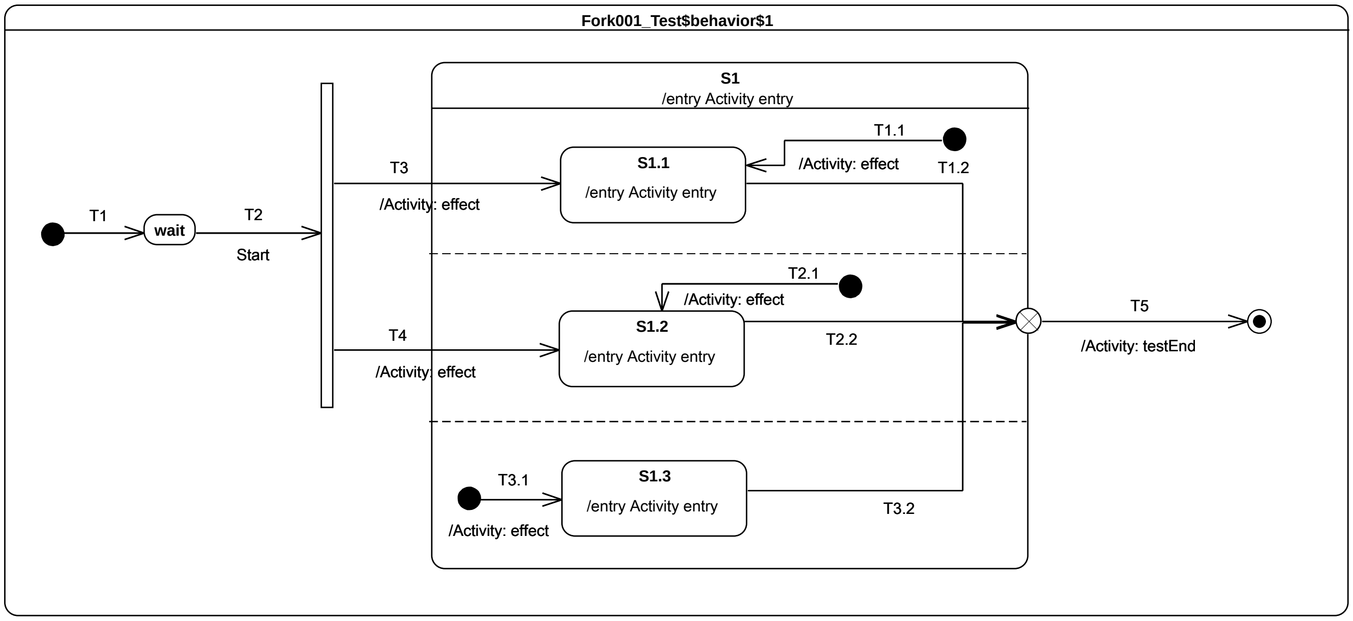


Figure 9.73 - Fork 001 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- T3(effect)::S1(entry)::T3.1(effect)::S1.3(entry)::S1.1(entry)::T4(effect)::S1.2(entry).

Note. The purpose of this test is to demonstrate the support for fork pseudostate semantics, as well as the preservation of region entry rules, when this pseudostate is used. When the Start event occurrence is dispatched and accepted by the state machine, it triggers *T2*. Traversal of this transition brings the state machine to the fork pseudostate. Both of its outgoing transitions are fired concurrently. However neither *S1.1* nor *S1.2* are entered immediately. S1 is entered first, its entry behavior is executed, and all regions that are not entered explicitly are started concurrently. This implies that the third region (i.e., the one containing *S1.3*) starts its execution starting from the initial pseudostate. Hence, *T3.1* is traversed and *S1.3* is entered (a completion event is generated for that state when its entry behavior has finished). Finally both *S1.1* and

S1.2 are entered. This concludes the RTC step that was initiated by the acceptance of the *Start* event occurrence. The three completion events generated by *S1.3*, *S1.1* and *S2.1* will be used in next RTC steps to trigger *T3.2*, *T1.2* and *T2.2*.

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(T3.1, T3, T4)]
4	[CE(S1.2), CE(S1.1), **CE(S1.3)**]	[S1[S1.1, S1.2, S1.3]]	[T3.2]
5	[CE(S1.2), **CE(S1.1)**]	[S1[S1.1, S1.2]]	[T1.2]
6	[**CE(S1.2)**]	[S1[S1.2]]	[T2.2(T5)]

Alternative execution traces

The presence of the fork pseudostate and the three orthogonal regions in *S1* imply the existence of multiple execution traces. These execution traces are listed below:

1. T3(effect)::T4(effect)::S1(entry)::S1.1(entry)::S1.2(entry)::T3.1(effect)::S1.3(entry)
2. T3(effect)::T4(effect)::S1(entry)::S1.1(entry)::T3.1(effect)::S1.2(entry)::S1.3(entry)
3. T3(effect)::T4(effect)::S1(entry)::S1.1(entry)::T3.1(effect)::S1.3(entry)::S1.2(entry)
4. T3(effect)::T4(effect)::S1(entry)::S1.2(entry)::S1.1(entry)::T3.1(effect)::S1.3(entry)
5. T3(effect)::T4(effect)::S1(entry)::S1.2(entry)::T3.1(effect)::S1.3(entry)::S1.1(entry)
6. T3(effect)::T4(effect)::S1(entry)::S1.2(entry)::T3.1(effect)::S1.1(entry)::S1.3(entry)
7. T3(effect)::T4(effect)::S1(entry)::T3.1(effect)::S1.1(entry)::S1.2(entry)::S1.3(entry)
8. T3(effect)::T4(effect)::S1(entry)::T3.1(effect)::S1.1(entry)::S1.3(entry)::S1.2(entry)
9. T3(effect)::T4(effect)::S1(entry)::T3.1(effect)::S1.2(entry)::S1.1(entry)::S1.3(entry)
10. T3(effect)::T4(effect)::S1(entry)::T3.1(effect)::S1.2(entry)::S1.3(entry)::S1.1(entry)
11. T3(effect)::T4(effect)::S1(entry)::T3.1(effect)::S1.3(entry)::S1.1(entry)::S1.2(entry)
12. T3(effect)::T4(effect)::S1(entry)::T3.1(effect)::S1.3(entry)::S1.2(entry)::S1.1(entry)
13. T3(effect)::S1(entry)::T3.1(effect)::T4(effect)::S1.1(entry)::S1.3(entry)::S1.2(entry)
14. T3(effect)::S1(entry)::T3.1(effect)::T4(effect)::S1.1(entry)::S1.2(entry)::S1.3(entry)
15. T3(effect)::S1(entry)::T3.1(effect)::T4(effect)::S1.3(entry)::S1.1(entry)::S1.2(entry)

16. T3(effect)::S1(entry)::T3.1(effect)::T4(effect)::S1.3(entry)::S1.2(entry)::S1.1(entry)
17. T3(effect)::S1(entry)::T3.1(effect)::T4(effect)::S1.2(entry)::S1.1(entry)::S1.3(entry)
18. T3(effect)::S1(entry)::T3.1(effect)::T4(effect)::S1.2(entry)::S1.3(entry)::S1.1(entry)
19. T3(effect)::S1(entry)::T3.1(effect)::S1.1(entry)::T4(effect)::S1.2(entry)::S1.3(entry)
20. T3(effect)::S1(entry)::T3.1(effect)::S1.1(entry)::T4(effect)::S1.3(entry)::S1.2(entry)
21. T3(effect)::S1(entry)::T3.1(effect)::S1.1(entry)::S1.3(entry)::T4(effect)::S1.2(entry)
22. T3(effect)::S1(entry)::T3.1(effect)::S1.3(entry)::T4(effect)::S1.1(entry)::S1.2(entry)
23. T3(effect)::S1(entry)::T3.1(effect)::S1.3(entry)::T4(effect)::S1.2(entry)::S1.1(entry)
24. T3(effect)::S1(entry)::T3.1(effect)::S1.3(entry)::S1.1(entry)::T4(effect)::S1.2(entry)
25. T3(effect)::S1(entry)::T4(effect)::T3.1(effect)::S1.2(entry)::S1.1(entry)::S1.3(entry)
26. T3(effect)::S1(entry)::T4(effect)::T3.1(effect)::S1.2(entry)::S1.3(entry)::S1.1(entry)
27. T3(effect)::S1(entry)::T4(effect)::T3.1(effect)::S1.1(entry)::S1.2(entry)::S1.3(entry)
28. T3(effect)::S1(entry)::T4(effect)::T3.1(effect)::S1.1(entry)::S1.3(entry)::S1.2(entry)
29. T3(effect)::S1(entry)::T4(effect)::T3.1(effect)::S1.3(entry)::S1.2(entry)::S1.1(entry)
30. T3(effect)::S1(entry)::T4(effect)::T3.1(effect)::S1.3(entry)::S1.1(entry)::S1.2(entry)
31. T3(effect)::S1(entry)::T4(effect)::S1.1(entry)::S1.2(entry)::T3.1(effect)::S1.3(entry)
32. T3(effect)::S1(entry)::T4(effect)::S1.1(entry)::T3.1(effect)::S1.3(entry)::S1.2(entry)
33. T3(effect)::S1(entry)::T4(effect)::S1.1(entry)::T3.1(effect)::S1.2(entry)::S1.3(entry)
34. T3(effect)::S1(entry)::T4(effect)::S1.2(entry)::T3.1(effect)::S1.1(entry)::S1.3(entry)
35. T3(effect)::S1(entry)::T4(effect)::S1.2(entry)::T3.1(effect)::S1.3(entry)::S1.1(entry)
36. T3(effect)::S1(entry)::T4(effect)::S1.2(entry)::S1.1(entry)::T3.1(effect)::S1.3(entry)
37. T3(effect)::S1(entry)::S1.1(entry)::T3.1(effect)::S1.3(entry)::T4(effect)::S1.2(entry)
38. T3(effect)::S1(entry)::S1.1(entry)::T3.1(effect)::T4(effect)::S1.3(entry)::S1.2(entry)
39. T3(effect)::S1(entry)::S1.1(entry)::T3.1(effect)::T4(effect)::S1.2(entry)::S1.3(entry)
40. T3(effect)::S1(entry)::S1.1(entry)::T4(effect)::S1.2(entry)::T3.1(effect)::S1.3(entry)
41. T3(effect)::S1(entry)::S1.1(entry)::T4(effect)::T3.1(effect)::S1.2(entry)::S1.3(entry)
42. T3(effect)::S1(entry)::S1.1(entry)::T4(effect)::T3.1(effect)::S1.3(entry)::S1.2(entry)
43. T4(effect)::T3(effect)::S1(entry)::S1.1(entry)::S1.2(entry)::T3.1(effect)::S1.3(entry)

44. T4(effect)::T3(effect)::S1(entry)::S1.1(entry)::T3.1(effect)::S1.2(entry)::S1.3(entry)
45. T4(effect)::T3(effect)::S1(entry)::S1.1(entry)::T3.1(effect)::S1.3(entry)::S1.2(entry)
46. T4(effect)::T3(effect)::S1(entry)::S1.2(entry)::S1.1(entry)::T3.1(effect)::S1.3(entry)
47. T4(effect)::T3(effect)::S1(entry)::S1.2(entry)::T3.1(effect)::S1.3(entry)::S1.1(entry)
48. T4(effect)::T3(effect)::S1(entry)::S1.2(entry)::T3.1(effect)::S1.1(entry)::S1.3(entry)
49. T4(effect)::T3(effect)::S1(entry)::T3.1(effect)::S1.1(entry)::S1.2(entry)::S1.3(entry)
50. T4(effect)::T3(effect)::S1(entry)::T3.1(effect)::S1.1(entry)::S1.3(entry)::S1.2(entry)
51. T4(effect)::T3(effect)::S1(entry)::T3.1(effect)::S1.2(entry)::S1.1(entry)::S1.3(entry)
52. T4(effect)::T3(effect)::S1(entry)::T3.1(effect)::S1.2(entry)::S1.3(entry)::S1.1(entry)
53. T4(effect)::T3(effect)::S1(entry)::T3.1(effect)::S1.3(entry)::S1.1(entry)::S1.2(entry)
54. T4(effect)::T3(effect)::S1(entry)::T3.1(effect)::S1.3(entry)::S1.2(entry)::S1.1(entry)
55. T4(effect)::S1(entry)::T3.1(effect)::T3(effect)::S1.1(entry)::S1.3(entry)::S1.2(entry)
56. T4(effect)::S1(entry)::T3.1(effect)::T3(effect)::S1.1(entry)::S1.2(entry)::S1.3(entry)
57. T4(effect)::S1(entry)::T3.1(effect)::T3(effect)::S1.3(entry)::S1.1(entry)::S1.2(entry)
58. T4(effect)::S1(entry)::T3.1(effect)::T3(effect)::S1.3(entry)::S1.2(entry)::S1.1(entry)
59. T4(effect)::S1(entry)::T3.1(effect)::T3(effect)::S1.2(entry)::S1.1(entry)::S1.3(entry)
60. T4(effect)::S1(entry)::T3.1(effect)::T3(effect)::S1.2(entry)::S1.3(entry)::S1.1(entry)
61. T4(effect)::S1(entry)::T3.1(effect)::S1.3(entry)::T3(effect)::S1.1(entry)::S1.2(entry)
62. T4(effect)::S1(entry)::T3.1(effect)::S1.3(entry)::T3(effect)::S1.2(entry)::S1.1(entry)
63. T4(effect)::S1(entry)::T3.1(effect)::S1.3(entry)::S1.2(entry)::T3(effect)::S1.1(entry)
64. T4(effect)::S1(entry)::T3.1(effect)::S1.2(entry)::T3(effect)::S1.3(entry)::S1.1(entry)
65. T4(effect)::S1(entry)::T3.1(effect)::S1.2(entry)::T3(effect)::S1.1(entry)::S1.3(entry)
66. T4(effect)::S1(entry)::T3.1(effect)::S1.2(entry)::S1.3(entry)::T3(effect)::S1.1(entry)
67. T4(effect)::S1(entry)::T3(effect)::T3.1(effect)::S1.2(entry)::S1.1(entry)::S1.3(entry)
68. T4(effect)::S1(entry)::T3(effect)::T3.1(effect)::S1.2(entry)::S1.3(entry)::S1.1(entry)
69. T4(effect)::S1(entry)::T3(effect)::T3.1(effect)::S1.1(entry)::S1.2(entry)::S1.3(entry)
70. T4(effect)::S1(entry)::T3(effect)::T3.1(effect)::S1.1(entry)::S1.3(entry)::S1.2(entry)
71. T4(effect)::S1(entry)::T3(effect)::T3.1(effect)::S1.3(entry)::S1.2(entry)::S1.1(entry)

72. T4(effect)::S1(entry)::T3(effect)::T3.1(effect)::S1.3(entry)::S1.1(entry)::S1.2(entry)

73. T4(effect)::S1(entry)::T3(effect)::S1.1(entry)::S1.2(entry)::T3.1(effect)::S1.3(entry)

74. T4(effect)::S1(entry)::T3(effect)::S1.1(entry)::T3.1(effect)::S1.3(entry)::S1.2(entry)

75. T4(effect)::S1(entry)::T3(effect)::S1.1(entry)::T3.1(effect)::S1.2(entry)::S1.3(entry)

76. T4(effect)::S1(entry)::T3(effect)::S1.2(entry)::T3.1(effect)::S1.1(entry)::S1.3(entry)

77. T4(effect)::S1(entry)::T3(effect)::S1.2(entry)::T3.1(effect)::S1.3(entry)::S1.1(entry)

78. T4(effect)::S1(entry)::T3(effect)::S1.2(entry)::S1.1(entry)::T3.1(effect)::S1.3(entry)

79. T4(effect)::S1(entry)::S1.2(entry)::T3.1(effect)::S1.3(entry)::T3(effect)::S1.1(entry)

80. T4(effect)::S1(entry)::S1.2(entry)::T3.1(effect)::T3(effect)::S1.3(entry)::S1.1(entry)

81. T4(effect)::S1(entry)::S1.2(entry)::T3.1(effect)::T3(effect)::S1.1(entry)::S1.3(entry)

82. T4(effect)::S1(entry)::S1.2(entry)::T3(effect)::T3.1(effect)::S1.1(entry)::S1.3(entry)

83. T4(effect)::S1(entry)::S1.2(entry)::T3(effect)::T3.1(effect)::S1.3(entry)::S1.1(entry)

84. T4(effect)::S1(entry)::S1.2(entry)::T3(effect)::S1.1(entry)::T3.1(effect)::S1.3(entry)

Consider trace 84. The RTC steps leading to the production of this are presented in the table below.

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T4, T3, T3.1)]
4	[CE(S1.3), CE(S1.1), **CE(S1.2)**]	[S1[S1.1, S1.2, S1.3]]	[T2.2]
5	[CE(S1.3), **CE(S1.1)**]	[S1[S1.1, S1.2]]	[T1.1]
6	[**CE(S1.3)**]	[S1[S1.3]]	[T3.2(T5)]

9.3.11.3 Fork 002

Tested state machine

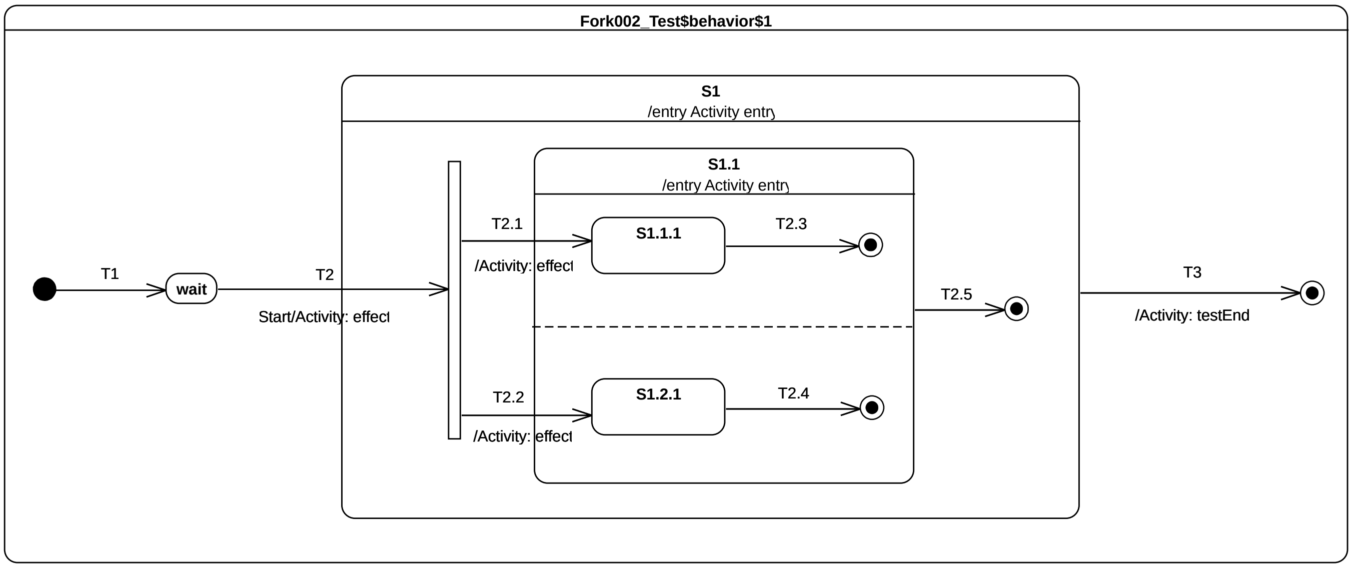


Figure 9.74 - Fork 002 Test Classifier Behavior

The state machine that is executed for this test is presented in Figure 9.74.

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- T2(effect)::S1(entry)::T2.1(effect)::S1.1(entry)::T2.2(effect).

  Note. The purpose of this test is to consolidate fork semantics by evaluating that, if a fork is used in a nested context, the composite state explicit entry rule is preserved. When the *Start* event occurrence is dispatched and accepted by the state machine, *T2* is triggered and its effect behavior is executed. This brings the state machine to the fork pseudostate. As the Fork pseudostate is located within a composite state that is not already active, the latter is entered first. Hence, the *S1* entry behavior is executed. Next, the fork pseudostate outgoing transitions are traversed. The attempt to enter a state that is not already active leads to entering of that state and the execution of its entry behavior. Only at this point can explicit entry of both regions of *S1.1* proceed.

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(T2.1, T2.2)]
4	[CE(1.2.1), **CE(S1.1.1)**]	S1[S1.1[S1.1.1, S1.2.1]]	[T2.3]
5	[**CE(1.2.1)**]	S1[S1.1[S1.2.1]]	[T2.4]
6	[**CE(S1.1)**]	[S1[S1.1]]	[T2.5]
7	[**CE(S1)**]	[S1]	[T3]

Alternative execution traces

The presence of the fork pseudostate and the two orthogonal regions of *S1.1* imply the existence of alternative execution traces. These traces are listed below:

1. T2(effect)::S1(entry)::T2.1(effect)::T2.2(effect)::S1.1(entry)
2. T2(effect)::S1(entry)::T2.2(effect)::T2.1(effect)::S1.1(entry)
3. T2(effect)::S1(entry)::T2.2(effect)::S1.1(entry)::T2.1(effect)

Lets consider the trace 3 which shows that *T2.1* is fired fired after *T2.2*. The RTC steps leading to the production of this trace are described below:

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

9.3.12 Join

9.3.12.1 Overview

Test presented in this subclause assess that join semantics conform to what is specified in UML.

9.3.12.2 Join 001

Tested state machine

The state machine that is executed for this test is presented in Figure 9.75.

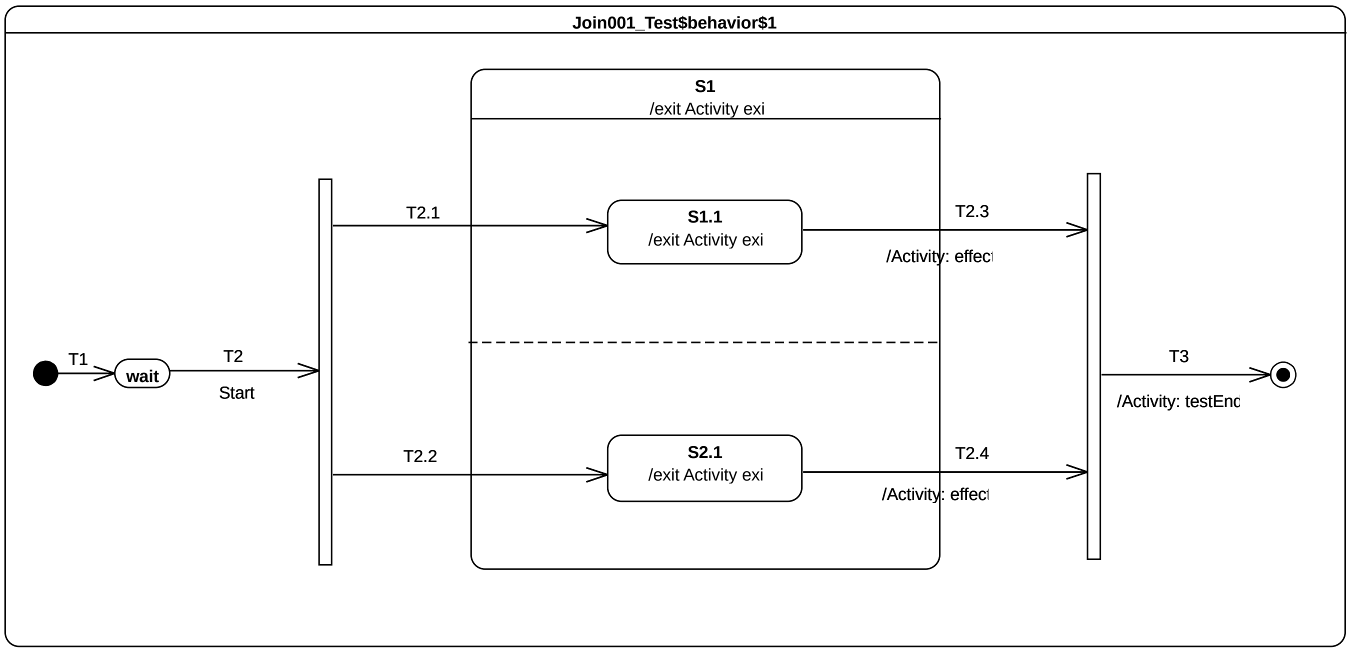


Figure 9.75 - Join 001 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- S1.1(exit)::T2.3(effect)::S2.1(exit)::S1(exit)::T2.4(effect)

Note. The purpose of this test is to demonstrate that the join pseudostate can only be traversed when all incoming transitions have been traversed. Consider the situation where the state machine is currently in configuration *S1[S1.1, S2.1]*. Two completion event occurrences (one for *S1.1* and the other one for *S2.1*) are available in the pool. When the completion event occurrence generated by *S1.1* is dispatched and accepted, it triggers *T2.3*. Next, *S1.1* is exited, the effect behavior of *T2.3* is executed, but *S1* is not exited and the join pseudostate is not traversed. The next step consists in accepting the *S2.1* completion event occurrence. This means that *T2.4* is triggered, so that *S2.1* is exited and *T2.4* is executed. In addition, *S1* is exited and the join pseudo state is traversed. The continuation transition *T3* is traversed. 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(T2.1, T2.2)]
4	[CE(S2.1), **CE(S1.1)**]	[S1[S1.1, S2.1]]	[T2.3]
5	[**CE(S2.1)**]	[S1[S2.1]]	[T2.4(T3)]

Alternative execution traces

The presence of fork pseudostate and the orthogonal regions in *S1.1* implies that an alternative execution trace is possible for this test case. This trace is:

- S1.2(exit)::T2.4(effect)::S1.1(exit)::S1(exit)::T2.4(effect)

The trace shows that *T2.4* is fired before *T2.3*. The RTC steps leading to the production of this trace are described in the table below.

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	**[]**	**[] - Initial RTC step**	**[T1]**
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T2.2, T2.1)]
4	[CE(S1.1), **CE(S2.1)**]	[S1[S1.1, S2.1]]	[T2.4]
5	[**CE(S1.1)**]	[S1[S1.1]]	[T2..3(T3)]

9.3.12.3 Join 002

Tested state machine

The state machine that is executed for this test is presented in Figure 9.76.

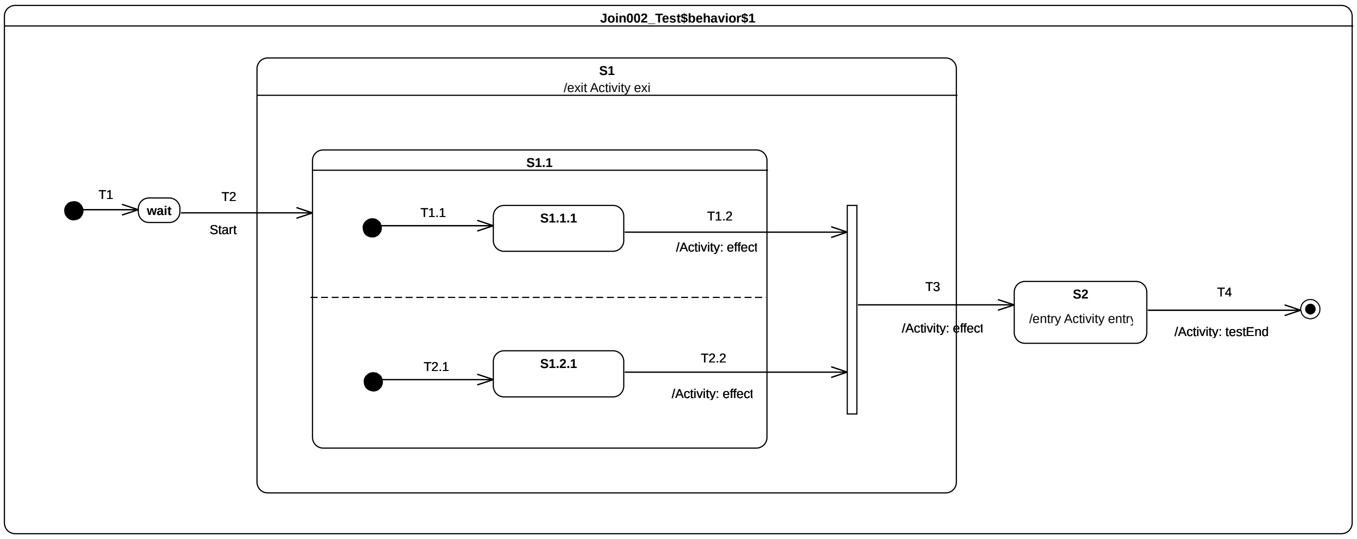


Figure 9.76 - Join 002 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- T2.2(effect)::T1.2(effect)::S1(exit)::T3(effect)::S2(entry)

Note. The purpose of this test is to consolidate join semantics and demonstrate that, if used in a nested context, the exit rule of a composite state is still preserved. In addition, it shows that the composite state is only exited when all transitions leaving the internal vertices located in different orthogonal regions have been traversed. Consider the situation where the state machine is in configuration *S1[S1.1[S1.1.1]]*. The completion event occurrence generated by *S1.1.1* is dispatched and accepted. Next, *T1.2* is traversed, its effect behavior is executed, *S1.1* is exited, and the join pseudostate is reached. All incoming transitions have been fired so the join pseudostate can be traversed. When continuation transition *T3* is traversed, *S1* is exited, the effect behavior of the transition is executed, and, finally, *S2* is entered.

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, T2.1)]
4	[CE(S1.2.1), **CE(S1.1.1)**]	[S1[S1.1[S1.1.1, S1.2.1]]]	[T1.2]
5	[**CE(S1.2.1)**]	[S1[S1.1[S1.2.1]]]	[T2.2(T3)]

6	[**CE(S2)**]	[S2]	[T4]

Alternative execution traces

The presence of orthogonal regions in *S1.1* implies that an alternative execution trace is possible for this test case. This trace is:

- T1.2(effect)::T2.2(effect)::S1(exit)::T3(effect)::S2(entry)

The trace shows the situation where *T1.2* is fired after *T2.2*. The RTC steps leading to the production of this trace are described in the table below.

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	**[]**	**[] - Initial RTC step**	**[T1]**
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T2.1, T1.1)]
4	[CE(S1.1.1), **CE(S1.2.1)**]	[S1[S1.1[S.1.1.1, S1.2.1]]]	[T2.2]
5	[**CE(S1.1.1)**]	[S1[S1.1[S1.1.1]]]	[T1.2(T3)]
6	[CE(S2)]	[S2]	[T4]

9.3.12.4 Join 003

Tested state machine

The state machine that is executed for this test is presented in Figure 9.77.

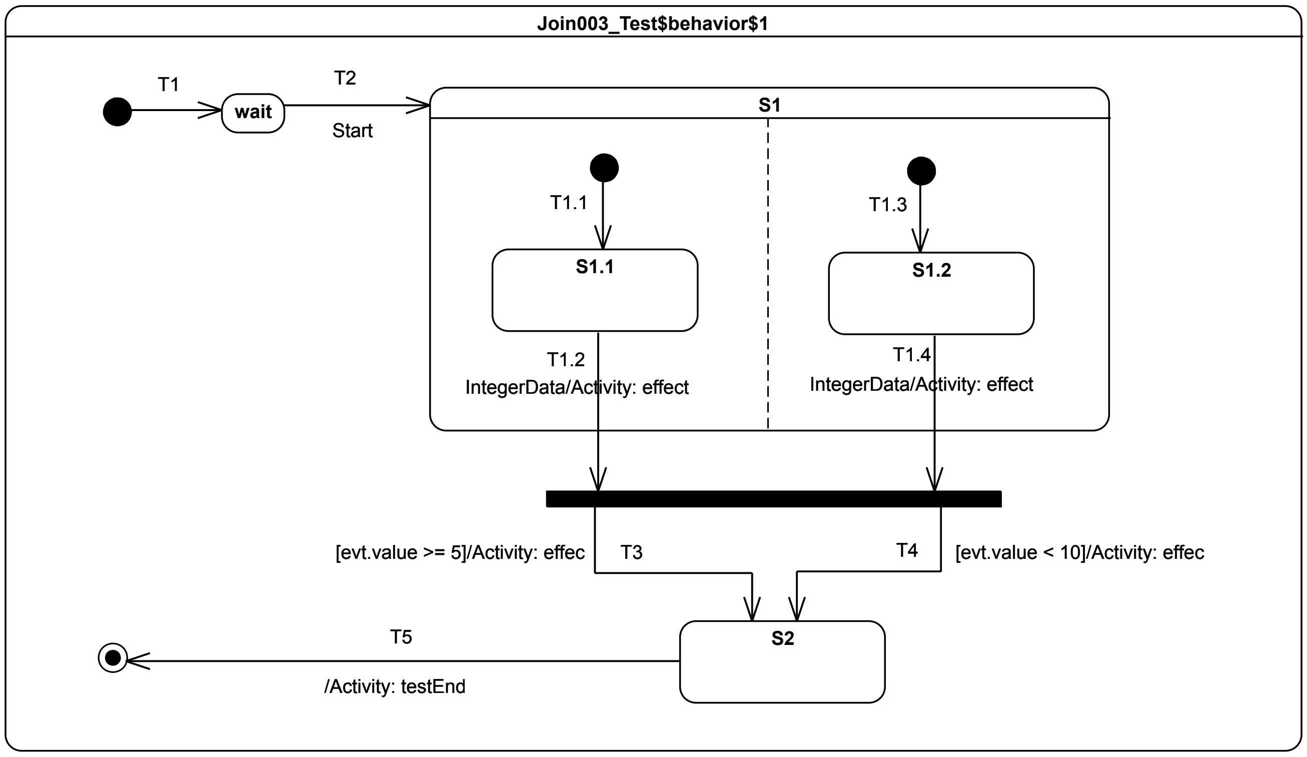

Join003_Test$behavior$1
T1
wait
T2
Start
S1
T1.1
S1.1
T1.3
S1.2
T1.2
IntegerData/Activity: effect
T1.4
IntegerData/Activity: effect
[evt.value >= 5]/Activity: effec
T3
T4
[evt.value < 10]/Activity: effec
T5
/Activity: testEnd
S2

Figure 9.77 - Join 003 - Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- IntegerData(8) – received when in configuration *S1.1*.

Generated trace

- T1.2(effect)[in=8]::T1.4(effect)[in=8]::T3(effect)[in=8]

Note. The test is focused on the application of the static analysis when a join pseudostate is encountered in a compound transition. It also shows triggering of multiple transitions located in different regions (see *T1.2* and *T1.4*) using the same event occurrence and conflict resolution (see *T3* and *T4*). In this test case, when a Start event occurrence is about to be dispatched, the static analysis perceives two possible compound transitions ([*T1.2*, *T1.4(T4)*] and [*T1.2*, *T4(T3)*]) leading to the same state machine configuration. During the RTC step initiated by the Start event occurrence, one of them is traversed to reach this state machine configuration. Note the conflict between *T3* and *T4*. This conflict is resolved by a semantic strategy that chooses the first transition available in the set of enabled transitions. For this test case, *T3* is always chosen. The last RTC step is realized when the completion event for *S2* is dispatched.

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, T1.3)]
4	[IntegerData(8), CE(S1.2), **CE(S1.1)**]	[S1[S1.1, S1.2]]	[]
5	[IntegerData(8), **CE(S1.2)**]	[S1[S1.1, S1.2]]	[]
6	[**IntegerData(8)**]	[S1[S1.1, S1.2]]	[T1.2, T1.4(T3)]
7	[**CE(S2)**]	[S2]	[T5]

Alternative execution traces

The orthogonal regions existing in S1 imply the existence of an alternative execution trace. This trace is:

1. T1.4(effect)[in=8]::T1.2(effect)[in=8]::T3(effect)[in=8]
2. T1.4(effect)[in=8]::T1.2(effect)[in=8]::T4(effect)[in=8]
3. T1.2(effect)[in=8]::T1.4(effect)[in=8]::T4(effect)[in=8]

The trace 1 shows the situation where *T1.4* is fired before *T1.2*. The RTC steps leading to the production of this trace are described in the table below.

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	**[]**	**[] - Initial RTC step**	**[T1]**
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T1.3, T1.1)]
4	[IntegerData(8), CE(S1.1), **CE(S1.2)**]	[S1[S1.1, S1.2]]	[]
5	[IntegerData(8), **CE(S1.1)**]	[S1[S1.1, S1.2]]	[]
6	[**IntegerData(8)**]	[S1[S1.1, S1.2]]	[T1.4, T1.2(T3)]
7	[**CE(S2)**]	[S2]	[T5]

9.3.13 Terminate

9.3.13.1 Overview

Tests presented in this subclause assess that terminate semantics conform to what is specified in UML.

9.3.13.2 Terminate 001

Tested state machine

The state machine that is executed for this test is presented in Figure 9.78.

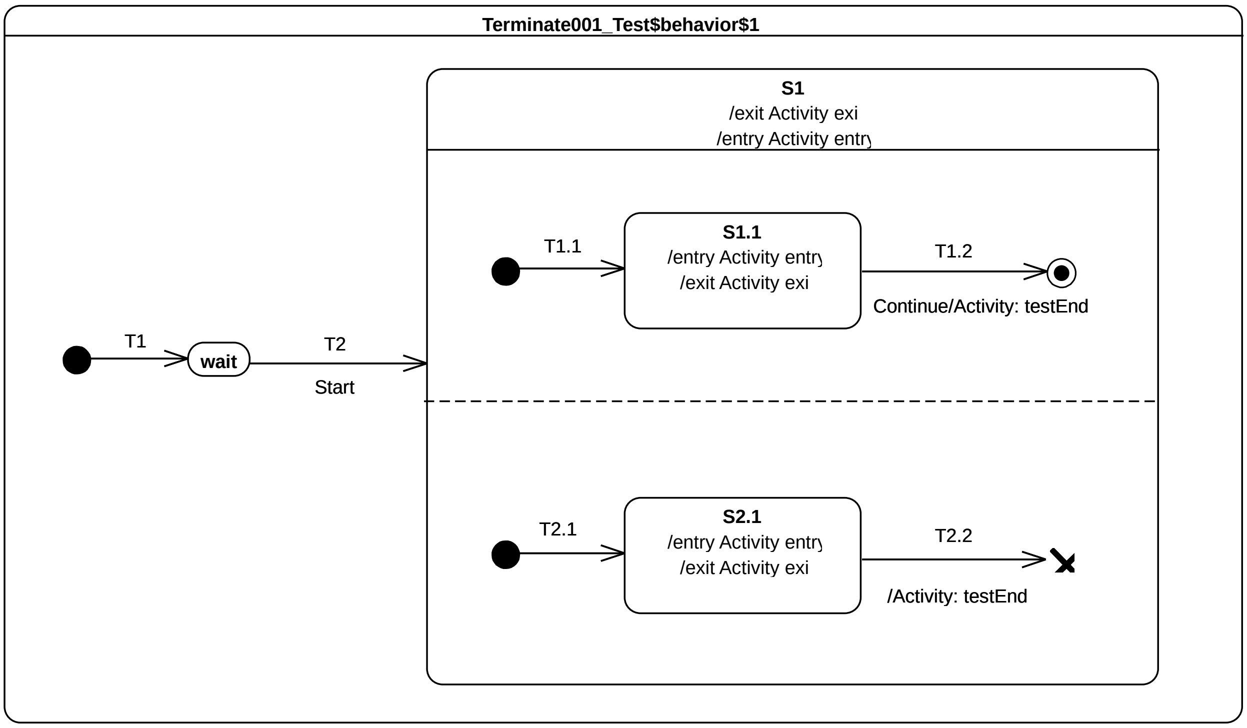


Figure 9.78 - Terminate 001 Test Classifier behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- S1(entry)::S1.1(entry)::S2.1(entry)::S2.1(exit)

Note. The purpose of this test is to demonstrate support for terminate semantics. It especially shows that, when the terminate pseudostate is entered, the state machine execution terminates and no exit behavior is executed. Consider the situation where the state machine is in configuration *S1[S1.1, S2.1]*. Two completion events are in the event pool, one of for *S2.1* and the other one for *S1.1*. When the *S2.1* completion event is accepted, it triggers *T2.2*. The traversal of this transition leads the state machine to reach the terminate pseudostate. No state can be exited when the terminate pseudostate is entered since the state machine terminates its execution. This behavior can be observed in the generated trace. Indeed neither *S1.1* nor *S1* have executed their exit behaviors after the execution of the terminate pseudostate.

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, T2.1)]
4	[CE(S2.1), **CE(S1.1)**]	[S1[S1.1, S2.1]]	[]
5	[**CE(2.1)**]	[S1[S1.1, S2.1]]	[T2.2]

Alternative execution traces

The orthogonal regions existing in *S1* implies the existence of an alternative execution trace. This trace is:

- S1(entry)::S2.1(entry)::S1.1(entry)::S2.1(exit)

The trace shows the situation where *S2.1* is entered before *S1.1*. Such execution implies that the *S1.1* completion event will never be dispatched because the *S2.1* completion event triggers *T2.2* which implies the termination of the state machine. The RTC steps leading to the production of this trace are described in the table below.

Step	Event pool	State machine configuration	Fired transition(s)
1	**[]**	**[] - Initial RTC step**	**[T1]**
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T2.1, T1.1)]
4	[CE(S1.1), **CE(S2.1)**]	[S1[S1.1, S2.1]]	T2.2

9.3.13.3 Terminate 002

Tested state machine

The state machine that is executed for this test is presented in Figure 9.79.

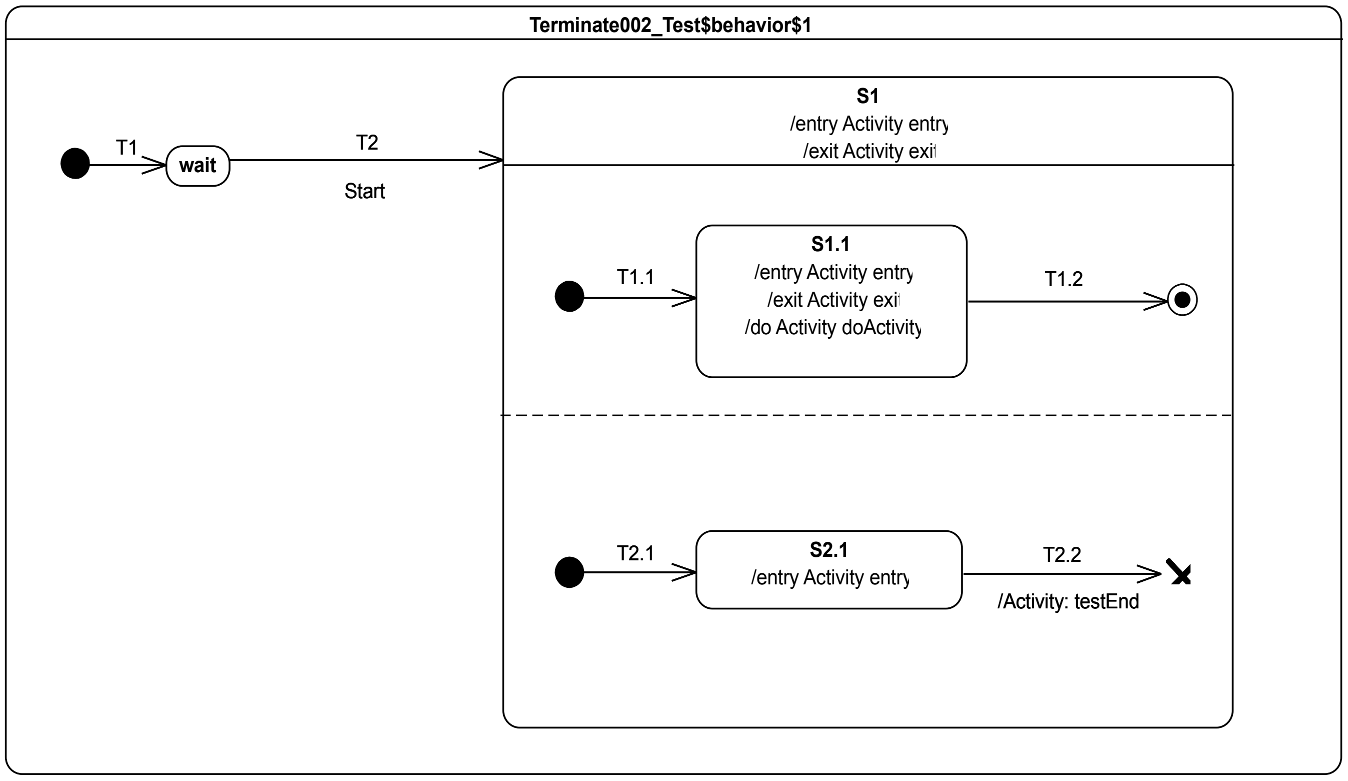


Figure 9.79 - Terminate 002 Test Classifier Behavior

The behavior specified for the doActivity for *S1.1* is presented in Table 9.3. It contributes to the trace by producing the fragment *S1.1doActivityPartI* and waits for a *Continue* event occurrence. This occurrence is never available during the test, therefore the doActivity remains suspended until it is aborted.

```
activity doActivity() {
     this.trace("S1.1(doActivityPartI)");
     accept(Continue);
     this.trace("S1.1(doActivityPartII)");
}
```

Table 9.3 - S1.1 doActivity Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- S1(entry)::S1.1(entry)::S1.1(doActivityPartI)::S2.1(entry)

Note. The purpose of this test is to demonstrate that running doActivity behaviors are aborted if a terminate pseudostate is reached. At the end of the RTC step initiated by the acceptance of the *Start* event occurrence, the state machine is in configuration *S1[S1.1, S2.1]*. The doActivity behavior of *S1.1* has been invoked (which automatically implies it has already started) and a single completion event occurrence is waiting in the pool, the one for *S2.1.* For this execution, it is assumed that the *S2.1* entry behavior is executed after the *S1.1* entry behavior and also after the invoked doActivity has contributed to the trace. When the completion event occurrence generated by *S2.1* is dispatched, then *T2.2* is traversed and the terminate pseudostate is reached. This implies that the execution of the running doActivity is aborted, the state machine execution terminates and no exit behavior is executed. Note that, in this test, *S1.1* can never complete, since the doActivity behavior invoked from that state cannot complete.

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, T2.1)]
4	[**CE(S2.1)**]	[S1[S1.1, S2.1]]	[T2.2]

Alternative execution traces

The two orthogonal regions of *S1* and the doActivity behaviors specified on *S1.1* imply the existence of alternative execution traces. These traces are listed below.

1. S1(entry)::S1.1(entry)::S2.1(entry)::S1.1(doActivityPartI)"
2. S1(entry)::S2.1(entry)::S1.1(entry)::S1.1(doActivityPartI)"
3. S1(entry)::S1.1(entry)::S2.1(entry)
4. S1(entry)::S2.1(entry)::S1.1(entry)

Consider trace 4. It shows the situation where the *S2.1* entry behavior gets executed before the *S1.1* entry behavior and the *S1.1* doActivity has not contributed to the trace before the state machine execution terminates.

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T2.1, T1.1)]
4	[**CE(S2.1)**]	[S1[S1.1, S2.1]]	[T2.2]

Here the *S1.1* doActivity is aborted before it has actually completed its initial first RTC step.

9.3.13.4 Terminate 003

Tested state machine

The state machine that is executed for this test is presented in Figure 9.80.

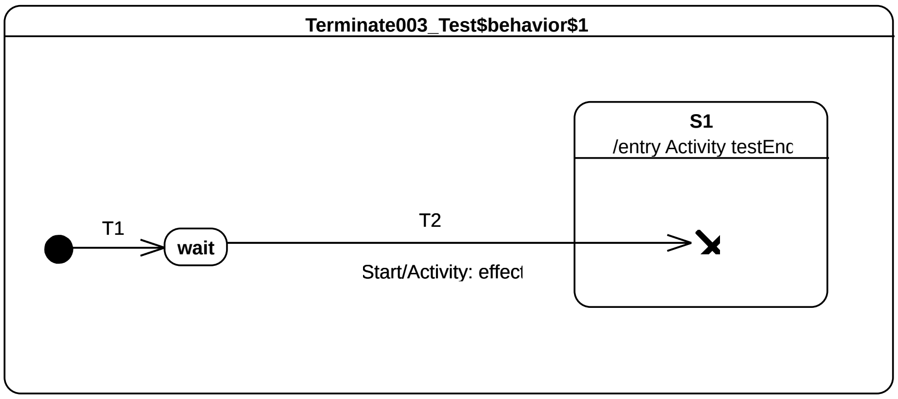


Figure 9.80 - Terminate 003 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- T2(effect)

Note. The purpose of this test is to ensure that, when a terminate pseudostate is used in a nested context, the composite state entry rule is preserved. When the *Start* event occurrence is dispatched and accepted by the state machine, *T2* is triggered. Next, *wait* is exited, the effect behavior of the transition is executed, *S1* is entered, and, finally, the terminate pseudostate is reached. The execution of this pseudostate implies the termination of the state machine. Note that the entry behavior for *S1* is *testEnd,* which registers the ending of the test (but is not shown explicitly in the trace). If the entry behavior *testEnd* of *S1* was not executed, then the *SemanticTest* object for this test would not have been notified of the termination of the test target, and, hence, the test would not have been considered as having passed.

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]

9.3.14 Final

9.3.14.1 Overview

Tests presented in this subclause assess that final-state semantics conform to what is specified in UML.

9.3.14.2 Final 001

Tested state machine

The state machine that is executed for this test is presented in Figure 9.81.

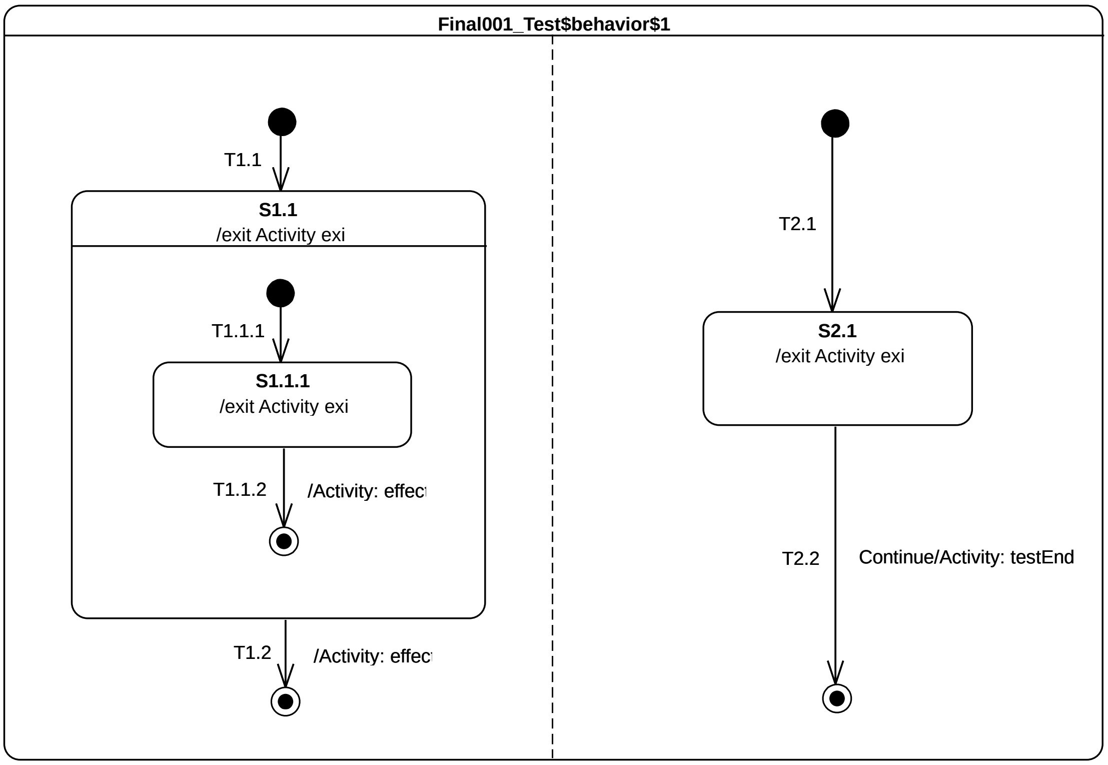


Figure 9.81 - Final 001 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration [S1[S1.1.1], S2.1].

Generated trace

- S1.1.1(exit)::T1.1.2(effect)::S1.1(exit)::T1.2(effect)::S2.1(exit)

Note. The purpose of this test is to demonstrate support for the final state, both at the state machine and composite state levels. It especially shows that, when the final state is executed, the containing region completes, regardless of its container (that is, either a state machine or a composite state). When the state machine starts its execution, both regions start their executions concurrently. The initial RTC step ends up with the following configuration: *[S1.1[S1.1.1], S2.1]*. At this point, there are two completion event occurrences placed in the pool. The completion event occurrences were generated when *S2.1* and *S1.1.1* were entered. Assuming that the *S2.1* completion event is at the head of the event pool, it is dispatched first. However, it does not initiate an RTC step. Since *S2.1* has no completion transition the completion event occurrence is lost. The next RTC step consists of dispatching and accepting the *S1.1.1* completion event occurrence. It triggers traversal of *T1.1.2* and leads the state machine to reach the final state located in the *S1.1* region. This region completes and a completion event occurrence is generated for *S1.1*. This completion event occurrence is dispatched and accepted in the next RTC step. It triggers *T1.2* so that the final state is reached, and the left-hand region of the state machine completes. When the *Continue* event occurrence is dispatched and accepted, *T2.2* is triggered. The state machine execution completes when the target final state is reached.

RTC Steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T1.1(T1.1.1), T2.1]
4	[Continue, CE(S2.1), **CE(S1.1.1)**]	[S1.1[S1.1.1], S2.1]	[T1.1.2]
5	[Continue, CE(S1.1), **CE(S2.1)**]	[S1.1, S2.1]	[]
6	[Continue, **CE(S1.1)**]	[S1.1, S2.1]	[T1.2]
7	[**Continue**]	[S2.1]	[T2.2]

9.3.15 History

9.3.15.1 Overview

Tests presented in this subclause assess that history pseudostate semantics conform to what is specified in UML.

9.3.15.2 History 001-A

Tested state machine

The state machine that is executed for this test is presented in Figure 9.82.

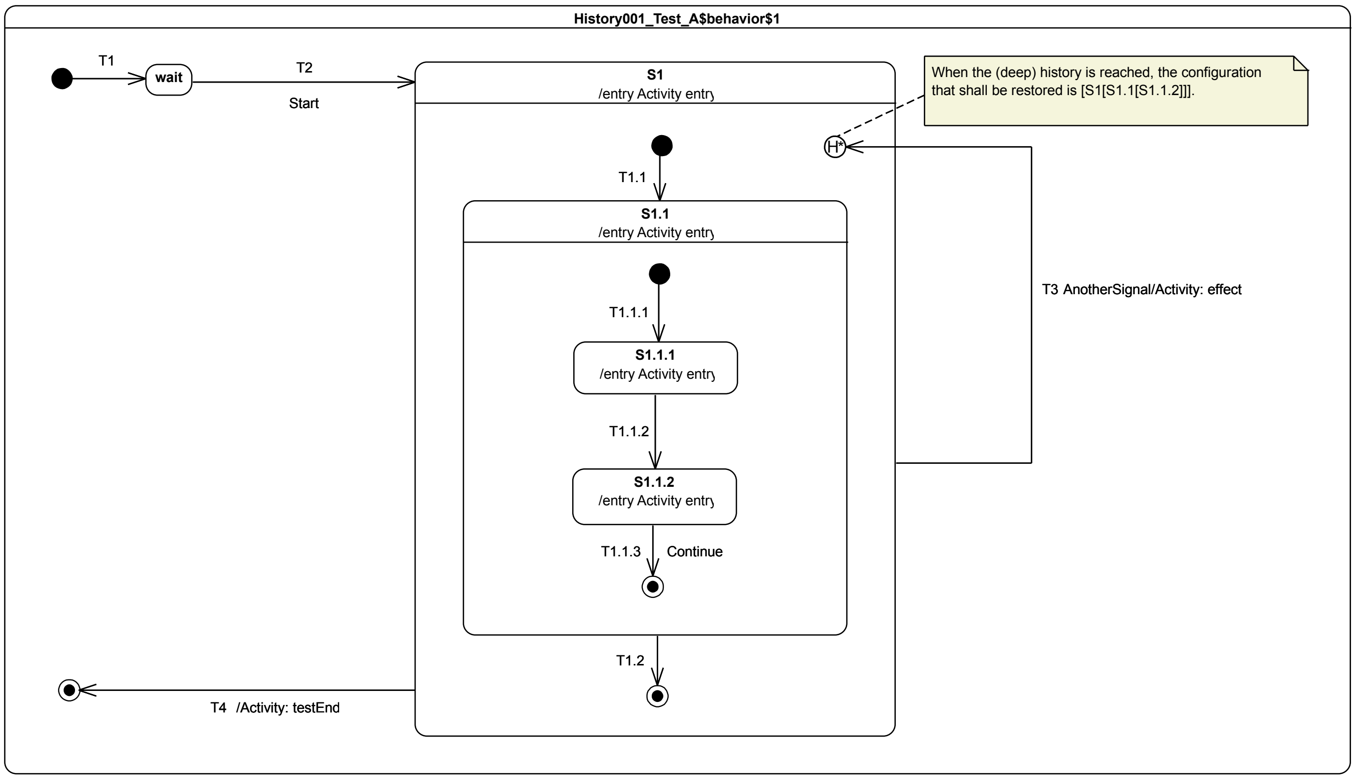


Figure 9.82 - History 001 - A Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- AnotherSignal – received when in configuration *S1[S1.1[S1.1.2]]*.
- Continue – received when in configuration *S1[S1.1[S1.1.2]]*.

Generated trace

- S1(entry)::S1.1(entry)::S1.1.1(entry)::S1.1.2(entry)::T3(effect)::S1(entry)::S1.1(entry)::S1.1.2(entry)

Note. The purpose of this test is to demonstrate that, when a deep-history pseudostate is reached and the region in which it is placed already has a history, then the full configuration corresponding to this pseudostate is restored. Consider the situation where the state machine is in configuration *S1[S1.1[S1.1.2]]*. When the event occurrence for *AnotherSignal* is dispatched, *T3* is fired. This means that *S1.1.2*, *S1.1*, and *S1* are exited (in that order) before the history pseudostate is reached. At that point, *S1* is entered and the process of restoring the state-machine configuration based on history occurs. In this case, *S1.1* and *S1.1.2* are restored. When these states are restored, their entry behaviors are executed. Note that, in this process, states are entered directly without going through transitions.

RTC Steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[IntialTransition]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T1.1(T1.1.1))]
4	[**CE(S1.1.1)**]	[S1[S1.1[S1.1.1]]]	[T1.1.2]
5	[AnotherSignal, **CE(S1.1.2)**]	[S1[S1.1[S1.1.2]]]	[]
6	[**AnotherSignal**]	[S1[S1.1[S1.1.2]]]	[T3]
7	[Continue, **CE(S1.1.2)**]	[S1[S1.1[S1.1.2]]]	[]
8	[**Continue**]	[S1[S1.1[S1.1.2]]]	[T1.1.3]
9	[**CE(S1.1)**]	[S1[S1.1]]	[T1.2]
10	[**CE(S1)**]	[S1]	[T4]

9.3.15.3 History 001-B

Tested state machine

The state machine that is executed for this test is presented in Figure 9.83.

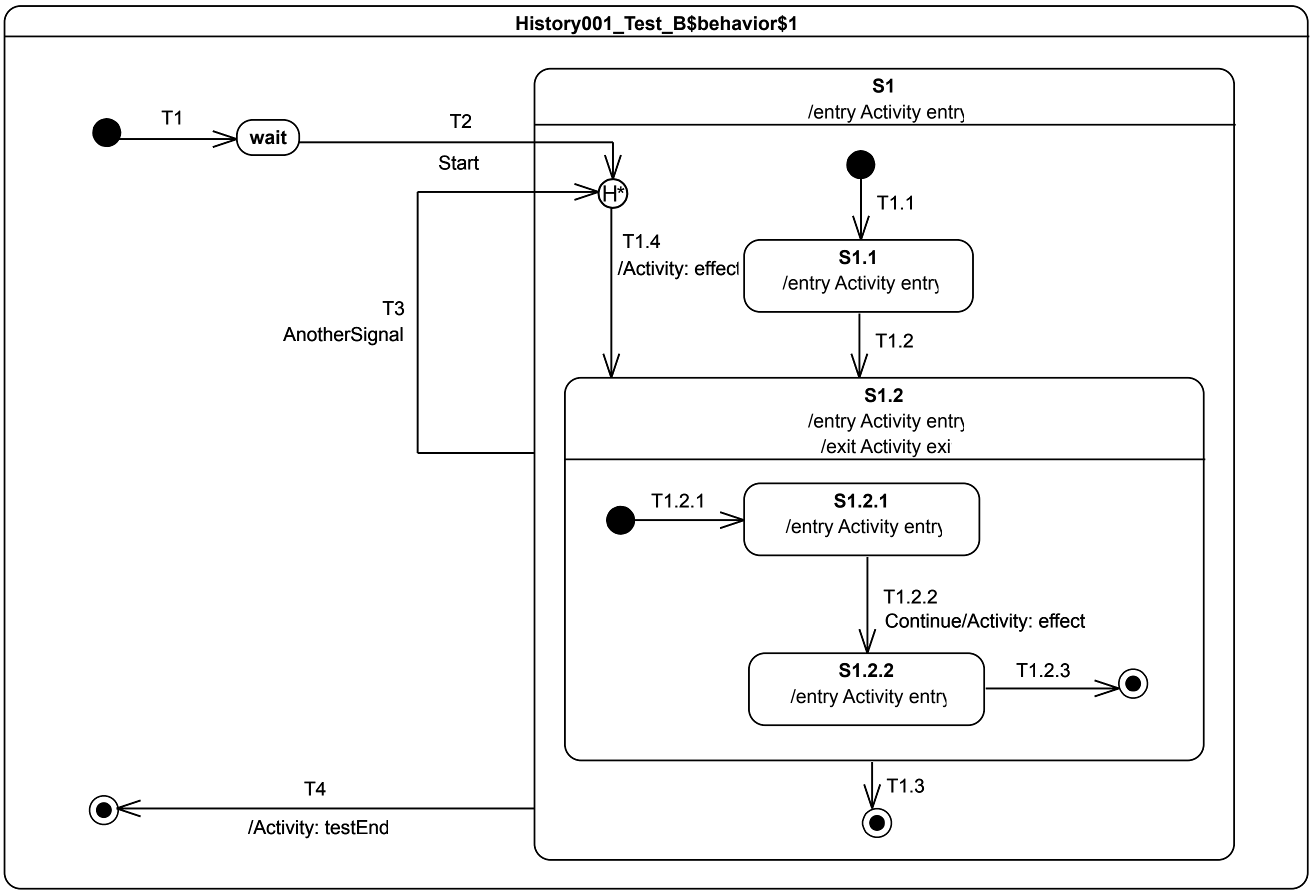


Figure 9.83 - History 001 - B Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- AnotherSignal – received when in configuration *S1[S1.2[S1.2.1]]*.
- Continue – received when in configuration *S1[S1.2[S1.2.1]]*.

Generated trace

- S1(entry)::T1.4(effect)::S1.2(entry)::S1.2.1(entry)::S1.2(exit)::S1(entry)::S1.2(entry)::S1.2.1(entry)::T1.2.2(effect)::S1.2.2(entry)::S1.2(exit)

Note. The purpose of this test is to demonstrate that, if a deep-history pseudostate is entered but the region which contains it has no prior history, then, if the pseudostate has a default transition (i.e., an outgoing transition), this transition is taken. Consider the situation where the state machine is in configuration *wait*. When the event occurrence for the *Start* signal is dispatched, *T2* fires. *S1* is entered, but the deep history pseudo state is in a region that, at this point, has no history. Therefore, transition *T1.4*, which is the pseudostate's default transition, is fired and state *S1.2* is entered. At that point, *S1.2* executes its entry behavior and the execution of its region begins. The RTC step ends with the state machine in the configuration *S1[S1.2[S1.1.2]]*. When *AnotherSignal* event occurrence is dispatched, *S1.2.1*, *S1.2*, and *S1* are exited. During that step, the history pseudostate is entered again. But, this time, the region has a history. Hence, the state hierarchy *S1[S1.1.2]]* is restored and *T1.4* is not used.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[IntialTransition]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T1.4(T1.2.1))]
4	[AnotherSignal, **CE(S1.2.1)**]	[S1[S1.2[S1.2.1]]]	[]
5	[**AnotherSignal**]	[S1[S1.2[S1.2.1]]]	[T3]
6	[Continue, **CE(S1.2.1)**]	[S1[S1.2[S1.2.1]]]	[]
7	[**Continue**]	[S1[S1.2[S1.2.1]]]	[T1.2.2]
8	[**CE(S1.2.2)**]	[S1[S1.2[S1.2.2]]]	[T1.2.3]
9	[**CE(S1.2)**]	[S1[S1.2]]	[T1.3]
10	[**CE(S1)**]	[S1]	[T4]

9.3.15.4 History 001-C

Tested state machine

The state machine that is executed for this test is presented in Figure 9.84.

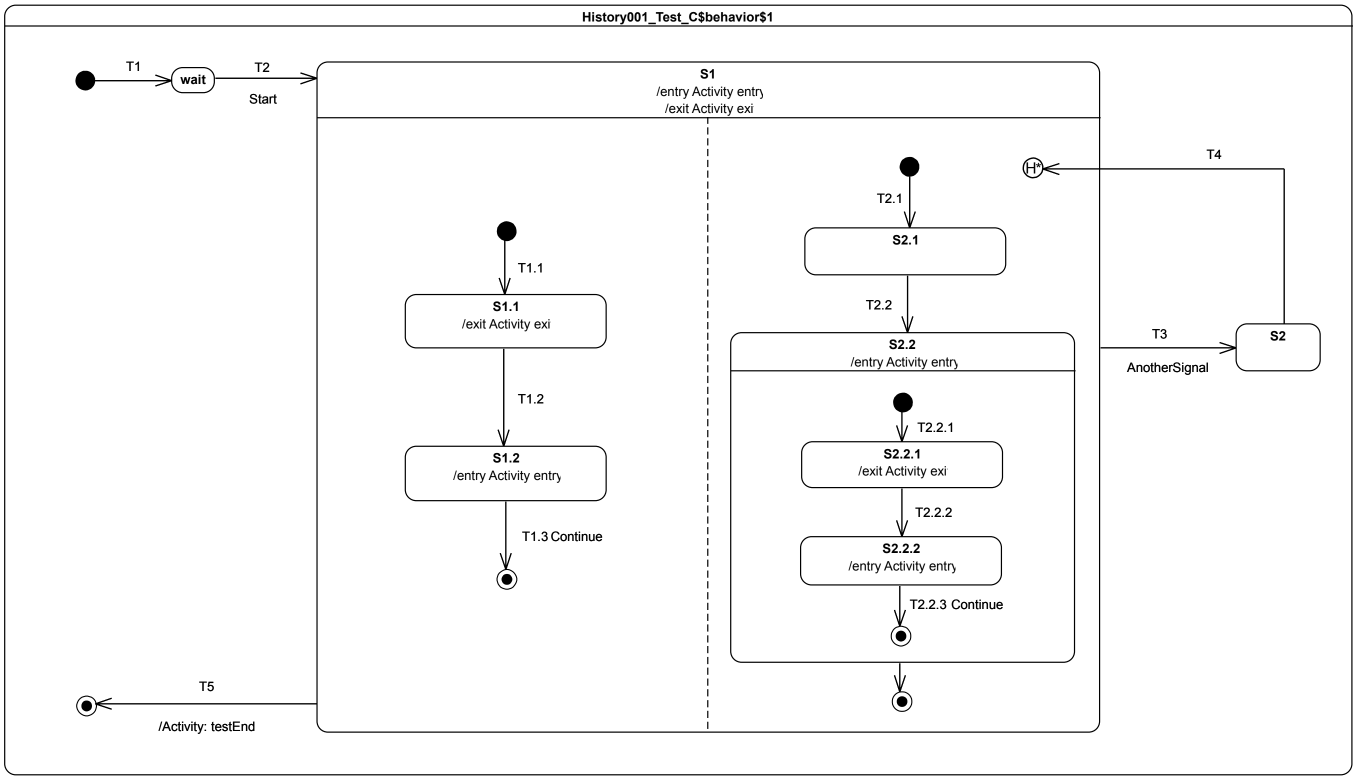


Figure 9.84 - History 001 - C Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- AnotherSignal – received when in configuration *S1[S1.2, S2.2[S2.2.2]]*.
- Continue – received when in configuration *S1[S1.2, S2.2[S2.2.2]]*.

Generated trace

- S1(entry)::S1.1(exit)::S1.2(entry)::S2.2(entry)::S2.2.1(exit)::S2.2.2(entry)::S1(exit)::S1(entry)::S2.2(entry):: S2.2.2(entry)::S1.1(exit)::S1.2(entry)::S1(exit)

Note. The purpose of this test is to demonstrate that, when orthogonal regions are involved, only the region that contains the deep history pseudo state is affected by the restoration process. Consider the situation where the state machine is in configuration *S1[S1.2, S2.2[S2.2.2]]*. When the *AnotherSignal* event occurrence is dispatched, the complete state hierarchy is exited, starting with the innermost active states (*S1.2* and *S2.2.2*) of both regions. *T3* is traversed and *S2* is entered. The completion event occurrence generated by *S2* is dispatched during the next RTC step and triggers *T4*. This means that *S1* is entered and the deep-history pseudostate is reached. As a result, the left region starts its execution from the initial pseudostate while the right region is restored to its last recorded configuration (i.e., *S1[S1.2, S2.2[S2.2.2]]*). This completes the step started by the firing of *T4*. When dispatched, the completion event occurrence generated by *S1.1* triggers *T1.2*. At this point, the state machine is in configuration *S1[S1.2, S2.2[S2.2.2]].* The dispatching of the *Continue* event occurrence forces the firing of both *T1.3* and *T2.2.3*. The completion event occurrence generated by *S2.2* is then used to trigger its outgoing transition. This enables *S1* to complete and to produce a new completion event occurrence that is used to trigger *T5*.

RTC Steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T1.1, T2.1)]
4	[CE(S2.1), **CE(S1.1)**]	[S1[S1.1, S2.1]]	[T1.2]
5	[CE(S1.2), **CE(S2.1)**]	[S1[S1.2, S2.1]]	[T2.2(T2.2.1)]
6	[CE(S2.2.1), **CE(S1.2)**]	[S1[S1.2, S2.2[S2.2.1]]]	[]
7	[**CE(S2.2.1)**]	[S1[S1.2, S2.2[S2.2.1]]]	[T2.2.2]
8	[AnotherSignal, **CE(S2.2.2)**]	[S1[S1.2, S2.2[S2.2.2]]]	[]
9	[**AnotherSignal**]	[S1[S1.2, S2.2[S2.2.2]]]	[T3]
10	[**CE(S2)**]	[S2]	[T4]
11	[CE(S1.1), **CE(S2.2.2)**]	[S1[S1.1, S2.2[S2.2.2]]]	[]
12	[**CE(S1.1)**]	[S1[S1.2, S2.2[S2.2.2]]]	[T1.2]
13	[Continue, **CE(S1.2)**]	[S1[S1.2, S2.2[S2.2.2]]]	[]
14	[**Continue**]	[S1[S1.2, S2.2[S2.2.2]]]	[T1.3, T2.2.3]
15	[**CE(S2.2)**]	[S1[S2.2]]	[T2.3]
16	[**CE(S1)**]	[S1]	[T5]

Alternative execution traces

The presence of orthogonal regions in S1 implies the possibility of different valid traces. These are listed below.

1. S1(entry)::S1.1(exit)::S1.2(entry)::S2.2(entry)::S2.2.1(exit)::S2.2.2(entry)::S1(exit)::S1(entry)::S2.2(entry)::S1.1(exit)::S2.2.2(entry)::S1.2(entry)::S1(exit)
2. S1(entry)::S1.1(exit)::S1.2(entry)::S2.2(entry)::S2.2.1(exit)::S2.2.2(entry)::S1(exit)::S1(entry)::S2.2(entry)::S1.1(exit)::S1.2(entry)::S2.2.2(entry)::S1(exit)
3. S1(entry)::S1.1(exit)::S1.2(entry)::S2.2(entry)::S2.2.1(exit)::S2.2.2(entry)::S1(exit)::S1(entry)::S1.1(exit)::S2.2(entry)::S2.2.2(entry)::S1.2(entry)::S1(exit)

4. S1(entry)::S1.1(exit)::S1.2(entry)::S2.2(entry)::S2.2.1(exit)::S2.2.2(entry)::S1(exit)::S1(entry)::S1.1(exit)::S2.2(entry)::S1.2(entry)::S2.2.2(entry)::S1(exit)

5. S1(entry)::S1.1(exit)::S1.2(entry)::S2.2(entry)::S2.2.1(exit)::S2.2.2(entry)::S1(exit)::S1(entry)::S1.1(exit)::S1.2(entry)::S2.2(entry)::S2.2.2(entry)::S1(exit)

6. S1(entry)::S2.2(entry)::S1.1(exit)::S1.2(entry)::S2.2.1(exit)::S2.2.2(entry)::S1(exit)::S1(entry)::S2.2(entry)::S2.2.2(entry)::S1.1(exit)::S1.2(entry)::S1(exit)

7. S1(entry)::S2.2(entry)::S1.1(exit)::S1.2(entry)::S2.2.1(exit)::S2.2.2(entry)::S1(exit)::S1(entry)::S2.2(entry)::S1.1(exit)::S2.2.2(entry)::S1.2(entry)::S1(exit)

8. S1(entry)::S2.2(entry)::S1.1(exit)::S1.2(entry)::S2.2.1(exit)::S2.2.2(entry)::S1(exit)::S1(entry)::S2.2(entry)::S1.1(exit)::S1.2(entry)::S2.2.2(entry)::S1(exit)

9. S1(entry)::S2.2(entry)::S1.1(exit)::S1.2(entry)::S2.2.1(exit)::S2.2.2(entry)::S1(exit)::S1(entry)::S1.1(exit)::S2.2(entry)::S2.2.2(entry)::S1.2(entry)::S1(exit)

10. S1(entry)::S2.2(entry)::S1.1(exit)::S1.2(entry)::S2.2.1(exit)::S2.2.2(entry)::S1(exit)::S1(entry)::S1.1(exit)::S2.2(entry)::S1.2(entry)::S2.2.2(entry)::S1(exit)

11. S1(entry)::S2.2(entry)::S1.1(exit)::S1.2(entry)::S2.2.1(exit)::S2.2.2(entry)::S1(exit)::S1(entry)::S1.1(exit)::S1.2(entry)::S2.2(entry)::S2.2.2(entry)::S1(exit)

9.3.15.5 History 001-D

Tested state machine

The state machine that is executed for this test is presented in Figure 9.85.

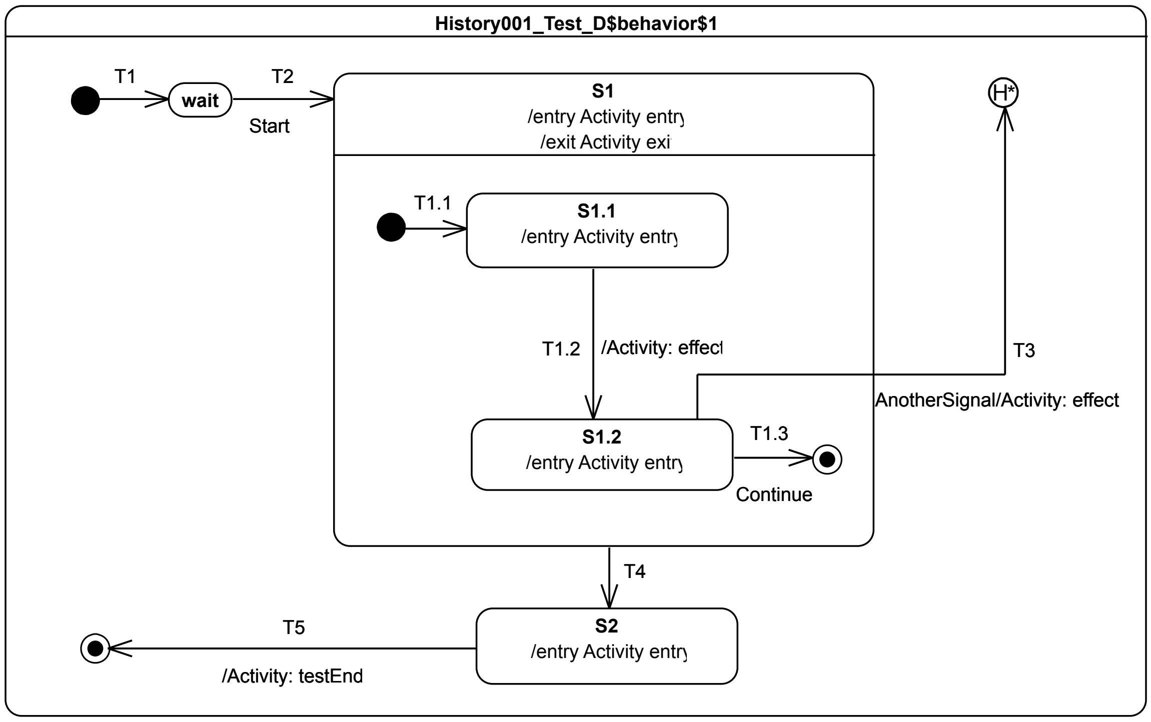


Figure 9.85 - History 001 - D Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- AnotherSignal – received when in configuration *S1[S1.2]*.
- Continue – received when in configuration *S1[S1.2]*.

Generated trace

- S1(entry)::S1.1(entry)::T1.2(effect)::S1.2(entry)::S1(exit)::T3(effect)::S1(entry)::S1.2(entry)::S1(exit)::S2(entry)

Note. The purpose of this test is to demonstrate that, when a deep-history pseudo state is owned by a state machine region, then, when that region is entered via that pseudostate, the last recorded configuration of the region is restored. Consider the situation where the state machine is in configuration *S1[S1.2]*.When the *AnotherSignal* event occurrence is dispatched, *T3* is fired. The traversal of *T3* implies that *S1.2* and *S1* are exited. The entrance of the deep-history pseudostate causes the last recorded configuration for that region to be restored: *S1[S1.2]*. When the *Continue* event occurrence is dispatched, *T1.3* fires and S1 completes. The completion event generated by that state causes the traversal of *T4* and entry into *S2*. The state machine execution completes at the end of the RTC step initiated by the dispatching of the *S2* completion event occurrence.

RTC Steps

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T1.1)]
4	[**CE(S1.1)**]	[S1[S1.1]]	[T1.2]
5	[AnotherSignal, **CE(S1.2)**]	[S1[S1.2]]	[]
6	[**AnotherSignal**]	[S1[S1.2]]	[T3]
7	[Continue, **CE(S1.2)**]	[S1[S1.2]]	[]
8	[**Continue**]	[S1[S1.2]]	[T1.3]
9	[**CE(S1)**]	[S1]	[T4]
10	[**CE(S2)**]	[S2]	[T5]

9.3.15.6 History 002-A

Tested state machine

The state machine that is executed for this test is presented in Figure 9.86.

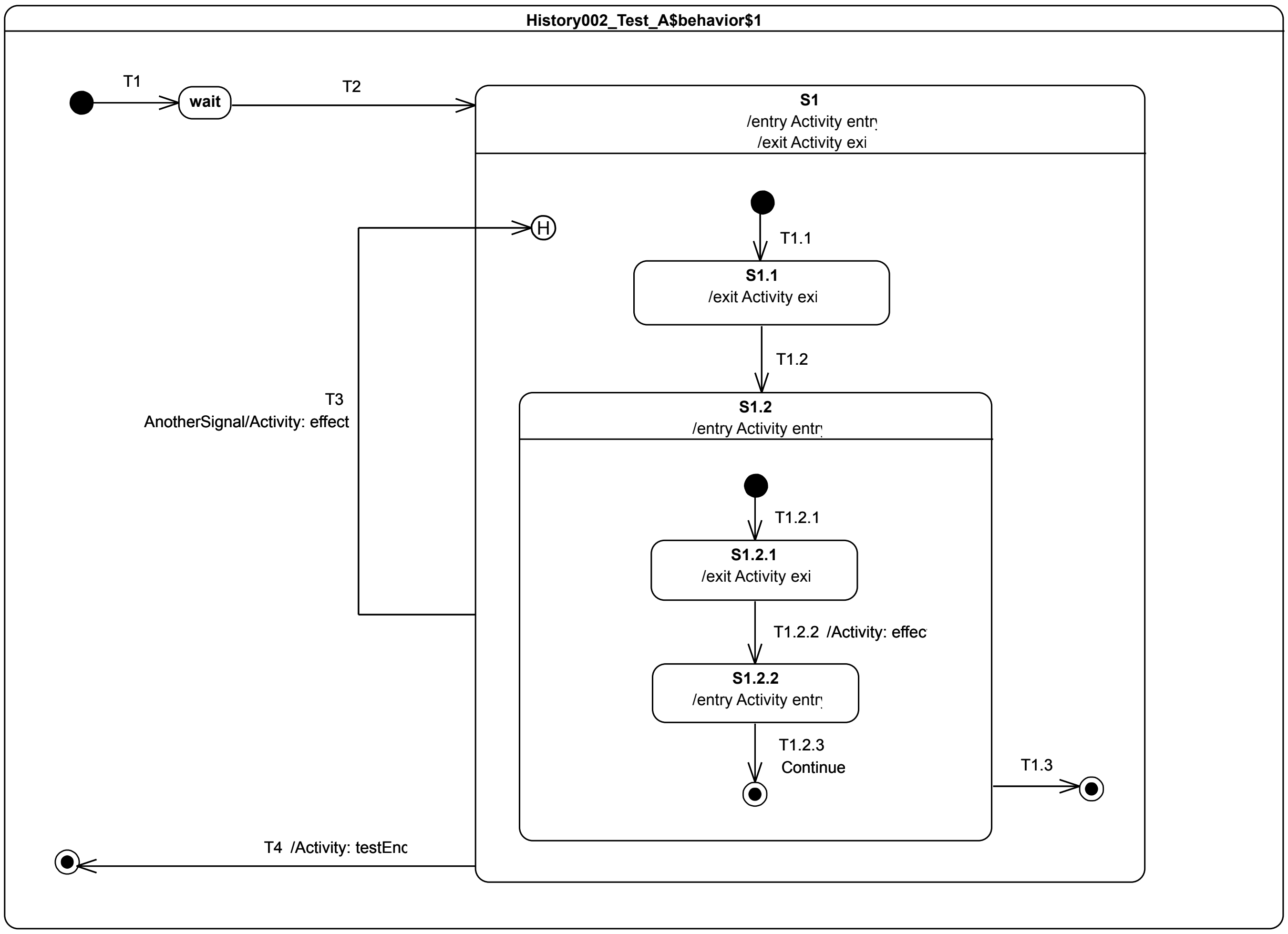


Figure 9.86 - History 002 - A Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- AnotherSignal – received when in configuration *S1[S1.2[S1.2.2]]*.
- Continue – received when in configuration *S1[S1.2[S1.2.2]]*.

Generated trace

- S1(entry)::S1.1(exit)::S1.2(entry)::S1.2.1(exit)::T1.2.2(effect)::S1.2.2(entry)::S1(exit)::T3(effect)::S1(entry)::S1.2(entry)::S1.2.1(exit)::T1.2.2(effect)::S1.2.2(entry)::S1(exit)

Note. The purpose of this test is to demonstrate that, when a shallow-history pseudostate is entered and the region containing that pseudostate has a history, only the top state in the current deep history of that state is restored (and, if that state is a composite, the remainder of its internal deep-history configuration is not restored). Consider the situation where the state machine is in configuration *[S1[S1.2[S1.2.2]]]*. When the *AnotherSignal* event occurrence is dispatched, *T3* is fired. First, *S1.2.2*, *S1.2*, and are *S1* exited. Next, the shallow-history pseudostate is reached. This implies that *S1* is entered and *S1.2* is restored. When *S1.2* is restored, its internal state configuration is not restored. Indeed, its region is entered using the default entry rule. That means the region execution starts from the initial pseudostate. The RTC step initiated by the dispatching of *AnotherSignal* ends after the completion event occurrence for *S1.2.1* is generated. When this completion event occurrence is dispatched, *T1.2.2* is fired and *S1.2.2* is entered. At this point, the state machine is in configuration *[S1[S1.2[S1.2.2]]]* and waits for the dispatching of a *Continue* signal event occurrence. When this event occurrence is dispatched, *T1.2.3* fires and *S1.2* completes. The completion event occurrence generated for that state is used to trigger the transition *T1.3* in the next RTC step. *T4* will be traversed due to the completion event occurrence generated by *S1* and will lead to the completion of the state machine execution.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T1.1)]
4	[**CE(S1.1)**]	[S1[S1.1]]	[T1.2(T1.2.1)]
5	[**CE(S1.2.1)**]	[S1[S1.2[S1.2.1]]]	[T1.2.2]
6	[AnotherSignal, **CE(S1.2.2)**]	[S1[S1.2[S1.2.2]]]	[]
7	[**AnotherSignal**]	[S1[S1.2[S1.2.2]]]	[T3(T1.2.1)]
8	[**CE(S1.2.1)**]	[S1[S1.2[S1.2.1]]]	[T1.2.2]
9	[Continue, **CE(S1.2.2)**]	[S1[S1.2[S1.2.2]]]	[]
10	[**Continue**]	[S1[S1.2[S1.2.2]]]	[T1.2.3]
11	[**CE(S1.2)**]	[S1[S1.2]]	[T1.3]
12	[**CE(S1)**]	[S1]	[T4]

9.3.15.7 History 002-B

Tested state machine

The state machine that is executed for this test is presented in Figure 9.84.

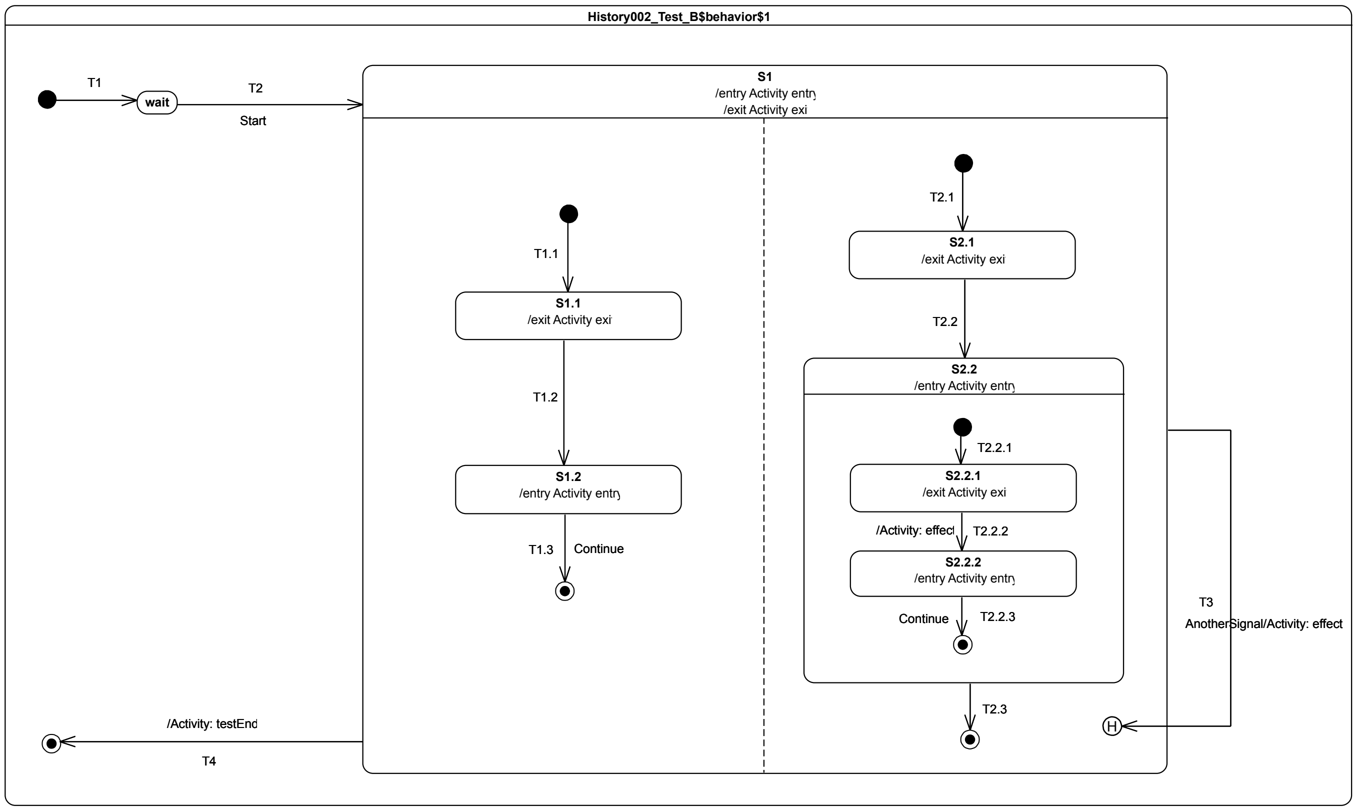


Figure 9.87 - History 002 - B Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- AnotherSignal – received when in configuration *S1[S1.2, S2.2[S2.2.2]]*.
- Continue – received when in configuration *S1[S1.2, S2.2[S2.2.2]]*.

Generated trace

- S1(entry)::S1.1(exit)::S1.2(entry)::S2.1(exit)::S2.2(entry)::S2.2.1(exit)::T2.2.2(effect)::S2.2.2(entry)::S1(exit)::T3(effect)::S1(entry)::S2.2(entry)::S1.1(exit)::S1.2(entry)::S2.2.1(exit)::T2.2.2(effect)::S2.2.2(entry)::S1(exit)

Note. The purpose of this test is to demonstrate that, in a situation where orthogonal regions are involved, the usage of a shallow-history pseudostate only affects the region in which this pseudostate is located. Consider the situation where the state machine is in configuration *S1[S1.2, S2.2[S2.2.2]]*. When the *AnotherSignal* event occurrence is dispatched, the full configuration is exited starting with innermost active states located in both regions (i.e., *S1.2* and *S2.2.2*). Next, *S1* is entered, the left region starts executing from its initial pseudo state, while in the right region *S2.2* is restored. The action of restoring *S2.2* involves the execution of its own region, which starts from the initial pseudostate. At the end of the RTC step initiated by dispatching of the AnotherSignal event occurrence, the state machine is in configuration *S1[S1.1, S2.2[S2.2.1]].* The next step consists in the firing of *T1.2* upon the dispatching of the completion event occurrence generated by *S1.1*. At this point, the only remaining event occurrence in the event pool is the completion event occurrence generated for *S2.2.1*. When dispatched, this will trigger *T2.2.2* and enable *S2.2.2* to be entered. The execution then waits for a *Continue* event occurrence, which, when dispatched, will trigger both *T1.3* and *T2.2.3*. The two remaining execution steps are initiated by the completion event occurrences generated respectively by *S2.2* and *S1*.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[IntialTransition]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T1.1, T2.1)]
4	[CE(S2.1), **CE(S1.1)**]	[S1[S1.1, S2.1]]	[T1.2]
5	[CE(S1.2), **CE(S2.1)**]	[S1[S1.2, S2.1]]	[T2.2(T2.2.1)]
6	[CE(S2.2.1), **CE(1.2)**]	[S1[S1.2, S2.2[S2.2.1]]]	[]
7	[**CE(S2.2.1)**]	[S1[S1.2, S2.2[S2.2.1]]]	[T2.2.2]
8	[AnotherSignal, **CE(S2.2.2)**]	[S1[S1.2, S2.2[S2.2.2]]]	[]
9	[**AnotherSignal**]	[S1[S1.2, S2.2[S2.2.2]]]	[T3(T1.1, T2.2.1)]
10	[CE(S2.2.1), **CE(S1.1)**]	[S1[S1.1, S2.2[S2.2.1]]]	[T1.2]
11	[CE(S1.2), **CE(S2.2.1)**]	[S1[S1.2, S2.2[S2.2.1]]]	[T2.2.2]
12	[CE(S2.2.2), **CE(S1.2)**]	[S1[S1.2, S2.2[S2.2.2]]]	[]
13	[Continue, **CE(S2.2.2)**]	[S1[S1.2, S2.2[S2.2.2]]]	[]
14	[**Continue**]	[S1[S1.2, S2.2[S2.2.2]]]	[T1.3, T2.2.3]
15	[**CE(S2.2)**]	[S1[S2.2]]	[T2.3]
16	[**CE(S1)**]	[S1]	[T4]

Alternative execution traces

The presence of orthogonal regions in S1 implies the possibility to obtain other valid traces. These are listed below.

1. S1(entry)::S1.1(exit)::S1.2(entry)::S2.1(exit)::S2.2(entry)::S2.2.1(exit)::T2.2.2(effect)::S2.2.2(entry)::S1(exit)::T3(effect)::S1(entry)::S2.2(entry)::S2.2.1(exit)::T2.2.2(effect)::S2.2.2(entry)::S1.1(exit)::S1.2(entry)::S1(exit)

2. S1(entry)::S1.1(exit)::S1.2(entry)::S2.1(exit)::S2.2(entry)::S2.2.1(exit)::T2.2.2(effect)::S2.2.2(entry)::S1(exit)::T3(effect)::S1(entry)::S1.1(exit)::S1.2(entry)::S2.2(entry)::S2.2.1(exit)::T2.2.2(effect)::S2.2.2(entry)::S1(exit)

3. S1(entry)::S2.1(exit)::S2.2(entry)::S2.2.1(exit)::T2.2.2(effect)::S2.2.2(entry)::S1.1(exit)::S1.2(entry)::S1(exit)::T3(effect)::S1(entry)::S2.2(entry)::S1.1(exit)::S1.2(entry)::S2.2.1(exit)::T2.2.2(effect)::S2.2.2(entry)::S1(exit)

4. S1(entry)::S2.1(exit)::S2.2(entry)::S2.2.1(exit)::T2.2.2(effect)::S2.2.2(entry)::S1.1(exit)::S1.2(entry)::S1(exit)::T3(effect)::S1(entry)::S2.2(entry)::S2.2.1(exit)::T2.2.2(effect)::S2.2.2(entry)::S1.1(exit)::S1.2(entry)::S1(exit)
5. S1(entry)::S2.1(exit)::S2.2(entry)::S2.2.1(exit)::T2.2.2(effect)::S2.2.2(entry)::S1.1(exit)::S1.2(entry)::S1(exit)::T3(effect)::S1(entry)::S1.1(exit)::S1.2(entry)::S2.2(entry)::S2.2.1(exit)::T2.2.2(effect)::S2.2.2(entry)::S1(exit)

9.3.15.8 History 002-C

Tested state machine

The state machine that is executed for this test is presented in Figure 9.88.

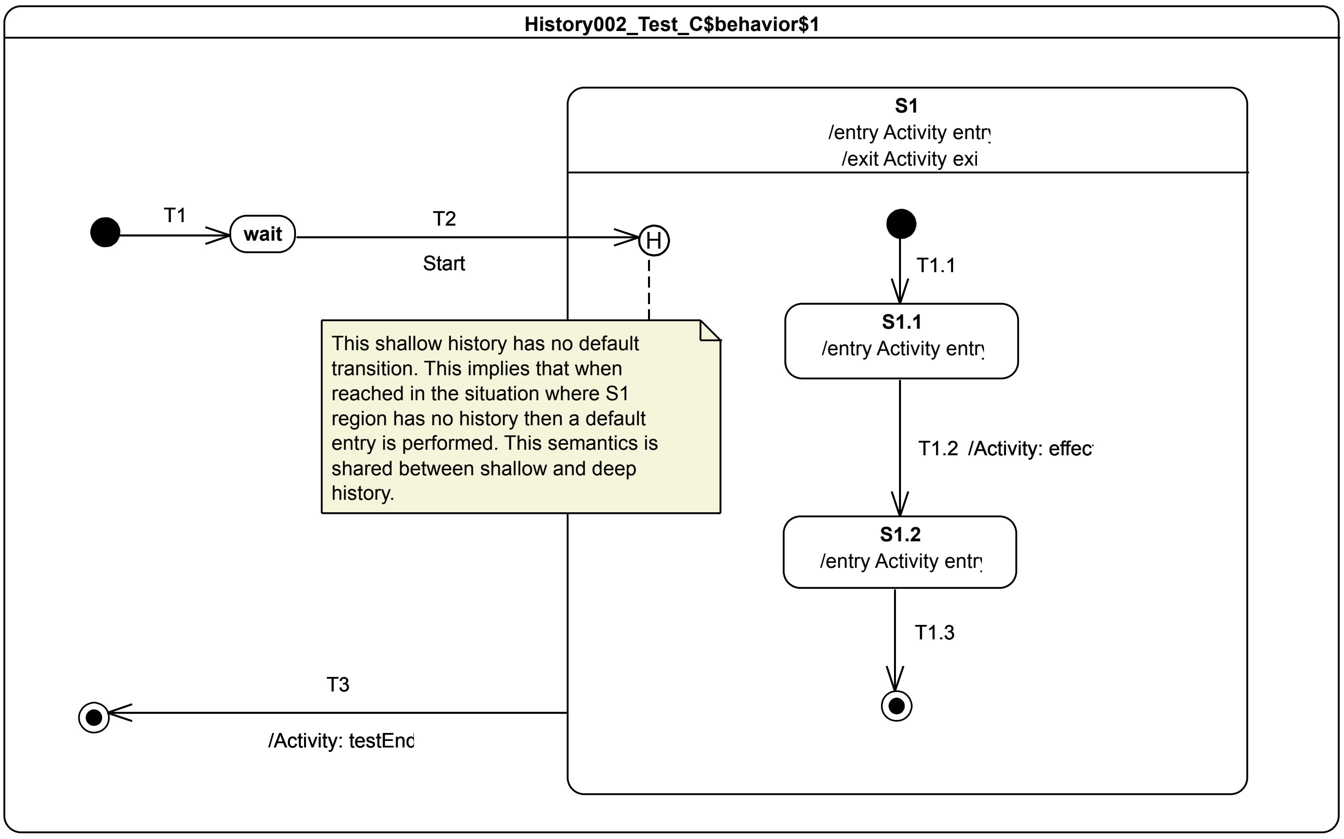

History002_Test_C$behavior$1
S1
/entry Activity entr
/exit Activity exi
T1
wait
T2
Start
H
T1.1
S1.1
/entry Activity entr
T1.2 /Activity: effec
S1.2
/entry Activity entr
T1.3
T3
/Activity: testEnd
This shallow history has no default transition. This implies that when reached in the situation where S1 region has no history then a default entry is performed. This semantics is shared between shallow and deep history.

Figure 9.88 - History 002 - C Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- S1(entry)::S1.1(entry)::T1.2(effect)::S1.2(entry)::S1(exit)

Note. The purpose of this test is to demonstrate that, if a history pseudostate (a shallow history in this test) has no default transition and its containing region has no history, then a default entry is performed for that region. Consider the situation where the state machine is in configuration *wait*. When the *Start* event occurrence is dispatched, transition *T2* is fired. Since the targeted history pseudostate has no default transition and the region of *S1* has not yet been entered, *S1* is entered, and its single region is entered using the default entry semantics. This means that the execution of the region starts from the initial pseudostate. After that RTC step, the state machine is in configuration *S1[S1.1]*. The completion event occurrence generated by *S1.1* is then used in the next step to trigger *T1.2*. The two remaining RTC steps for that state machine execution are initiated by the dispatching of the *S1.2* completion event occurrence followed by the *S1* completion event occurrence.

RTC steps

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

9.3.15.9 History 002-D

Tested state machine

The state machine that is executed for this test is presented in Figure 9.89.

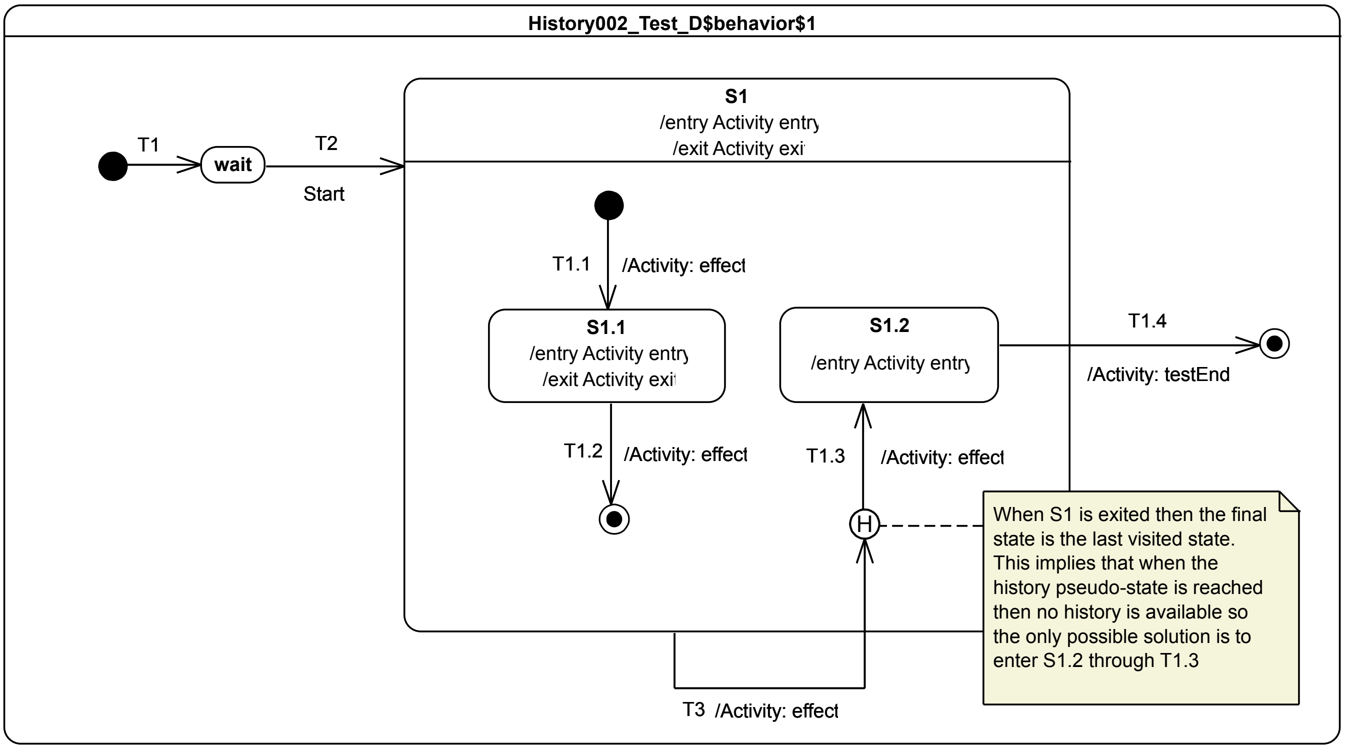

History002_Test_D$behavior$1
T1
wait
T2
Start
S1
/entry Activity entry
/exit Activity exit
T1.1
/Activity: effect
S1.1
/entry Activity entry
/exit Activity exit
S1.2
/entry Activity entry
T1.4
/Activity: testEnd
T1.2
/Activity: effect
T1.3
/Activity: effect
H
When S1 is exited then the final state is the last visited state. This implies that when the history pseudo-state is reached then no history is available so the only possible solution is to enter S1.2 through T1.3
T3 /Activity: effect

Figure 9.89 - History 002 - D Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- S1(entry)::T1.1(effect)::S1.1(entry)::S1(exit)::T1.2(effect)::S1(exit)::T3(effect)::S1(entry)::T1.3(effect)::S1.2(entry)::S1(exit)

Note. The purpose of this test is to demonstrate that when a final state is reached by the execution, then the region containing that final state no longer has any history. Consider the situation where the state machine is in configuration *S1*. The next event to be dispatched is the completion event occurrence for *S1*. When dispatched, this completion event occurrence triggers *T3*. Hence, *S1* is exited, entered again and finally the history pseudostate is reached. As the last state to be executed in the *S1* region was a final state, the region has no history. The only possible execution here is therefore to fire the default transition provided by the history state in order to enter *S1.2*. The completion event occurrence generated by *S1.2* is used to trigger *T1.4* in the next RTC step. When the state machine's final state is reached, its execution completes.

RTC steps

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

9.3.16 Deferred

9.3.16.1 Overview

Tests presented in this subclause assess that deferred event semantics conform to what is specified in UML.

9.3.16.2 Deferred 001

Tested state machine

The state machine that is executed for this test is presented in Figure 9.90.

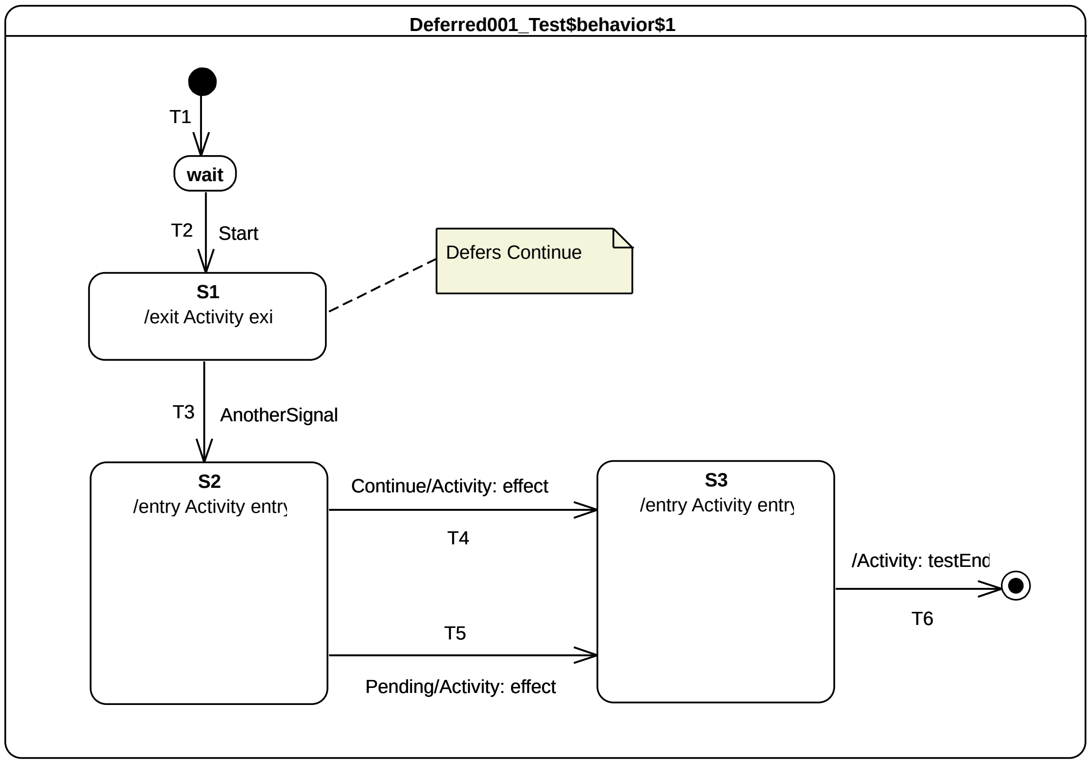


Figure 9.90 - Deferred 001 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1*.
- AnotherSignal – received when in configuration *S1*.
- Pending – received when in configuration *S2*.

Generated trace

- S1(exit)::S2(entry)::T4(effect)::S3(entry)

Note. The purpose of this test is to demonstrate support for event deferral in the context of simple states. It especially shows that, if the state machine is in a configuration enabling the dispatched event occurrence to be deferred, then the event occurrence is deferred and only released when its deferring state has left the state-machine configuration. Consider the situation where the state machine is in configuration *S1*. *S1* has no completion transitions, hence its completion event is lost when it is dispatched. The *Continue* event occurrence is dispatched and can be accepted, since it is indicated as being deferred in configuration *S1*. The acceptance of the event occurrence does not change the state-machine configuration. The next RTC step consists of dispatching and accepting *AnotherSignal*. This leads to the triggering of *T3*. *S1* is exited, which means that the *Continue* event occurrence is no longer deferred, so that *S2* is entered. At this point, three event occurrences are available in the pool. The one at the head of the pool is the completion event for *S2*, the second is the *Continue* event occurrence, and the last is the *Pending* event occurrence. The dispatching of the completion event does not trigger a new RTC step, since *S2* has no completion transition. The next event to be dispatched (i.e., the one that was originally deferred) matches the trigger declared in *T4*. Hence, *T4* is taken, its effect behavior is executed, and *S3* is entered. As the completion event generated by *S3* has priority over the *Pending* event occurrence, it is dispatched first. The *S3* completion transition is traversed, and the state machine execution completes when the final state is reached. Note that, in this execution, the *Pending* event occurrence is never dispatched. However, if the *Continue* event occurrence was actually not deferred, a different execution path would have occurred. That is, *T5* would have been taken instead of *T4*.

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]
4	[AnotherSignal, Continue, **CE(S1)**]	[S1]	[]
5	[AnotherSignal, **Continue**]	[S1]	[]
6	[**AnotherSignal**]	[S1]	[T3]
7	[Pending, **Continue**]	[S2]	[T4]
8	[Pending, **CE(S3)**]	[S3]	[T6]

9.3.16.3 Deferred 002

Tested state machine

The state machine that is executed for this test is presented in Figure 9.91.

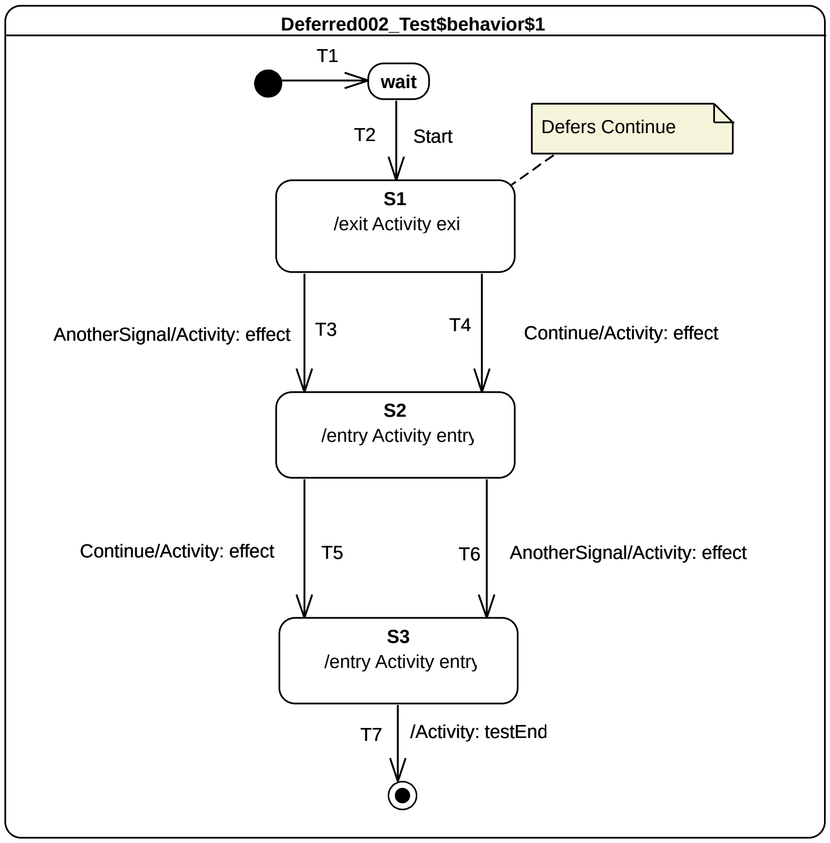

Deferred002_Test$behavior$1
T1
wait
T2
Start
Defers Continue
S1
/exit Activity exi
AnotherSignal/Activity: effect
T3
T4
Continue/Activity: effect
S2
/entry Activity entry
Continue/Activity: effect
T5
T6
AnotherSignal/Activity: effect
S3
/entry Activity entry
T7
/Activity: testEnd

Figure 9.91 - Deferred 002 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1*.
- AnotherSignal – received when in configuration *S1*.
- Continue – received when in configuration *S2*.

Generated trace

- S1(exit)::T4(effect)::S2(entry)::T6(effect)::S3(entry)

Note. The purpose of this test is to demonstrate that, in a specific situation, an event that is declared as being deferred is not actually deferred due to the presence of an *overriding* transition. Consider the situation where the state machine is in configuration *S1* and the completion event of *S1* was dispatched but did not trigger any RTC step. The event pool contains, at this time, two event occurrences. The first is a *Continue* event occurrence and the second is an *AnotherSignal* event occurrence. When the *Continue* event occurrence is accepted, it is not deferred. Indeed a transition originating from the deferring state has a trigger which explicitly refers to the *Continue* event. This overrides the deferring constraint, and transition *T4* is triggered. Next, *S1* is exited, the effect behavior of the transition is executed, and *S2* is entered. The completion event generated by *S2* will not trigger an RTC step, since this state has no completion transition. The next RTC step consists of dispatching and accepting the *AnotherSignal* event occurrence. This means that *T6* is triggered, so that *S2* is exited, the effect behavior is executed, and *S3* is entered. The completion event of *S3* is used to initiate the next RTC step, which leads the state machine to reach the final state and to complete its execution.

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]
4	[AnotherSignal, Continue, **CE(S1)**]	[S1]	[]
5	[AnotherSignal, **Continue**]	[S1]	[T4]
6	[Continue, AnotherSignal**, CE(S2)**]	[S2]	[]
7	[Continue, **AnotherSignal**]	[S2]	[T6]
8	[Continue, **CE(S3)**]	[S3]	[T7]

9.3.16.4 Deferred 003

Tested state machine

The state machine that is executed for this test is presented in Figure 9.92.

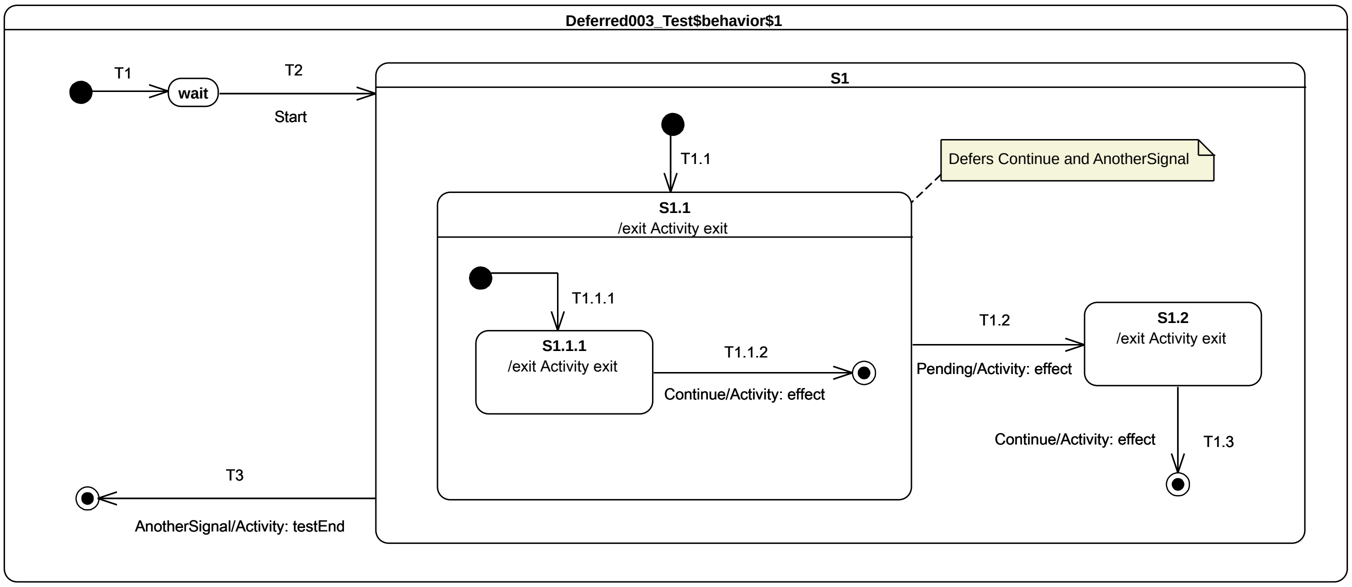

Deferred003_Test$behavior$1
T1
wait
T2
Start
S1
T1.1
Defers Continue and AnotherSignal
S1.1
/exit Activity exit
T1.1.1
S1.1.1
/exit Activity exit
T1.1.2
Continue/Activity: effect
T1.2
Pending/Activity: effect
S1.2
/exit Activity exit
Continue/Activity: effect
T1.3
T3
AnotherSignal/Activity: testEnd

Figure 9.92 - Deferred 003 Test 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]]*.
- Continue – received when in configuration *S1[S1.1]*.
- AnotherSignal – received when in configuration *S1[S1.1]*.
- Pending – received when in configuration *S1[S1.1]*.

Generated trace

- S1.1.1(exit)::T1.1.2(effect)::S1.1(exit)::T1.2(effect)::S1.2(exit)::T1.3(effect)

Note. The purpose of this test is to demonstrate the support for deferred-event semantics when the deferred event is declared by a composite state. Consider the situation where the state machine is in the configuration *S1[S1.1[S1.1.1]]*. There are, at this time, two event occurrences that are ready to be dispatched: the first is a completion event for state *S1.1.1* and the second is a *Continue* event occurrence. The completion event is lost since *S1.1.1* has no completion transition. Next, the *Continue* event occurrence is dispatched and accepted. Transition *T1.1.2* is triggered by this event occurrence. This transition has priority over the deferring constraint added by *S1.1* since it is more deeply nested in the state hierarchy. Therefore, *T1.1.2* is triggered and *S1.1.1* leaves the state machine configuration so that *S1.1* region completes. The completion event generated by *S1.1* cannot be used to trigger any transition, so it is lost. The next RTC step consists of dispatching a new *Continue* event occurrence. This event occurrence is accepted and deferred due to the constraint required by *S1.1*. The next event occurrence to be dispatched and accepted is for *AnotherSignal*. *S1.1* also defers this type of event occurrence. Since the state machine configuration did not change, the event occurrence is deferred. To summarize, at this point of the execution two events occurrences are deferred: one *Continue* event occurrence and one *AnotherSignal* event occurrence. The next RTC step is initiated by the acceptance of the *Pending* event occurrence. As *T1.2* can be triggered using this event occurrence, state *S1.1* is exited (the deferred events are released), the effect behavior of the transition is executed, and *S1.2* is entered. The *S1.2* completion event does not trigger an RTC step, since it has no completion transitions. The *Continue* event occurrence that was originally deferred is used to trigger *T1.3,* which leads the *S1* region to complete. The *S1* completion event is lost, however. The final RTC step is initiated by the dispatching of the *AnotherSignal* event occurrence. It triggers *T3*, which enables the state machine to reach the final state and to complete its execution.

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)]
4	[Continue, **CE(S1.1.1)**]	[S1[S1.1[S1.1.1]]]	[]
5	[**Continue**]	[S1[S1.1]]	[T1.1.2]
6	[Pending, AnotherSignal, Continue, **CE(S1.1)**]	[S1[S1.1]]	[]
7	[Pending, AnotherSignal, **Continue**]	[S1[S1.1]]	[]
8	[Pending, **AnotherSignal**]	[S1[S1.1]]	[]
9	[**Pending**]	[S1[S1.1]]	[T1.2]
10	[AnotherSignal, Continue, **CE(S1.2)**]	[S1.2]	[]
11	[AnotherSignal, **Continue**]	[S1.2]	[T1.3]
12	[AnotherSignal, **CE(S1)**]	[S1]	[]
13	[AnotherSignal]	[S1]	[T3]

9.3.16.5 Deferred 004-A

Tested state machine

The state machine that is executed for this test is presented in Figure 9.93.

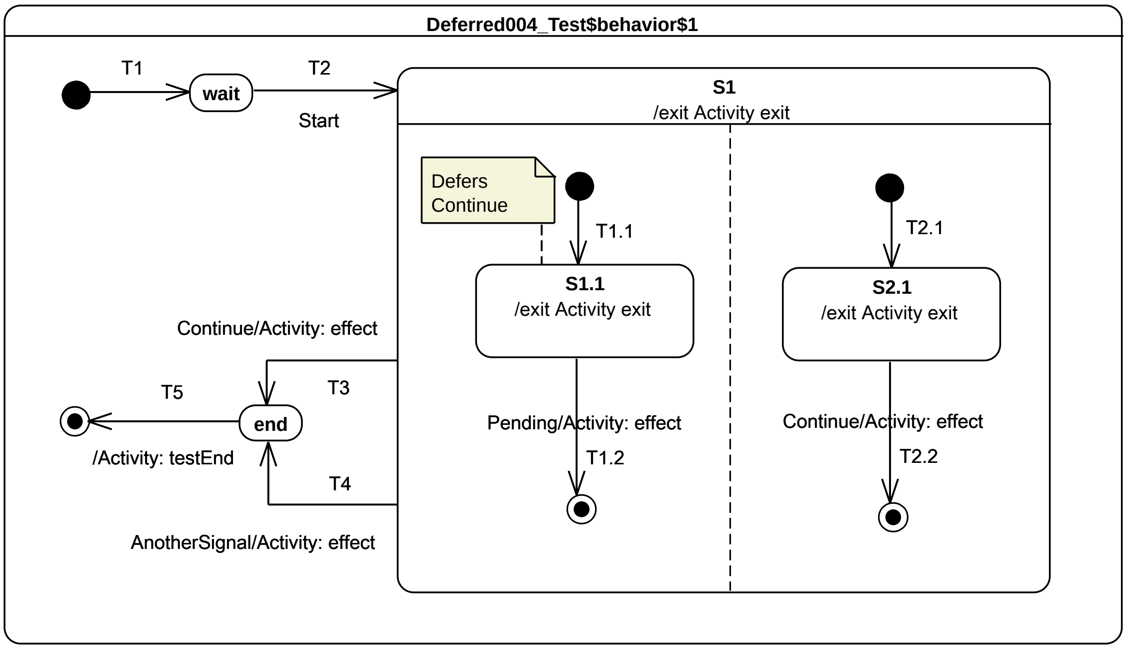


Figure 9.93 - Deferred 004 - A Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1[S1.1, S2.1]*.
- Pending – received when in configuration *S1[S1.1, 2.1]*.
- AnotherSignal – received when in configuration *S1*.

Generated trace

- S1.1(exit)::T1.2(effect)::S2.1(exit)::T2.2(effect)::S1(exit)::T4(effect)

Note. The purpose of this test is to assess deferred-event semantics when used in the context of orthogonal regions. Consider the situation where the state machine is in configuration *S1[S1.1, S2.1]*. After dispatching of completion events for *S1.1* and *S2.1,* there remains one event occurrence in the pool that is ready to be dispatched: a *Continue* event occurrence. When the *Continue* event occurrence is dispatched and accepted by the state machine, it is deferred by *S1.1*. The current state machine configuration remains *S1[S1.1, S2.1]*. The next RTC step is initiated by the acceptance of the *Pending* event occurrence. This starts by triggering *T1.2*. As *S1.1* leaves the state machine configuration, the *Continue* event occurrence is now available in the pool. The left-hand region completes. *T2.2* is triggered by the RTC step initiated by the dispatching of the Continue event occurrence. This leads *S1* to generate a completion event. The completion event is lost, since *S1* has no outgoing completion transition. *S1* is exited when *AnotherSignal* event is dispatched.

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, T2.1)]
4	[Pending, Continue, CE(S2.1), **CE(S1.1)**]	[S1[S1.1, S2.1]]	[]
5	[Pending, Continue, **CE(S2.1)**]	[S1[S1.1, S2.1]]	[]
6	[Pending, **Continue**]	[S1[S1.1, S2.1]]	[]
7	[**Pending**]	[S1[S1.1, S2.1]]	[T1.2]
8	[**Continue**]	[S1[S2.1]]	[T2.2]
9	[AnotherSignal, **CE(S1)**]	[S1]	[]
10	[**AnotherSignal**]	[S1]	[T4]
11	[**CE(end)**]	[end]	[T5]

9.3.16.6 Deferred 004-B

Tested state machine

The state machine that is executed for this test is presented in Figure 9.94.

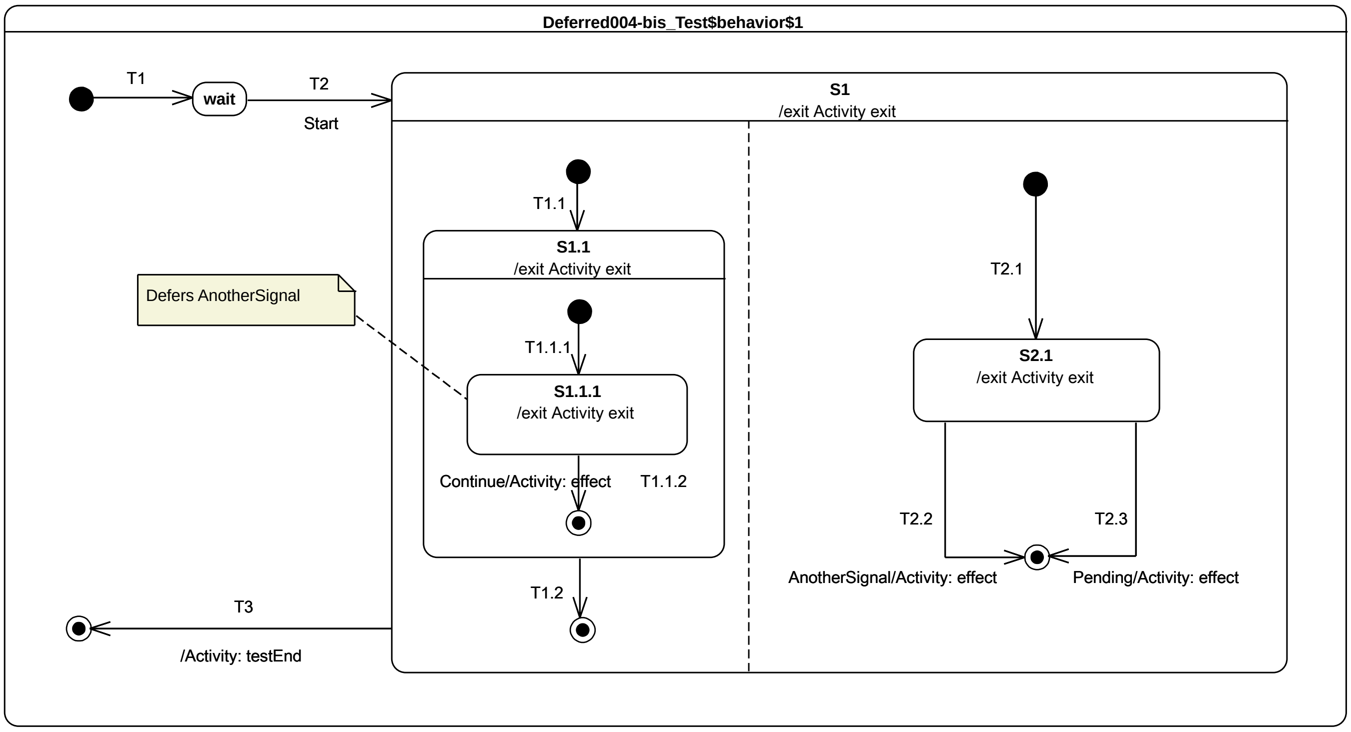


Figure 9.94 - Deferred 004 - B Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- AnotherSignal – received when in configuration *S1[S1.1[S1.1.1], S2.1]*.
- Continue – received when in configuration *S1[S1.1[S1.1.1], S2.1]*.
- Pending – received when in configuration *S1*.

Generated trace

- S1.1.1(exit)::T1.1.2(effect)::S1.1(exit)::S2.1(exit)::T2.2(effect)::S1(exit)

Note. The purpose of this test is is to assess deferred-event semantics when used in the context of orthogonal regions. Consider the situation where the state machine is in configuration *S1[S1.1[S1.1.1], S2.1]*. After dispatching of completion events for *S1.1.1* and *S2.1,* there remain two event occurrences in the pool: an *AnotherSignal* event occurrence and a *Continue* event occurrence. When accepted, the *AnotherSignal* event occurrence is deferred by *S1.1.1*. The next RTC step is initiated by the acceptance of the *Continue* event occurrence. *T1.1.2* is triggered, which means that *S1.1.1* leaves the state machine configuration, so that the event occurrence that was previously deferred becomes available. The unique region of *S1.1* completes, and a completion event occurrence is generated for *S1.1*. In the next RTC step, this completion event is dispatched and accepted. This means that *T1.2* is triggered, which leads to the completion of the left region of *S1*. At this point, one event occurrence (i.e., *AnotherSignal*) remains in the pool. When dispatched, it triggers *T2.2* which forces and exit of *S2.1,* and the final state is reached, leading to the completion of the *S1* region.

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), T2.1)]
4	[Continue, AnotherSignal, **CE(S1.1.1)**]	[S1[S1.1[S1.1.1], S2.1]]	[]
5	[Continue, **AnotherSignal**]	[S1[S1.1[S1.1.1], S2.1]]	[]
6	[**Continue**]	[S1[S1.1[S1.1.1], S2.1]]	[T1.1.2]
7	[AnotherSignal, **CE(S1.1)**]	[S1[S1.1, S2.1]]	[T1.2]
8	[**AnotherSignal**]	[S1[2.1]]	[T2.2]
9	[Pending, **CE(S1)**]	[S1]	[T3]

9.3.16.7 Deferred 005

Tested state machine

The state machine that is executed for this test is presented in Figure 9.95.

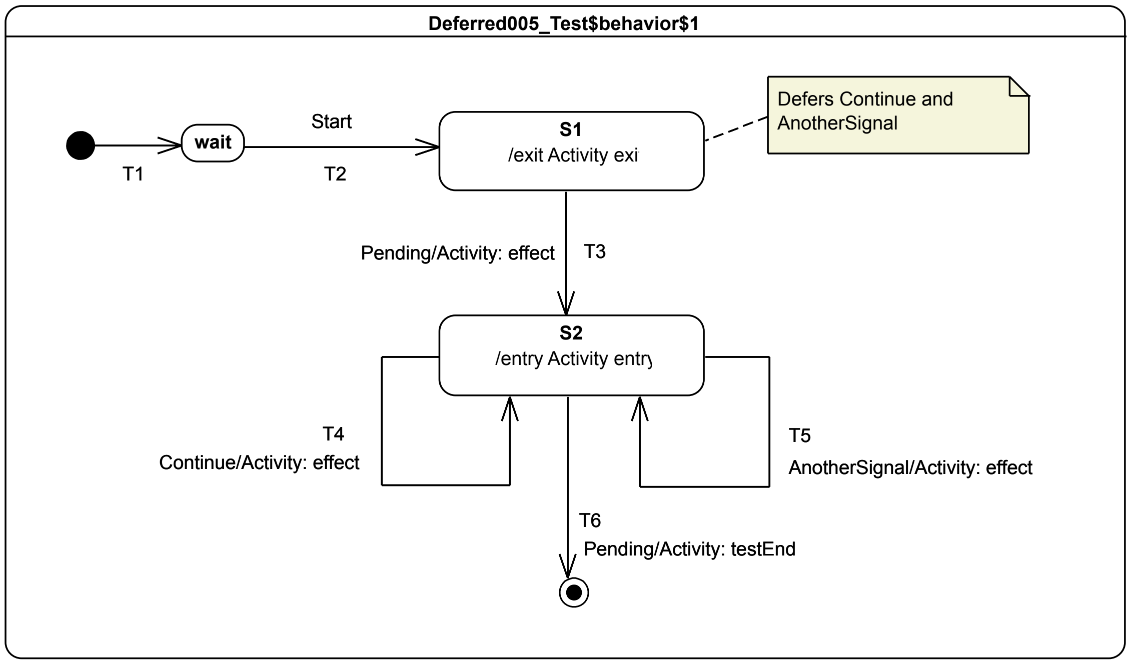


Figure 9.95 - Deferred 005 Test Classifier Behavior

```
activity exit() {
     this.Pending();
}
```

Table 9.4 - S1 exit behavior specification

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1*.
- AnotherSignal – received when in configuration *S1*.
- Pending – received when in configuration *S1*.

Generated trace

- T3(effect)::S2(entry)::T4(effect)::S2(entry)::T5(effect)::S2(entry)

  Note. The purpose of this test is to show that deferred event occurrences return to the event pool in the same order in which they have been deferred and that they will be dispatched before any event occurrence already available in the pool that is not a completion event occurrence. After the first RTC step, the state machine is in configuration *wait*. The completion event generated by *wait* is lost when dispatched, since there is no completion transition outgoing that state. When *Start* is dispatched *T2* is fired and *S1* is entered. During the execution, three event occurrences are received while in configuration *S1*: *Continue, AnotherSignal* and *Pending*. *Continue* and *AnotherSignal* signals are deferred by the state machine, since *S1* declares two deferring triggers for signal events for these signals. This means that, after two steps related to the deferral of *Continue* and *AnotherSignal,* the state machine is still in the same configuration: *S1*. When *Pending* is finally dispatched, *T3* is fired, which implies the execution of the *S1* entry behavior, the release of events deferred by *S1*, the execution of the *T3* effect behavior, and finally, the execution of the *S2* entry behavior. After that step, four event occurrences are available in the pool (see RTC step 8 in the table below): *Pending* (sent by the state machine to itself when the *S1* exit behavior is executed – see Table 9.4), *AnotherSignal*, *Continue*, and the completion event for *S2*. As *S2* does not have a completion transition, the completion event occurrence is lost when dispatched. However, when *Continue* is dispatched, *T4* is fired (see RTC step 9), and, when AnotherSignal is dispatched, *T5* is fired (see RTC step 11). The final step takes place when *Pending* is dispatched. During that step, *T6* is fired, which enables the execution to reach the state machine final state.

RTC steps

Step	**Event pool**	**Deferred events**	**State machine configuration**	**Fired transition(s)**
1	[]	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(wait)**]	[]	[wait]	[]
3	[**Start**]	[]	[wait]	[T2]
4	[AnotherSignal Continue, **CE(S1)**]	[]	[S1]	[]
5	[Pending, AnotherSignal, **Continue**]	[]	[S1]	[]
6	[Pending, A**notherSignal**]	[Continue]	[S1]	[]
7	[**Pending**]	[AnotherSignal, Continue]	[S1]	[T3]
8	[Pending, AnotherSignal Continue, **CE(S2)**]	[]	[S2]	[]
9	[Pending, AnotherSignal, **Continue**]	[]	[S2]	[T4]
10	[Pending, AnotherSignal, **CE(S2)**]	[]	[S2]	[]
11	[Pending, **AnotherSignal**]	[]	[S2]	[T5]
12	[Pending, **CE(S2)**]	[]	[S2]	[]
13	[**Pending**]	[]	[S2]	[T6]

9.3.16.8 Deferred 006-A

Tested state machine

The state machine that is executed for this test is presented in Figure 9.96. The specification of the doActivity associated with the state *S2* is given in Figure 9.97.

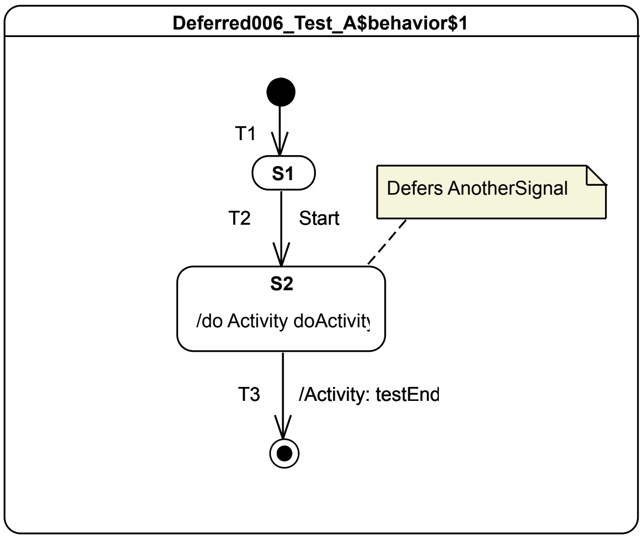


Figure 9.96 - Deferred 006 - A Test Classifier Behavior

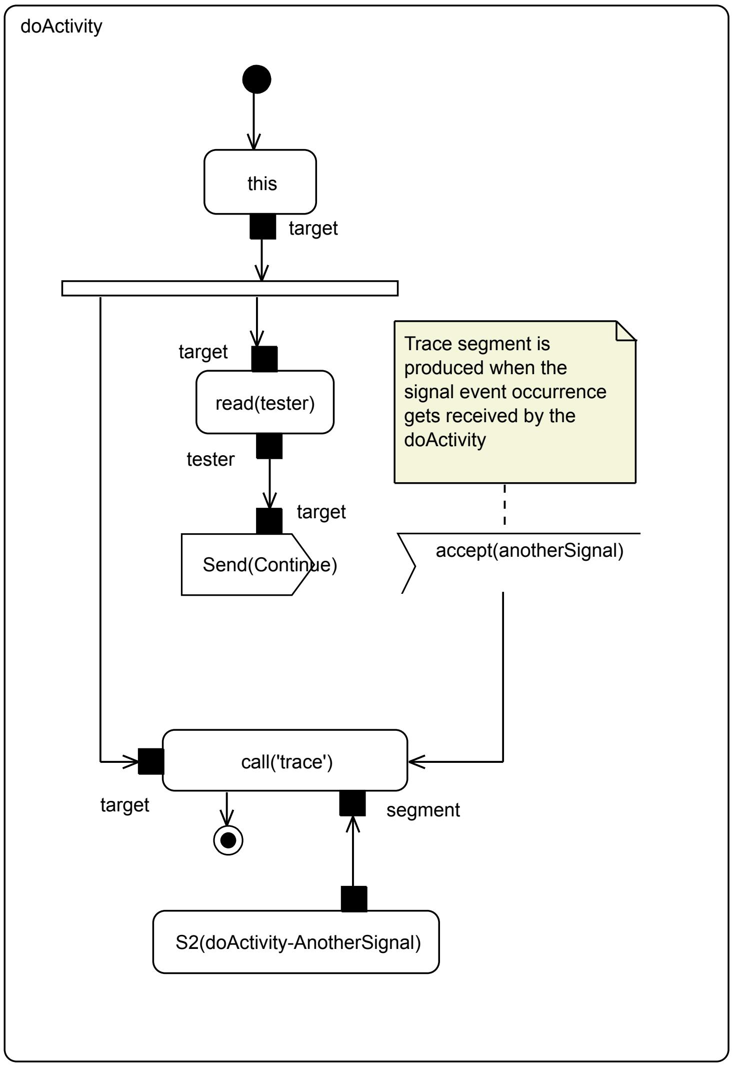

doActivity
this
target
target
read(tester)
tester
target
Send(Continue)
Trace segment is produced when the signal event occurrence gets received by the doActivity
accept(anotherSignal)
call('trace')
target
segment
S2(doActivity-AnotherSignal)

Figure 9.97 - Deferred 006 - A S2 doActivity Behavior

Test execution

Received event occurrences

- Start – received when in configuration *S1*.
- AnotherSignal – received when in configuration *S2*.

Generated trace

- S2(doActivity-AnotherSignal)

  Note. The purpose of this test is to demonstrate that, even if a state has a deferrable trigger that matches a dispatched event occurrence, the event occurrence is not actually deferred if an executing doActivity can accept the event occurrence, so that the doActivity is actually able to accept it (see also 8.5.6). After the first RTC step, the state machine is in the configuration *S1*. The completion event occurrence generated for that state is lost, since no completion transition outgoing *S1* is available. When the *Start* event occurrence is dispatched, *T2* is traversed and *S2* is entered. Upon the entrance of *S2*, its doActivity behavior is invoked. This doActivity behavior registers an accepter for *AnotherSignal* and sends a *Continue* signal to the tester. At this point of the execution, both the state machine and the doActivity are idle (i.e., they wait for the next event to be added in the pool). The next event occurrence coming into the pool is emitted by the tester. This is an *AnotherSignal* event occurrence. At this point, either the state machine can defer the event occurrence or the doActivity can accept this event occurrence to move forward in its execution. The priority is given to the doActivity. The event occurrence is accepted and leads the doActivity to complete. The completion of the doActivity implies the generation of a completion event for *S2*. This completion event is used to fire *T3*. When the final state is reached the state machine execution completes.

RTC steps

Step	**Event pool**	**Deferred events**	**State machine configuration**	**Fired transition(s)**
1	[]	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(S1)**]	[]	[S1]	[]
3	[**Start**]	[]	[S1]	[T2]
4	[**CE(S2)**]	[]	[S2]	[]

Step	**Event pool**	**doActivity configuration**	**Executed Node(s)**
1	[]	[] - Initial RTC step	[InitialNode, this, fork, read(tester), send(Continue), accept(AnotherSignal)]
2	[**AnotherSignal**]	[accept(AnotherSignal)]	[S2(doActivity-AnotherSignal), call(trace), Activity Final Node]

9.3.16.9 Deferred 006-B

Tested state machine

The state machine that is executed for this test is presented in Figure 9.98. The doActivity associated with state S2 is given in Figure 9.99.

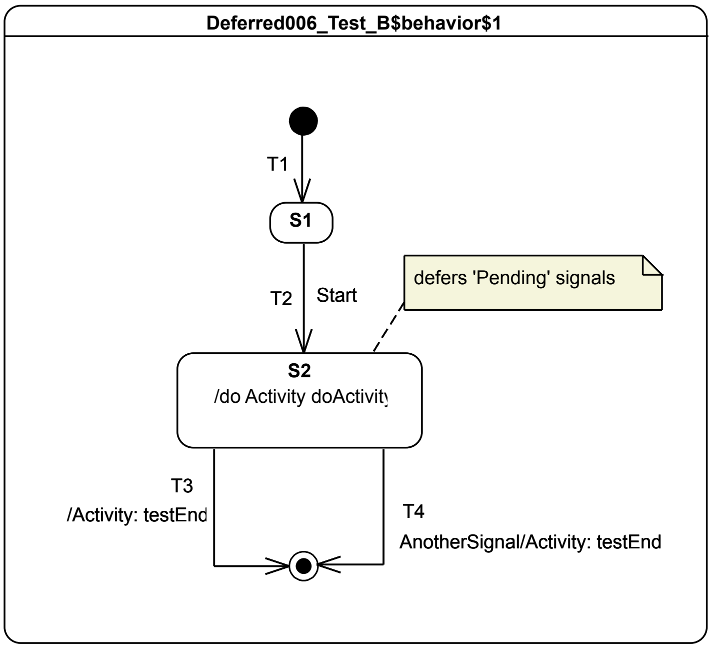


Figure 9.98 - Deferred 006 - B Test Classifier Behavior

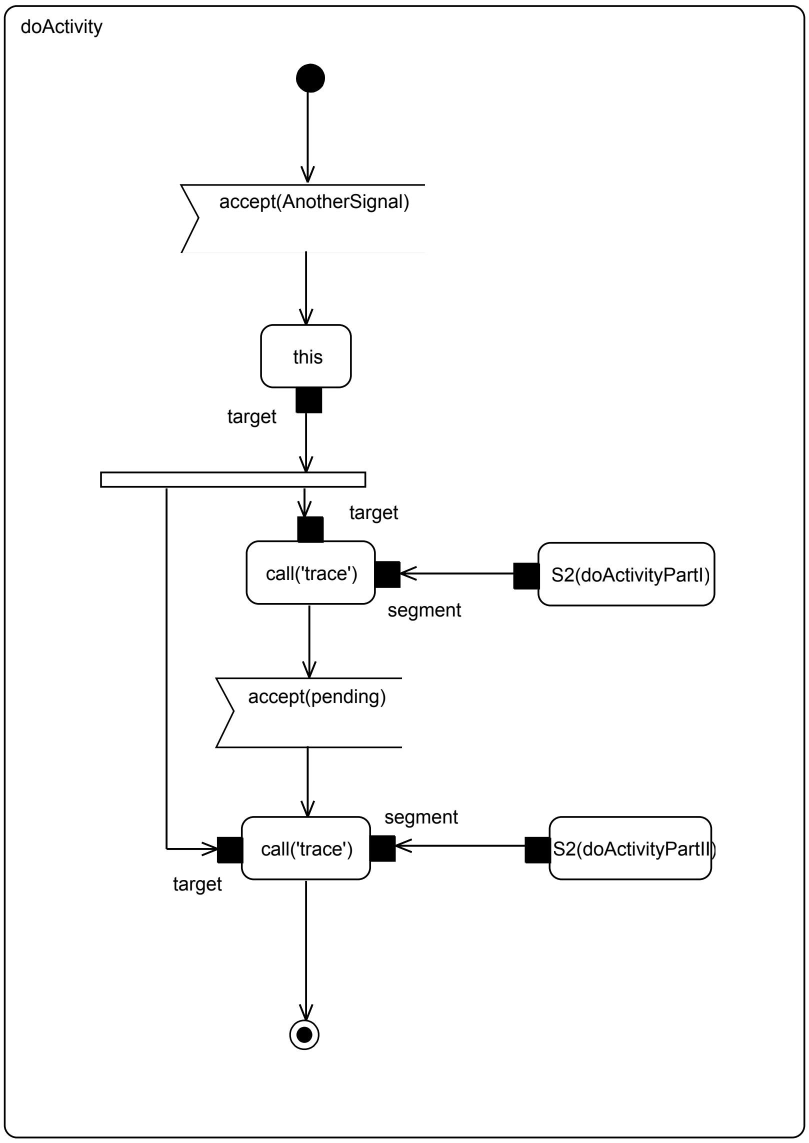


Figure 9.99 - Deferred 006 - B S2 doActivity Behavior Specification

Test execution

Received event occurrences

- Start – received when in configuration *S1*.
- Pending – received when in configuration *S2*.
- AnotherSignal – received when in configuration *S2*.

Generated trace

- S2(doActivityPartI)::S2(doActivityPartII)

Note. The purpose of this test is to demonstrate that, if a state has deferred an event occurrence, then, if the doActivity invoked from that state registers an accepter for the deferred event occurrence, that even occurrence is accepted directly from the deferred event pool. After the first step, the state machine is in the configuration *S1*. The completion event occurrence generated for that state is lost, since no completion transition is available from *S1*. When *Start* is dispatched, *T2* is traversed and *S2* is entered. Upon the entry to *S2,* the doActivity behavior is invoked and the step ends. Concurrently from the state machine execution, the doActivity executes its initial RTC step and registers an accepter for *AnotherSignal*. The state machine receives a *Pending* event occurrence from the tester. This event occurrence is deferred by *S2*. Next, the tester sends an *AnotherSignal* event occurrence. Only the doActivity has an event accepter for such an event occurrence. Hence, the doActivity accepts this event occurrence and moves forward in its execution until it registers an event accepter for a *Pending* event occurrence. When the accepter is registered, a new RTC steps is initiated in the doActivity, because state *S1*, which invoked the doActivity behavior, had previously deferred a *Pending* event occurrence. The new RTC step of the doActivity is initiated by the acceptance of this event occurrence from the deferred event pool. This RTC step implies the completion of the doActivity and the generation of a completion event occurrence for *S2*. When this completion event occurrence is dispatched, *T3* is fired, and the state machine execution completes.

RTC steps

Step	**Event pool**	**Deferred events**	**State machine configuration**	**Fired transition(s)**
1	[]	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(S1)**]	[]	[S1]	[]
3	[**Start**]	[]	[S1]	[T2]
4	[**Pending**]	[]	[S2]	[]
5	[**CE(S2)**]	[]	[S2]	[T3]

Step	**Event pool**	**doActivity configuration**	**Executed Node(s)**
1	[]	[] - Initial RTC step	[InitialNode, accept(AnotherSignal)]
2	[**AnotherSignal**]	[accept(AnotherSignal)]	[this, fork, S2(doActivityPartI), call(trace), accept(Pending)]
3	[**Pending**]	[accept(Pending)]	[S2(doActivityPartII), call(trace), ActivityFinalNode]

9.3.16.10 Deferred 006-C

Tested state machine

The state machine that is executed for this test is presented in Figure 9.100.

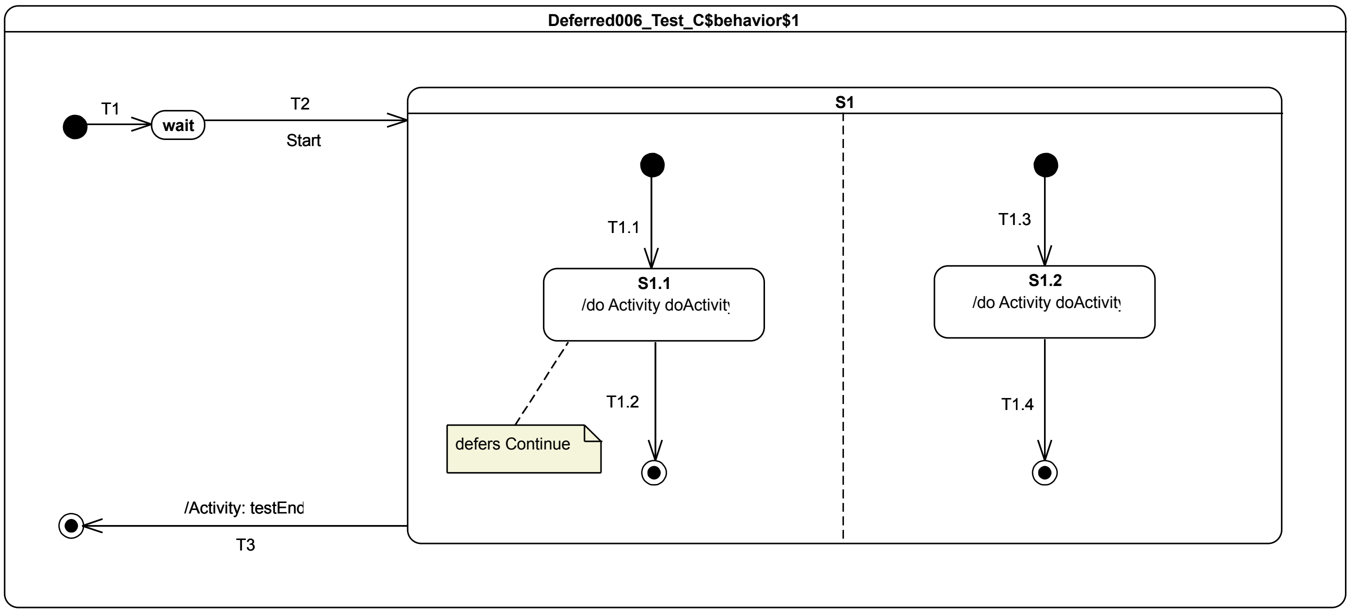


Figure 9.100 - Deferred 006 - C Test Classifier Behavior

Note that, in this state machine:

- The doActivity for *S1.1* registers an accepter for a *Continue* signal and contributes to the trace by adding the trace fragment *S1.1(doActivity)*.
- The doActivity for *S1.2* registers an accepter for a *Continue* signal and contributes to the trace by adding the trace fragment *S1.2(doActivity)*.

Test execution

Received event occurrences

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1[S1.1, S1.2]*
- Continue – received when in configuration *S1[S1.1, S1.2]*

Generated trace

- S1.1(doActivity)::S1.2(doActivity)

  Note. The purpose of this test is to demonstrate the application of deferral semantics in a concurrent context with multiple doActivities competing with the state machine to consume occurrences of the same event. After the initial RTC step, the state machine is in configuration *wait*. When the *Start* event occurrence is received, the compound transition

T2(T1.1, T1.3) is fired. This means that, at the end of the step, the state machine is in configuration *S1[S1.1, S1.2],* with the two doActivity behaviors invoked and evolving on their own threads of execution. Consider that, before the first *Continue* event is dispatched, both doActivities have already registered an accepter for a *Continue*. When a *Continue* event occurrence is dispatched, the state machine is not allowed to defer it. Instead, the priority is given to the doActivities that already have a registered accepter for this event. In such a situation, one of the doActivities will be selected nondeterministically to consume the dispatched event occurrence. Assume that the *S1.1* doActivity behavior is selected to consume the event occurrence. In this situation, *S1.1* will complete upon the completion of the doActivity behavior execution. When the second *Continue* event occurrence is dispatched, only the doActivity behavior invoked by *S1.2* will be able to accept it. A completion event occurrence will be generated for *S1.2* at the end of the RTC step of the doActivity. This will be used to fire *T1.4*. When the final state targeted by *T1.4* is reached, a completion event occurrence is generated for *S1*. That completion event will be used to trigger *T3*.

RTC steps

Step	**Event pool**	**Deferred events**	**State machine configuration**	**Fired transition(s)**
1	[]	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(wait)**]	[]	[wait]	[]
3	[**Start**]	[]	[S1[S1.1, S1.2]]	[T2(T1.1, T1.3)]
6	[Continue, **Continue**]	[]	[S1[S1.1, S1.2]]	[]
7	[Continue, **CE(S1.1)**]	[]	[S1[S1.1, S1.2]]	[T1.2]
8	[**Continue**]	[]	[S1[S1.2]]	[]
9	[**CE(S1.2)**]	[]	[S1[S1.2]]	[T1.4]
10	[**CE(S1)**]	[]	[S1]	[T3]

Step	**Event pool**	**S1.1 doActivity configuration**	**Executed Node(s)**
1	[]	[] - Initial RTC step	[InitialNode, accept(Continue), S1.1(doActivity)]
2	[**Continue**]	[accept(Continue)]	[this, call(trace)]

Step	**Event pool**	**S1.2 doActivity configuration**	**Executed Node(s)**
1	[]	[] - Initial RTC step	[InitialNode, accept(Continue), S1.2(doActivity)]
2	[**Continue**]	[accept(Continue)]	[this, call(trace)]

Alternative execution traces

There exists one alternative execution trace for this test. This trace is described below. It shows the situation where the *S1.2* doActivity behavior is selected to consume the first *Continue* event occurrence, rather than the *S1.1* doActivity behavior.

- S1.2(doActivity)::S1.1(doActivity)

9.3.16.11 Deferred 007

Tested state machine

The state machine that is executed for this test is presented in Figure 9.101.

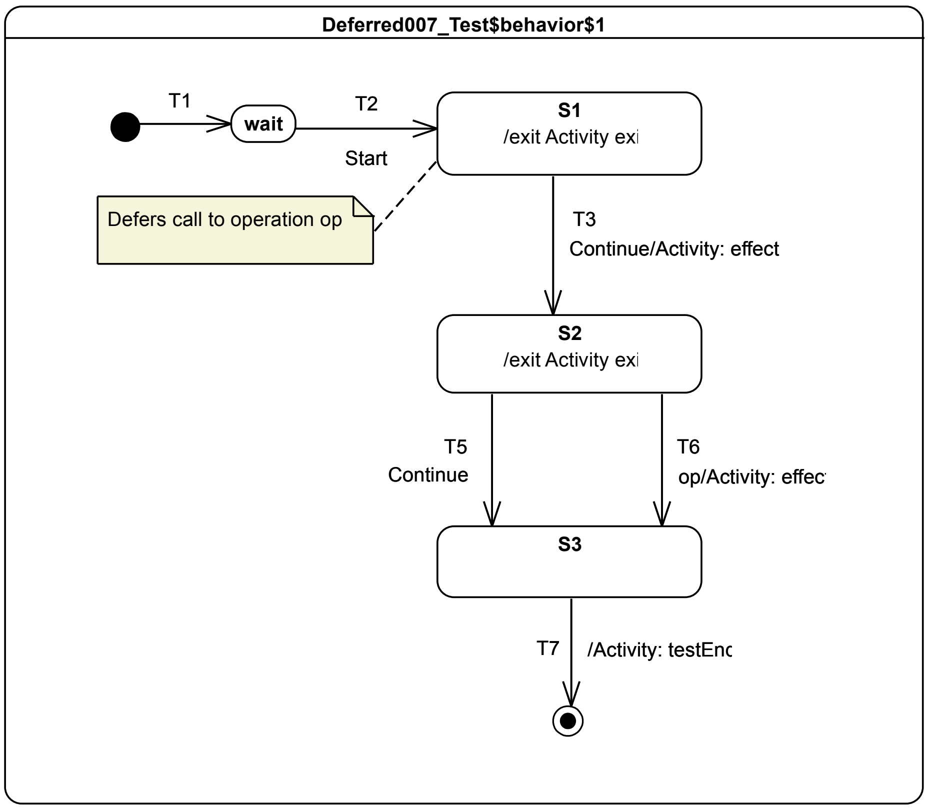


Figure 9.101 - Deferred 007 Test Classifier Behavior

Test execution

Received event occurrences

- Start – received when in configuration *wait*.
- op – received when in configuration *S1*.
- Continue – received when in configuration *S1*.
- Continue – received when in configuration *S2*.

Generated trace

- S1(exit)::T3(effect)::S2(exit)[in=true]::T6(effect)[in=true]

  Note. The purpose of this test is to demonstrate that semantics of deferral also applies to call events. Consider the situation where the state machine is in configuration *S1*. The completion event occurrence generated for *S1* is lost, since that state has no completion transition. The next event to be dispatched is the call event occurrence for the operation *op(in p1: Boolean).* When dispatched, the call event occurrence is deferred by the state machine. This is due to the fact that *S1* declares a deferrable trigger for that call event. Hence, the call event occurrence is placed in the deferred event pool of the state machine and will be released only when *S2* leaves the state-machine configuration. In this test, the call event is released when the first *Continue* event occurrence is dispatched. The dispatching of this event implies the firing of *T3* and the entrance of *S2*. The call event occurrence is placed in the event pool after the completion event occurrence generated for *S2* but before the *Continue* event occurrence already already in the event pool. Hence, *T6* always fires when the call event occurrence is dispatched and *S3* is entered. It is never possible to fire *T5*. The state machine execution completes during the step initiated by the dispatching of the *S3* completion event occurrence.

RTC steps

Step	**Event pool**	**Deferred events**	**State machine configuration**	**Fired transition(s)**
1	[]	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(wait)**]	[]	[wait]	[]
3	[**Start**]	[]	[wait]	[T2]
4	[Call(op(true)), **CE(S1)**]	[]	[S1]	[]
5	[Continue, **Call(op(true))**]	[]	[S1]	[]
6	[**Continue**]	[Call(op(true))]	[S1]	[T3]
7	[Continue, Call(op(true)), **CE(S2)**]	[]	[S2]	[]
8	[Continue, **Call(op(true))**]	[]	[S2]	[T6]
9	[Continue, **CE(S3)**]	[]	[S3]	[T7]

9.3.17 Redefinition

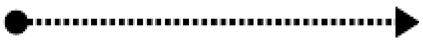
9.3.17.1 Overview

Test cases presented in this subclause assess whether state machine redefinition semantics conform to what is specified in UML.

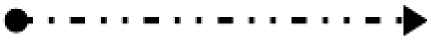
Test architecture

Each test (i.e., extension of Target) contains at least two state machines. This enables each test to specify at minimum one level of extension. The state machine for each presented in such a way that the reader is able to understand:

- Extension relationships existing between state machines as well as between regions. These relationships are shown by the following type of arrow inthe diagrams.



- Redefinition relationships existing between vertices as well as between transitions. These relationships are shown by the following type of arrow in the diagrams.



In addition, the description of each test includes the presentation of the corresponding *runtime* state machine. This state machine is the one effectively executed by the semantic model for PSSM (see the discussion under RegionActivation in 8.5.3). The runtime state machine is the result of merging all the extension and redefinition relationships that were specified for the test into a single state machine. Note that this runtime state machine does not actually exist in the user model.

Test naming conventions

A test involves at least two state machines, which are given different prefix names. The prefixed names are based on the following convention:

- *RedefinitionXXX_Test* – Represents the root level state machine (i.e., it does not extend any other state machine but is, instead, extended by others).
- *RedefinitionXXX_Test_Redefinition_Prime* – A state machine that extends the root state machine.
- *RedefinitionXXX_Test_Redefinition_Second* – A state machine that extends the state machines that extend the root state machine.

This naming convention still applies if additional levels of extensions are added.

Trace construction

The trace produced by a test is the result of the execution of the different behaviors (entry, doActivity, exit and effect) taking place during the test execution. These behaviors may be attached to states and transitions located in different state machines. To capture this, the trace fragment that is produced by a behavior must conform to a specific pattern.

- S<ID>(<BEHAVIOR_NAME>)[-redefined][-<LEVEL>] - denotes the pattern for identifying trace fragments produced by a behavior placed on a state.
    - Example 1: *S1(entry)* denotes a trace fragment produced by an entry behavior placed on state S1, where this state does *not* redefine another state.
    - Example 2: *S1(entry)-redefined* denotes a trace fragment produced by an entry behavior placed on state S1 which redefine a state in a redefined state machine.
    - Example 3: *S1(entry)-redefined-second* denotes a trace fragment produced by an entry behavior placed on state S1, where the state is defined in a state machine that extends another state machine that is itself an extension of another state machine.
- T<ID>(effect)[-redefined][-<LEVEL>] - denotes the pattern for identifying trace fragments produced by the effect behavior of a transition.
    - Example 1: *T1(entry)* denotes a trace fragment produced by an effect behavior placed on a transition T1, where this transition does not redefine another transition.

- Example 2: *T1(entry)-redefined-second* denotes a trace fragment produced by an effect behavior placed on a transition T1, which is defined in a state machine that itself extends another state machine.

9.3.17.2 Redefinition 001

Tested state machine

The test target for Redefinition 001 owns two state machines: *Redefinition001_Test* and *Redefinition001_Test_Redefinition*. The former is extended by the latter. The extension and redefinition relationships existing between the two state machines are shown in Figure 9.102.

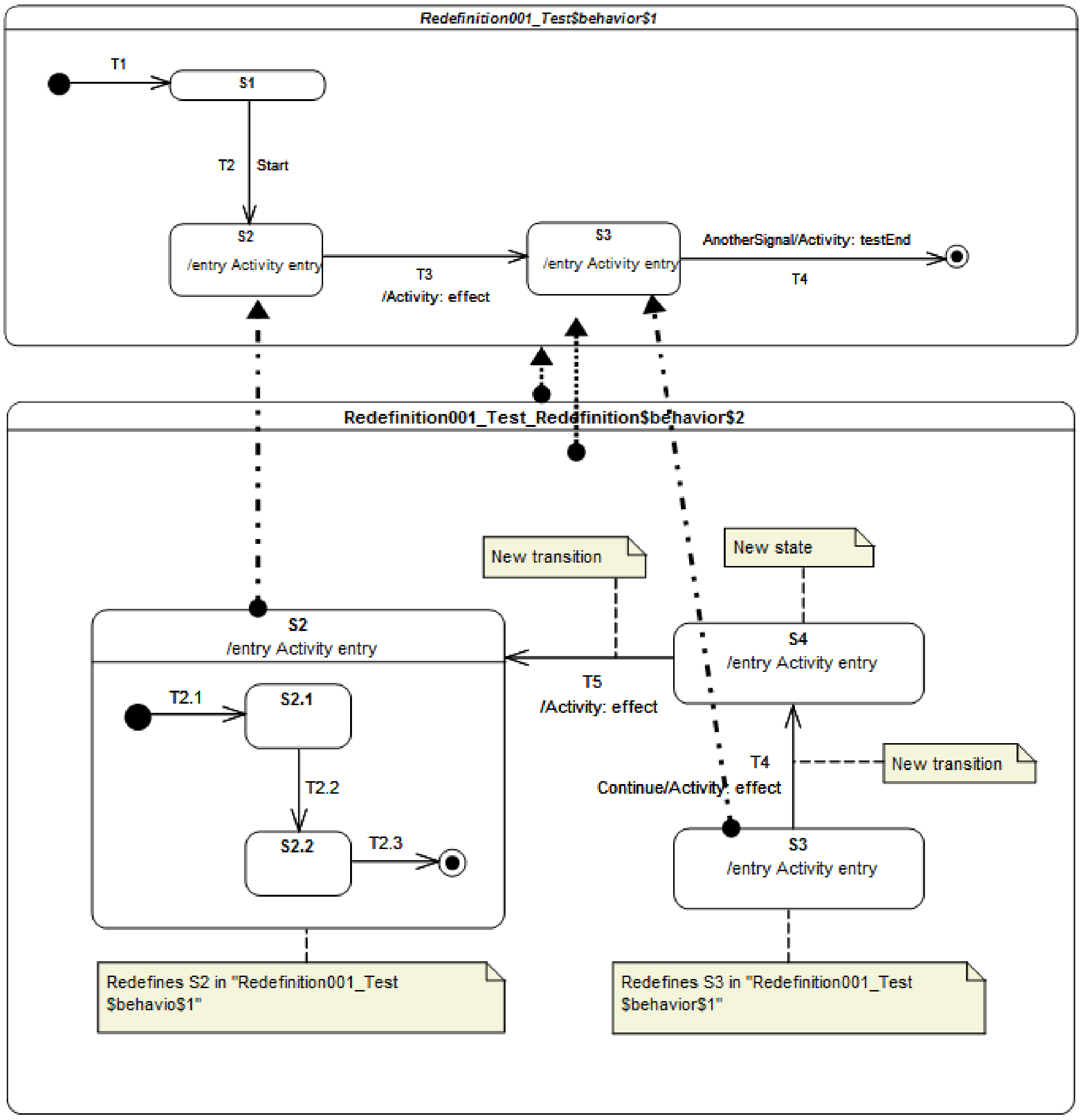

Redefinition001_Test$behavior$1
T1
S1
T2
Start
S2
/entry Activity entry
T3
/Activity: effect
S3
/entry Activity entry
AnotherSignal/Activity: testEnd
T4
Redefinition001_Test_Redefinition$behavior$2
New transition
New state
S2
/entry Activity entry
T2.1
S2.1
T2.2
S2.2
T2.3
T5
/Activity: effect
S4
/entry Activity entry
T4
Continue/Activity: effect
New transition
S3
/entry Activity entry
Redefines S2 in "Redefinition001_Test $behavio$1"
Redefines S3 in "Redefinition001_Test $behavior$1"

Figure 9.102 - Extension and Redefinition Relationships for Redefinition 001

The corresponding runtime state machine is presented in Figure 9.103.

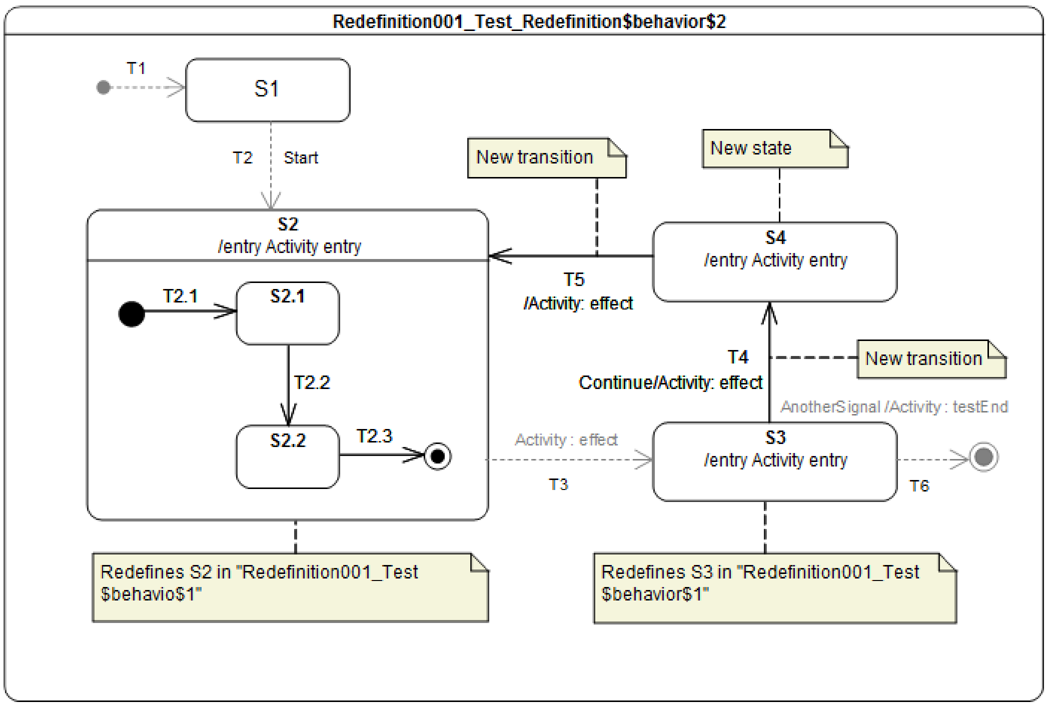

Redefinition001_Test_Redefinition$behavior$2
T1
S1
T2
Start
New transition
New state
S2
/entry Activity entry
S4
/entry Activity entry
T5
/Activity: effect
T2.1
S2.1
T2.2
T4
Continue/Activity: effect
New transition
AnotherSignal /Activity : testEnd
S2.2
T2.3
Activity : effect
S3
/entry Activity entry
T3
T6
Redefines S2 in "Redefinition001_Test $behavio$1"
Redefines S3 in "Redefinition001_Test $behavior$1"

Figure 9.103 - Redefinition 001 Test Classifier Behavior – Runtime State Machine

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S3*.
- AnotherSignal – received when in configuration *S3*.

Generated trace

- S2(entry)-redefined::T3(effect)::S3(entry)-redefined::T4(effect)::S4(entry)::T5(effect)::S2(entry)-redefined::T3(effect)::S3(entry)-redefined

  Note. The purpose of this test is to demonstrate the capability of computing a runtime state machine that is the result of the application of extension and redefinition relationships defined between state machines and internal elements of these state machines. The basic rules are: everything that is not redefined in the extending state machines will be present in the runtime state machine. In addition, all elements that are redefined will be replaced by their latest definition in the runtime state machine. In this test, *S1*, *T1*, *T2*, *T3* and *T6* are part of the extended state machine, and they are not redefined. Hence, they are present in the runtime state machine. *S2* and *S3* are redefined states, so their definitions in the runtime state machine will be those provided by the extending state machine. In addition, all transitions and states added by the extending state machine will be part of the runtime state machine. Following its derivation, the runtime state machine is executed according to the regular state machine semantics (see the RTC steps in the table below).

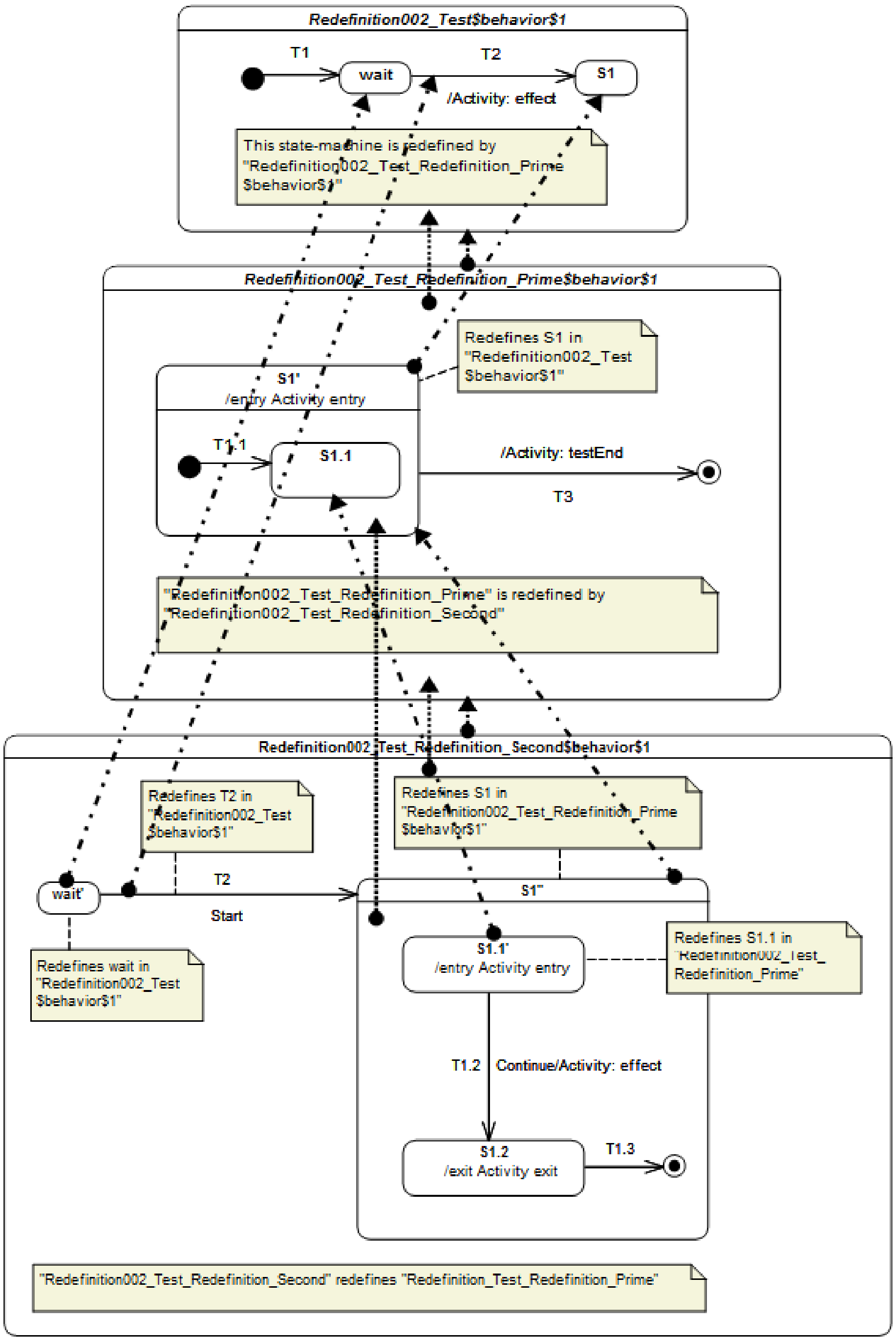
RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Start, **CE(S1)**]	[S1]	[T1]
3	[**Start**]	[S1]	[T2]
4	[**CE(S2.1)**]	[S2[S2.1]]	[T2.2]
5	[**CE(S2.2)**]	[S2[S2.2]]	[T2.3]
6	[**CE(S2)**]	[S2]	[T3]
7	[Continue, **CE(S3)**]	[S3]	[]
8	[**Continue**]	[S3]	[T4]
9	[**CE(S4)**]	[S4]	[T5]
10	[**CE(S2.1)**]	[S2[S2.1]]	[T2.2]
11	[**CE(S2.2)**]	[S2[S2.2]]	[T2.3]
12	[**CE(S2)**]	[S2]	[T3]
13	[AnotherSignal, **CE(S3)**]	[S3]	[]
14	[**AnotherSignal**]	[S3]	[T6]

9.3.17.3 Redefinition 002

Tested state machine

The test target for Redefinition 002 owns three state machines: *Redefinition002_Test, Redefinition002_Test_Redefinition_Prime* and *Redefinition002_Test_Redefinition_Second*. The first state machine is extended by the second which is itself extended by the third. The extension and redefinition relationships between the three state machines are shown in Figure 9.106.



Redefinition002_Test$behavior$1
T1
wait
T2
S1
/Activity: effect
This state-machine is redefined by "Redefinition002_Test_Redefinition_Prime $behavior$1"
Redefinition002_Test_Redefinition_Prime$behavior$1
Redefines S1 in "Redefinition002_Test $behavior$1"
S1'
/entry Activity entry
T1.1
S1.1
/Activity: testEnd
T3
"Redefinition002_Test_Redefinition_Prime" is redefined by "Redefinition002_Test_Redefinition_Second"
Redefinition002_Test_Redefinition_Second$behavior$1
Redefines T2 in "Redefinition002_Test $behavior$1"
Redefines S1 in "Redefinition002_Test_Redefinition_Prime $behavior$1"
wait'
T2
Start
S1"
S1.1'
/entry Activity entry
Redefines S1.1 in "Redefinition002_Test_ Redefinition_Prime"
Redefines wait in "Redefinition002_Test $behavior$1"
T1.2
Continue/Activity: effect
S1.2
/exit Activity exit
T1.3
"Redefinition002_Test_Redefinition_Second" redefines "Redefinition_Test_Redefinition_Prime"

Figure 9.104 - Extension and Redefinition Relationships for Redefinition 002

The corresponding runtime state machine is presented in Figure 9.105.

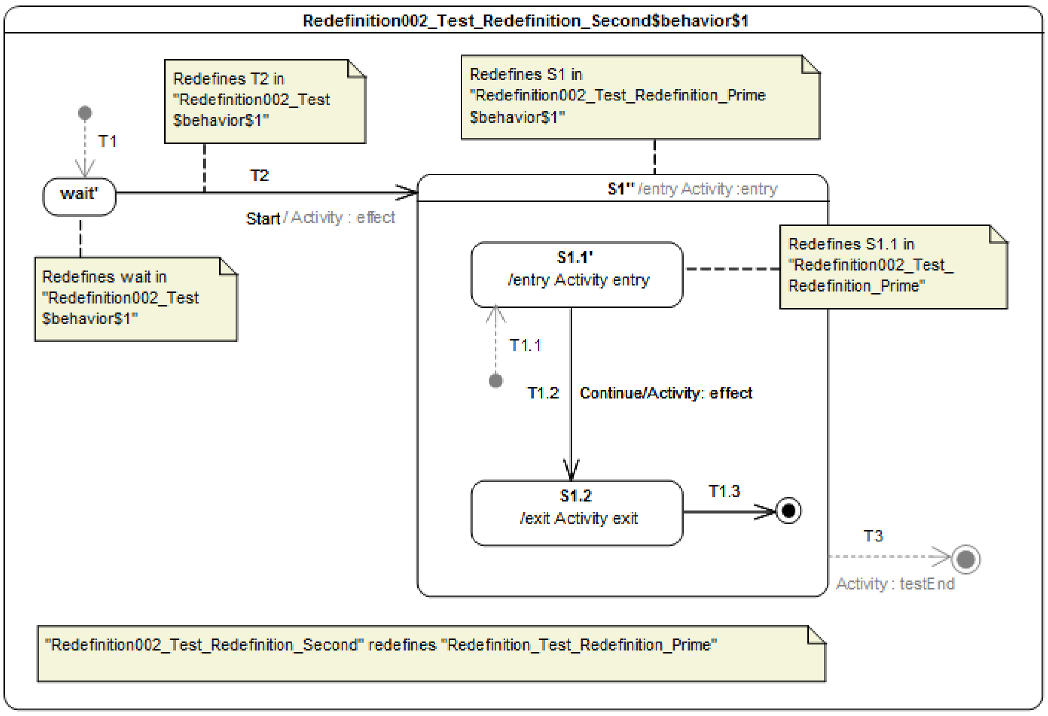

Redefinition002_Test_Redefinition_Second$behavior$1
Redefines T2 in "Redefinition002_Test $behavior$1"
Redefines S1 in "Redefinition002_Test_Redefinition_Prime $behavior$1"
T1
wait'
T2
Start / Activity : effect
S1'' /entry Activity :entry
Redefines wait in "Redefinition002_Test $behavior$1"
S1.1' /entry Activity entry
Redefines S1.1 in "Redefinition002_Test_ Redefinition_Prime"
T1.1
T1.2
Continue/Activity: effect
S1.2 /exit Activity exit
T1.3
T3
Activity : testEnd
"Redefinition002_Test_Redefinition_Second" redefines "Redefinition_Test_Redefinition_Prime"

Figure 9.105 - Redefinition 002 Test Classifier Behavior – Runtime State Machine

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received in configuration *S1''[S1.1']*.

Generated trace

- T2(effect)::S1(entry)-redefined-prime::S1.1(entry)-redefined-second::T1.2(effect)::S1.2(exit)

Note. The purpose of this test is to demonstrate the capability of computing the runtime state machine when multiple levels of extension and redefinition are involved. Note that in this test:

- Transition *T2* defined in *Redefinition002_Test_Redefinition_Second* redefines the transition *T2* that was originally defined in *Redefinition002_Test*. The redefining transition adds a trigger for the *Start* signal and, as it does not define an effect behavior, it inherits the one defined in the redefined transition. Finally, the version of *T2* in the runtime state machine can be fired only when a *Start* event occurrence is dispatched, and its traversal forces the execution of an effect behavior.
- State *S1'* defined in *Redefinition002_Test_Redefinition Prime* provides an entry behavior. Since the redefined version of that state described in *Redefinition002_Test_Redefinition_Second* does not provide an entry behavior, the original one is inherited. This means that when *S1''* is entered, the entry behavior of *S1'* is executed.

RTC steps

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

9.3.17.4 Redefinition 003

Tested state machine

The test target for Redefinition 003 owns three state machines: *Redefinition003_Test, Redefinition003_Test_Redefinition_Prime* and *Redefinition003_Test_Redefinition_Second*. The first state machine is extended by the second one, which is itself extended by the third. The extension and redefinition relationships between the three state machines are shown in Figure 9.106.

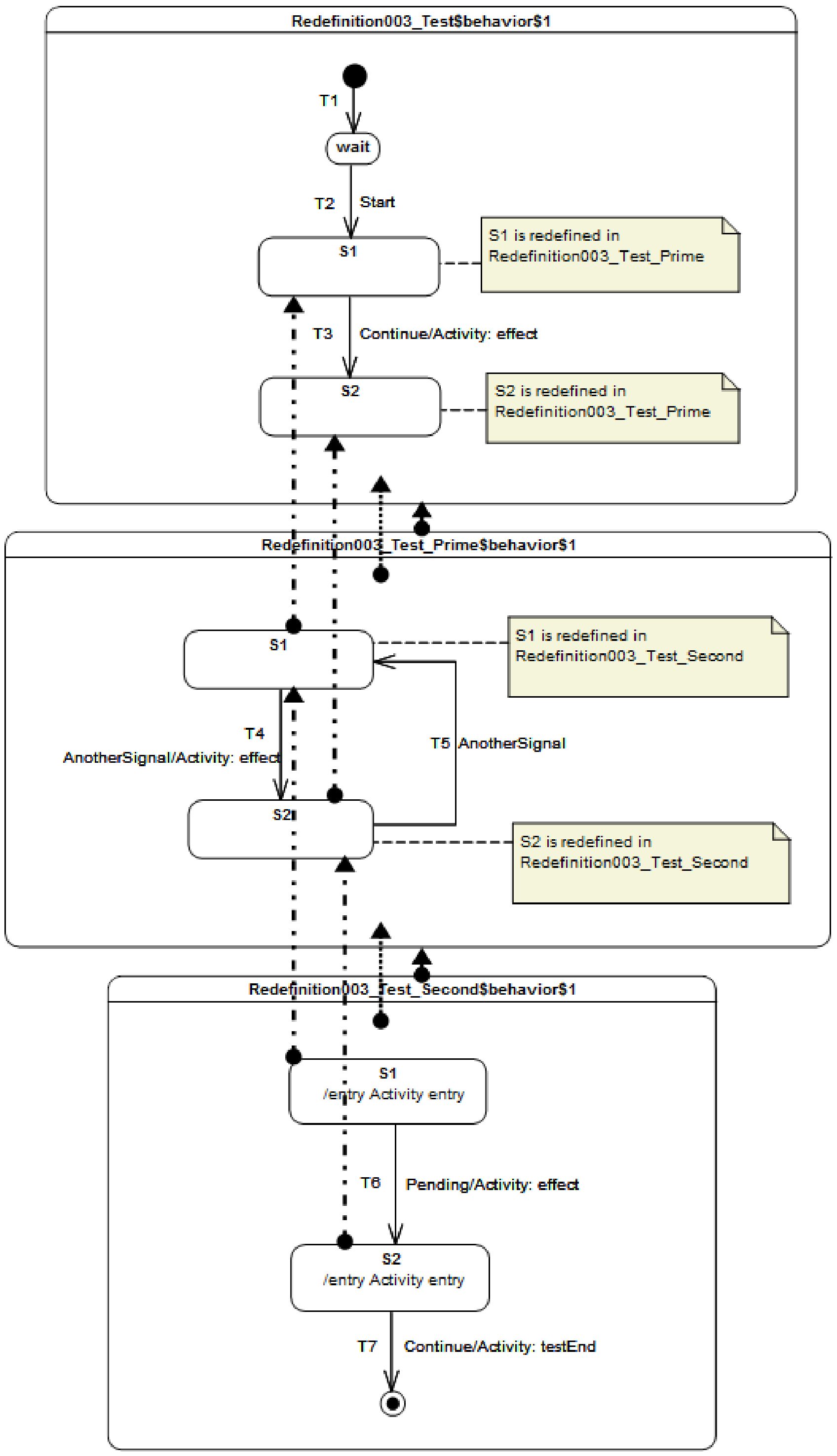

Redefinition003_Test$behavior$1
T1
wait
T2
Start
S1
S1 is redefined in Redefinition003_Test_Prime
T3
Continue/Activity: effect
S2
S2 is redefined in Redefinition003_Test_Prime
Redefinition003_Test_Prime$behavior$1
S1
S1 is redefined in Redefinition003_Test_Second
T4
AnotherSignal/Activity: effect
T5
AnotherSignal
S2
S2 is redefined in Redefinition003_Test_Second
Redefinition003_Test_Second$behavior$1
S1
/entry Activity entry
T6
Pending/Activity: effect
S2
/entry Activity entry
T7
Continue/Activity: testEnd

Figure 9.106 - Extension and Redefinition Relationships for Redefinition 003

The corresponding runtime state machine is presented in Figure 9.107.

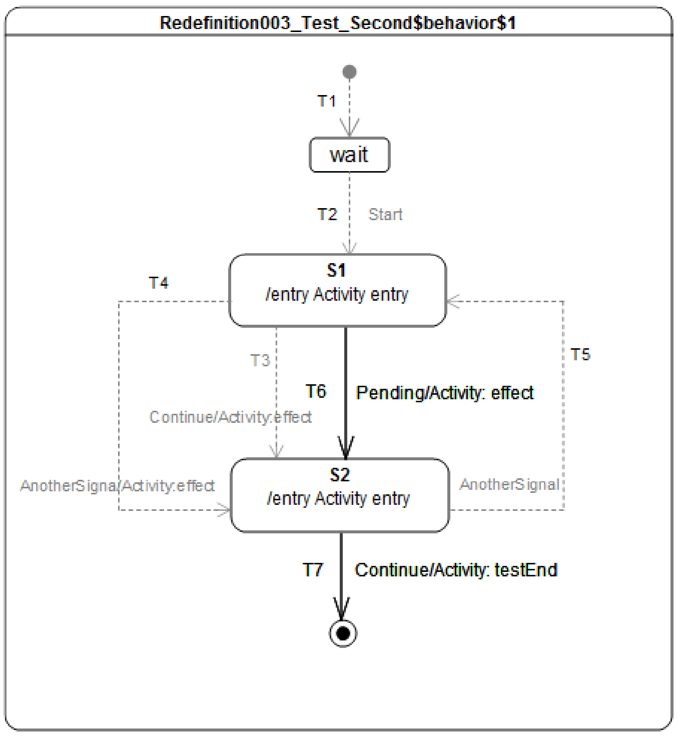


Figure 9.107 - Redefinition 003 Test Classifier Behavior – Runtime State Machine

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1*.
- AnotherSignal – received when in configuration *S2*.
- AnotherSignal – received when in configuration *S1*.
- AnotherSignal – received when in configuration *S2*.
- Pending – received when in configuration *S1*.
- Continue – received when in configuration *S2*.

Generated trace

- S1(entry)-redefined-second::T3(effect)::S2(entry)-redefined-second::S1(entry)-redefined-second::T4(effect)::S2(entry)-redefined-second::S1(entry)-redefined-second::T6(effect)::S2(entry)-redefined-second
- **Note.** The purpose of this test is to demonstrate the capability of computing the runtime state machine in a situation in which each level of extension adds new transitions to states originally defined in the root state machine. Figure 9.107 shows this runtime state machine. Note that no states are added by the successive extensions of *Redefinition003_Test*. However, transitions *T4* and *T5* come from *Redefinition003_Test_Redefinition_Prime* and transitions *T6* and *T7* come from *Redefinition003_Test_Redefinition_Second*. The RTC steps realized during this execution are described below.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2]
4	[Continue, **CE(S1)**]	[S1]	[]
5	[**Continue**]	[S1]	[T3]
6	[AnotherSignal, **CE(S2)**]	[S2]	[]
7	[**AnotherSignal**]	[S2]	[T5]
8	[AnotherSignal, **CE(S1)**]	[S1]	[]
9	[**AnotherSignal**]	[S1]	[T4]
10	[AnotherSignal, **CE(S2)**]	[S2]	[]
11	[**AnotherSignal**]	[S2]	[T5]
12	[Pending, **CE(S1)**]	[S1]	[]
13	[**Pending**]	[S1]	[T6]
14	[Continue, **CE(S2)**]	[S2]	[]
15	[**Continue**]	[S2]	[T7]

9.3.17.5 Redefinition 004

Tested state machine

The test target for Redefinition 004 owns two state machines: *Redefinition004_Test* and *Redefinition004_Test_Redefinition_Prime.* The first state machine is extended by the second. The extension and redefinition relationships between the two state machines are shown in Figure 9.108.

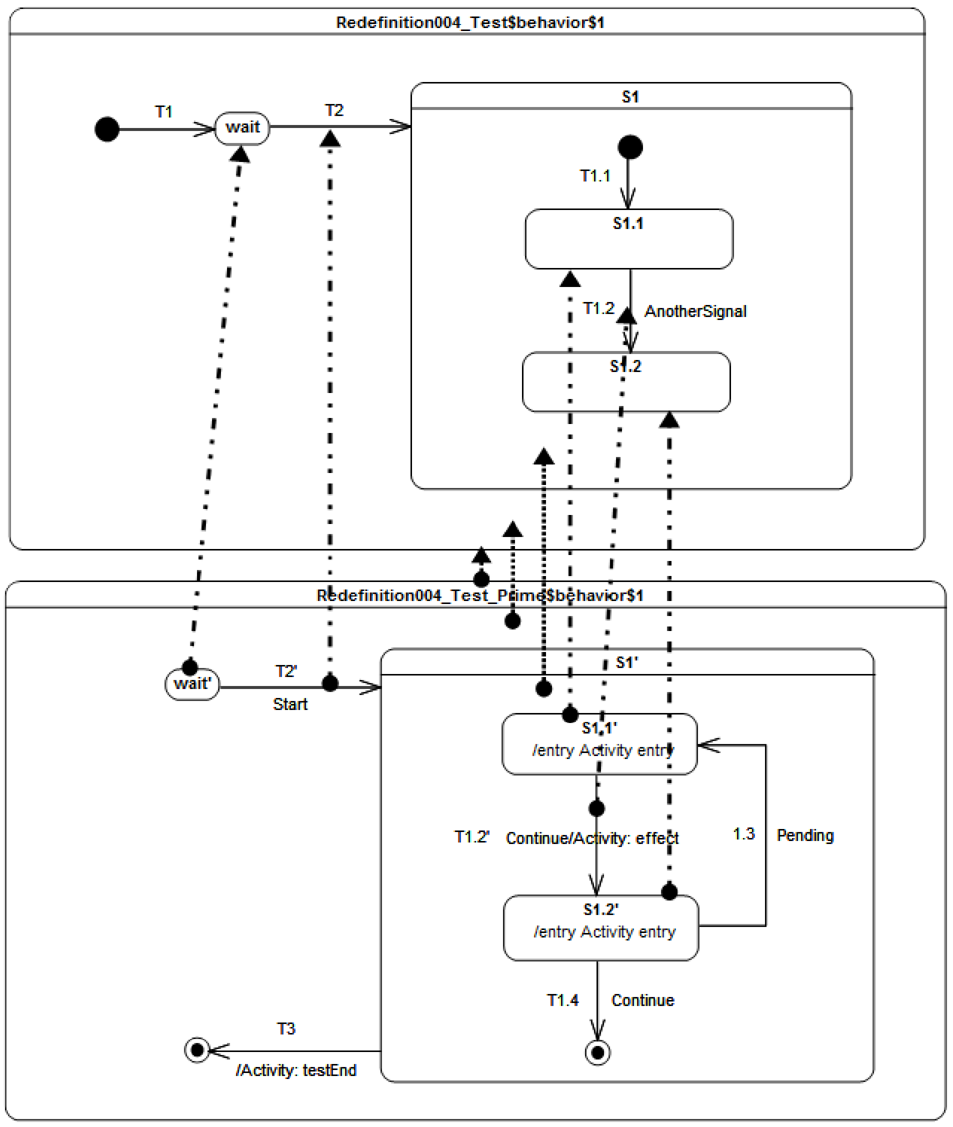


Figure 9.108 - Extension and Redefinition Relationships for Redefinition 004

The corresponding runtime state machine is presented in Figure 9.109.

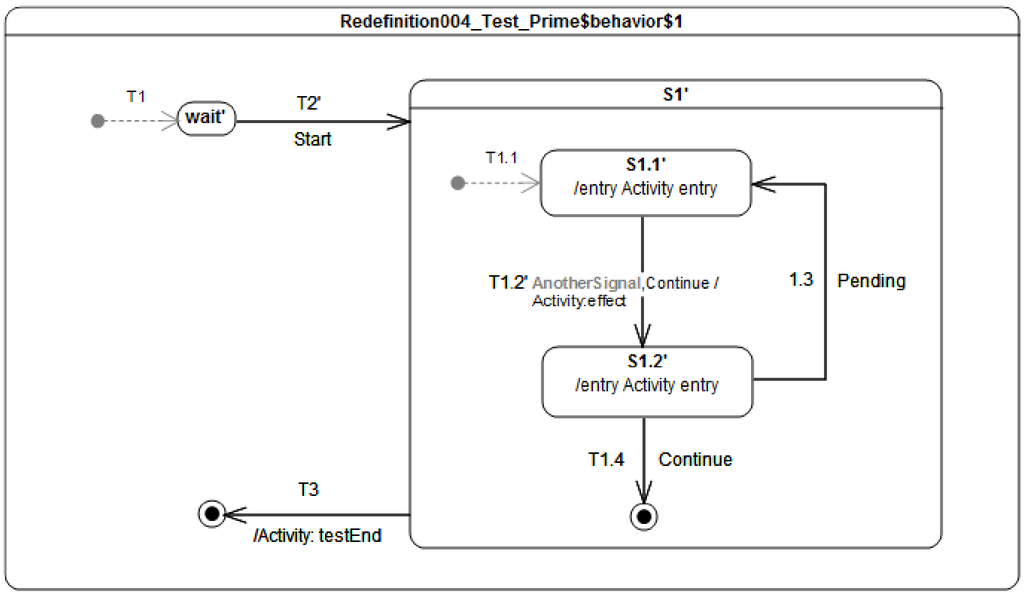


Figure 9.109 - Redefinition 004 Test Classifier Behavior – Runtime State Machine

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1.1'*.
- Pending – received when in configuration S1.2'.
- AnotherSignal – received when in configuration *S1.1'*.
- Continue – received when in configuration *S1.2'*.

Generated trace

- S1.1(entry)-redefined-prime::T1.2(effect)-redefined-prime::S1.2(entry)-redefined-prime::S1.1(entry)-redefined-prime::T1.2(effect)-redefined-prime::S1.2(entry)-redefined-prime

Note. The purpose of this test is to demonstrate that redefining transitions inherit triggers that were declared in directly or indirectly redefined transitions. In this test, *T1.2'*, from *Redefinition004_Test_Prime,* redefines *T1.2* declared in *Redefinition004_Test*. At runtime, *T1.2'* can be fired by the dispatching of either an *AnotherSignal* event occurrence or a *Continue* event occurrence. The trigger for the *AnotherSignal* event is inherited from *T1.2*, which is declared in *Redefinition004_Test*. The triggering of this transition can be observed in the table below at RTC steps 5 and 9.

RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Start, **CE(wait')**]	[wait']	[]
3	[**Start**]	[wait]	[T2'(T1.1)]
4	[Continue, **CE(S1.1')**]	[S1'[S1.1']]	[]
5	[**Continue**]	[S1'[S1.1']]	[T1.2']
6	[Pending, **CE(S1.2')**]	[S1'[S1.2']]	[]
7	[**Pending**]	[S1'[S1.2']]	[T1.3]
8	[AnotherSignal, **CE(S1.1')**]	[S1'[S1.1']]	[]
9	[**AnotherSignal**]	[S1'[S1.1']]	[T1.2']
10	[Continue, **CE(S1.2')**]	[S1'[S1.2']]	[]
11	[**Continue**]	[S1'[S1.2']]	[T1.4]
12	[**CE(S1')**]	[S1']	[T3]

9.3.17.6 Redefinition 005

Tested state machine

The test target for Redefinition 005 owns two state machines: *Redefinition005_Test* and *Redefinition005_Test_Redefinition_Prime.* The first state machine is extended by the second. The extension and redefinition relationships existing between the two state machines are shown in Figure 9.110.

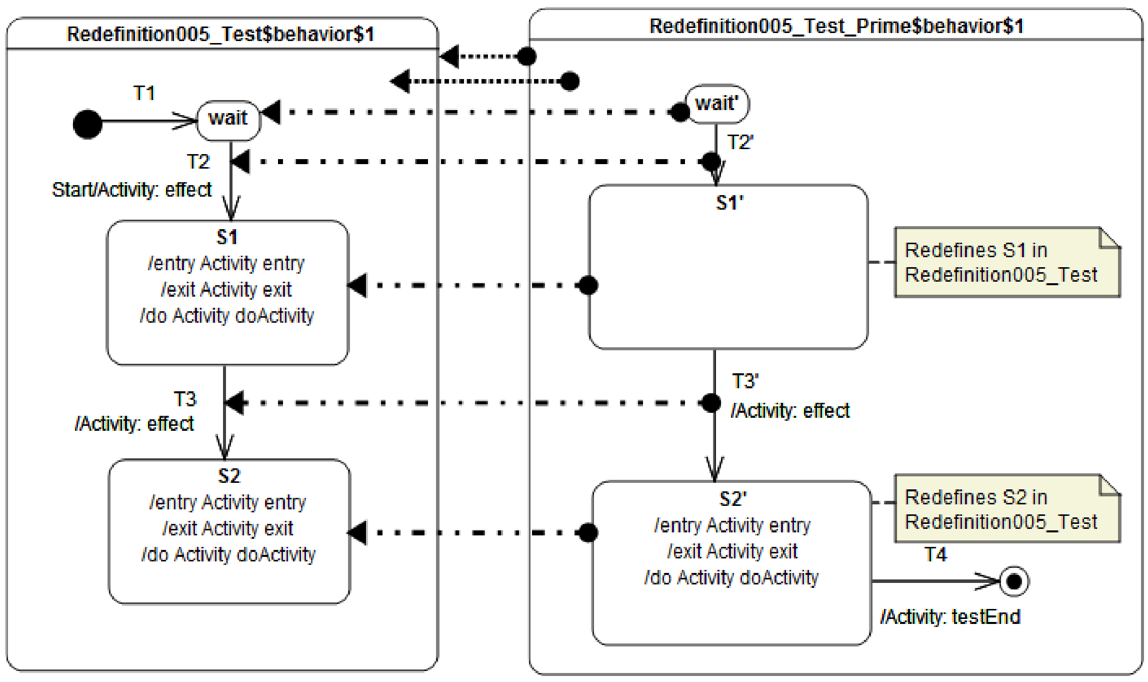


Figure 9.110 - Extension and Redefinition Relationships for Redefinition 005

The corresponding runtime state machine is presented in Figure 9.111.

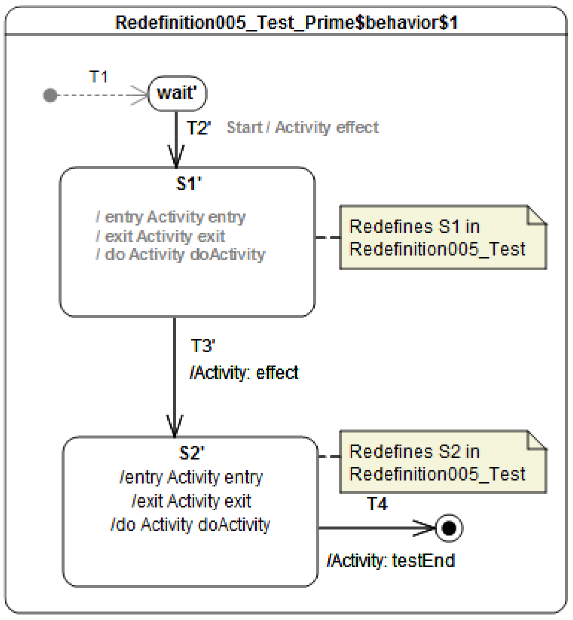


Figure 9.111 - Redefinition 005 Test Classifier Behavior – Runtime State Machine

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.

Generated trace

- T2(effect)::S1(entry)::S1(doActivity)::S1(exit)::T3(effect)-redefined::S2(entry)-redefined-prime::S2(doActivity)-redefined-prime::S2(exit)-redefined-prime

Note. The purpose of this test is to demonstrate two capabilities: First, it shows that, when a redefining state does not define state behaviors (entry, doActivity, exit) but the redefined state does, then the behaviors defined in the redefined state are inherited. Consequently, state *S1'* in the runtime state machine executes the entry behavior of *S1* and invokes its doActivity when entered. In addition, it executes the exit behavior defined in *S1* when exited. Second, the test verifies that, if both the redefining state and the redefined state specify state behaviors, then the behaviors that are executed at runtime are those defined in the redefining state. As an example consider *S2'*. This state defines new entry, doActivity and exit behaviors. Consequently, behaviors defined in *S2* in the extended state machine are not included in the runtime state machine. Note that the aforementioned rules also apply for transition effect behaviors. Such support is shown for *T2'* and *T3'*. Indeed *T2',* when traversed, executes the effect behavior inherited from redefined transition *T2. T3'* declares its own effect behavior and so overrides the one provided by the redefinition transition *T3.*

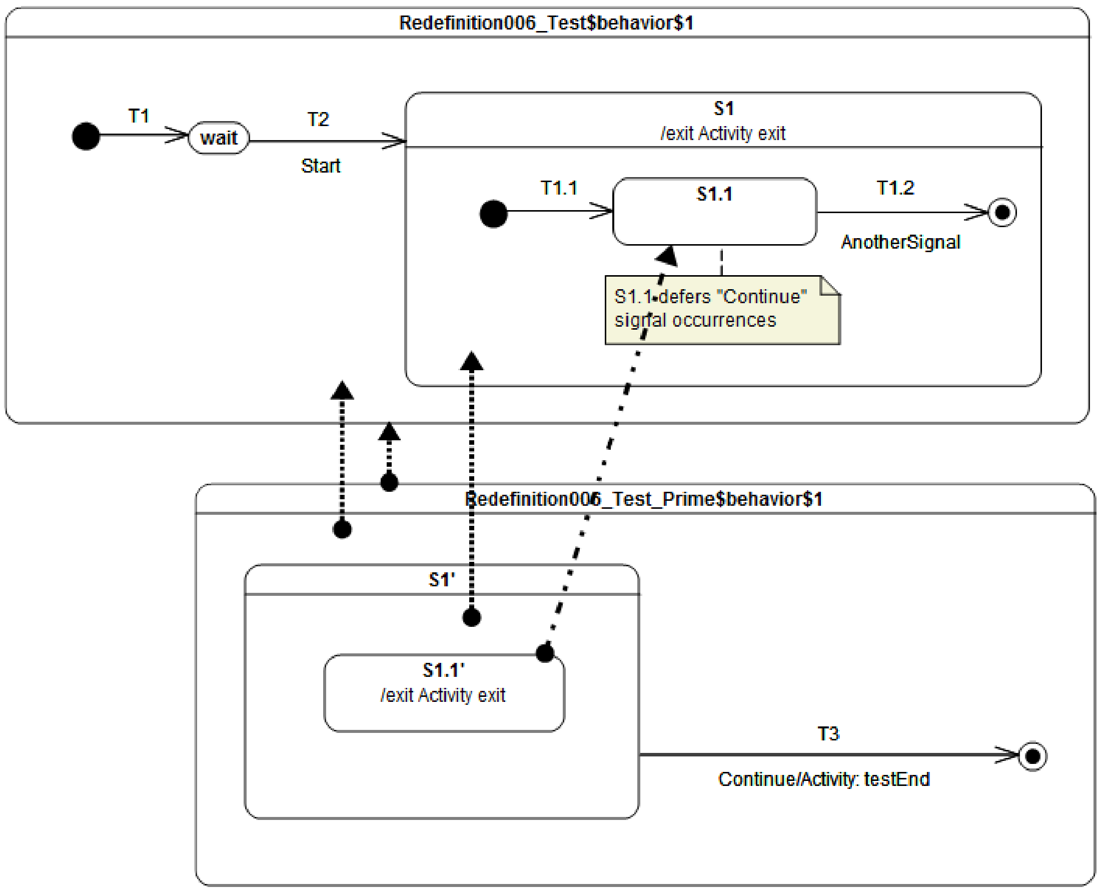
RTC steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2]
4	[**CE(S1')**]	[S1']	[T3]
5	[**CE(S2')**]	[S2']	[T4]

9.3.17.7 Redefinition 006

Tested state machine

The test target for Redefinition 006 owns two state machines: *Redefinition006_Test* and *Redefinition006_Test_Redefinition_Prime.* The first state machine is extended by the second. The extension and redefinition relationships existing between the two state machines are shown in Figure 9.112.



Redefinition006_Test$behavior$1
T1
wait
T2
Start
S1
/exit Activity exit
T1.1
S1.1
T1.2
AnotherSignal
S1.1 defers "Continue" signal occurrences
Redefinition006_Test_Prime$behavior$1
S1'
S1.1'
/exit Activity exit
T3
Continue/Activity: testEnd

Figure 9.112 - Extension and Redefinition Relationships for Redefinition 006

The corresponding runtime state machine is presented in Figure 9.113.

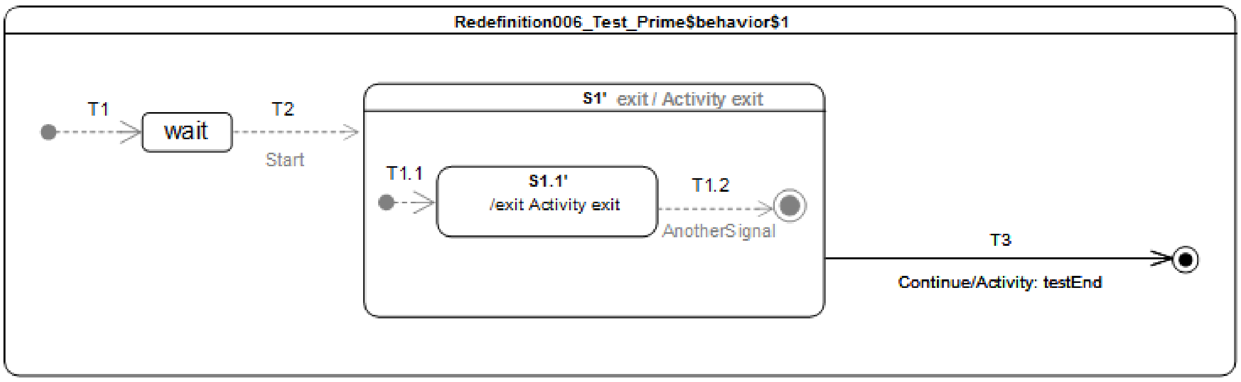


Figure 9.113 - Redefinition 006 Test Classifier Behavior

Test execution

Received event occurrence(s)

- Start – received when in configuration *wait*.
- Continue – received when in configuration *S1'[S1.1']*.
- AnotherSignal – received when in configuration *S1'[S1.1']*.

Generated trace

- S1.1(exit)-redefined-prime::S1(exit)

Note. The purpose of this test is to demonstrate the inheritance of deferrable triggers defined in a redefined state. In this test, *S1.1'* redefines state *S1.1*. Consequently, the deferrable trigger declared by *S1.1* is also deferrable for *S1.1'* at execution time. This means that, when the state machine is in configuration *S1'[S1.1'],* the state machine will still defer the occurrence of a *Continue* signal event. This situation is shown at RTC step 5 in the table below.

RTC steps

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

9.3.18 Standalone

9.3.18.1 Overview

This subclause includes tests related the execution of standalone state machines, that is, state machines that are themselves active behaviors, as opposed to being the classifier behaviors of other classes.

Given that a UML state machine is a kind of UML class, it is legal for this state machine to specialize the *Target* abstract class (see 9.2.2.2.2). Hence the state machine itself is the test target, which means that it is able to receive any signals that a *Target* can receive. Note that the standalone state machine is active (as is required by fUML in order for it to specialize an active class), but it does not have a classifier behavior, meaning that, dynamically, it acts as the context for its own execution.

The tests in this subclause are based on similar tests from other test categories, except for Standalone 001. The purpose here is to demonstrate that, if a state machine is active and does not play the role of a classifier behavior, it can still be executed according to the same semantics.

9.3.18.2 Standalone 001

Tested state machine

The state machine that is executed for this test is presented in Figure 9.114.

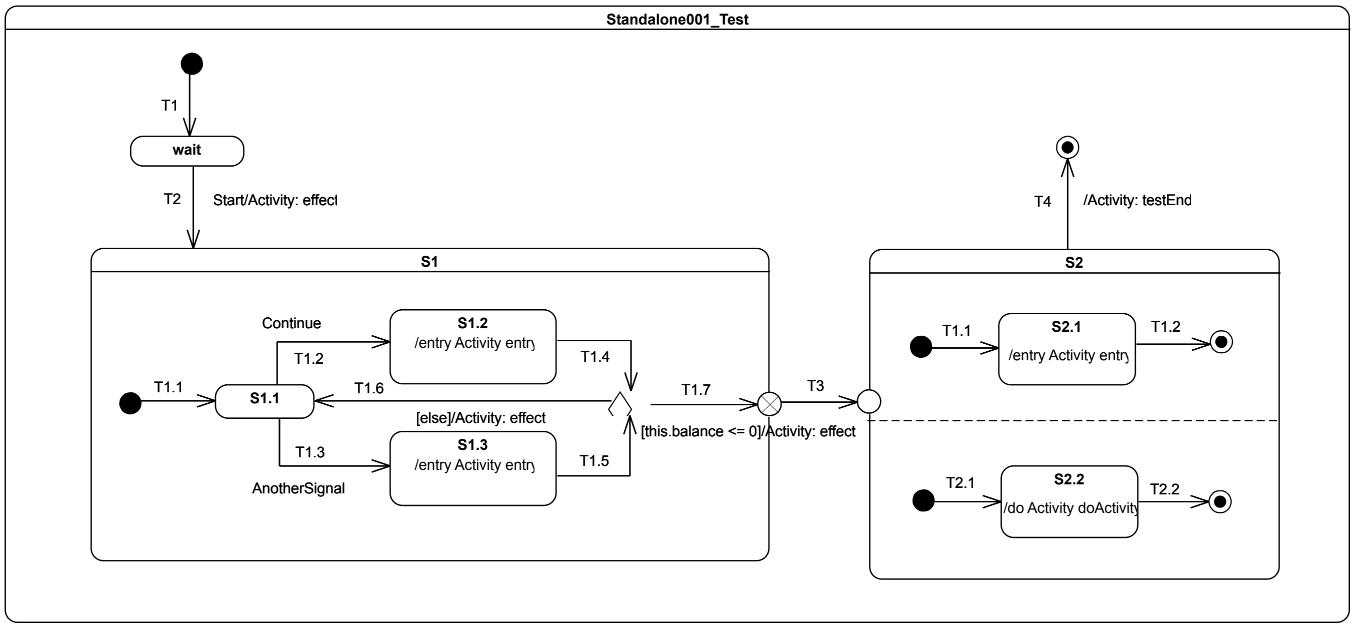


Figure 9.114 - Standalone 001 Test

Test execution

Received event occurrence(s)

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

Generated trace

- T2(effect)::S1.2(entry)::T1.6(effect)::S1.2(entry)::T1.7(effect)::S2.1(entry)::S2.2(doActivity)

Note. The *Start* event occurrence is dispatched and accepted while the state machine is in configuration *wait*. Hence, *T2* is triggered (see message *T2(effect)* in the trace) and *S1* is entered using the default entry approach. The *S1* region starts executing from the initial pseudostate, *T1.1* is traversed, and *S1.1* is entered. The next RTC step is initiated by the acceptance of the *Continue* event occurrence, which triggers *T1.2* and whose traversal leads to *S1.2* being entered. The execution of the *S1.2* entry behavior updates a property *balance* owned by the state machine. When this behavior has terminated its execution, a completion event occurrence is generated for *S1.2*. The completion event occurrence is used to trigger *T1.4*. The state machine reaches the choice point and evaluates the guard of *T1.7*. The *balance* (initial value 150) is not less than or equal to 0, hence the else transition *T1.6* is taken, and a completion event occurrence is generated upon *S1.1* entry. A second *Continue* event occurrence is dispatched and accepted, and the state machine returns to *S1.2,* generating a completion event occurrence for that state. This time, when *T1.4* is triggered, the choice point is reached and the *T1.7* guard is true, so it can be traversed. When the exit point is reached, *S1* is exited. The continuation transition *T3* is then traversed, and *S2* is entered through the entry point. Consequently, both orthogonal regions are entered using the default entry approach. *S2.1* generates a completion event occurrence when this entry behavior terminates its execution, and *S2.2* generates a completion event occurrence when its doActivity has completed. Completion event occurrences generated by these states are used to trigger *T1.2* and *T2.2*. When both completion event occurrences have been dispatched and accepted, *S2* can complete. The completion event occurrence will be used to trigger *T4,* and the final state is reached, which will complete the state machine execution.

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)]
4	[Continue, **CE(S1.1)**]	[S1[S1.1]]	[]
5	[**Continue**]	[S1[S1.1]]	[T1.2]
6	[**CE(S1.2)**]	[S1[S1.2]]	[T1.4(T1.6)]
7	[Continue, **CE(S1.1)**]	[S1[S1.1]]	[]
8	[**Continue**]	[S1[S1.1]]	[T1.2]
9	[**CE(S1.2)**]	[S1[S1.2]]	[T1.4(T1.7, T3, T1.1, T2.1))]
10	[**CE(2.1)**]	[S2[S2.1, S2.2]]	[T1.2]
11	[**CE(S2.2)**]	[S2[S2.2]]	[T2.2]
12	[**CE(S2)**]	[S2]	[T4]

9.3.18.3 Standalone 002

Tested state machine

The state machine that is tested is the one used in Transition 023. See 9.3.3.15.

Test execution

The state machine receives the same stimulation sequence as the one specified in 9.3.3.15 and can generate the same execution traces.

9.3.18.4 Standalone 003

Tested state machine

The state machine that is tested is the one used in Event 019-E. See 9.3.4.17.

Test execution

The state machine receives the same stimulation sequence than the one specified in 9.3.4.17 and can generate the same execution traces.

9.3.19 Other Test

9.3.19.1 Overview

This subclause includes an additional test based on an example from the UML specification. This test assesses that the intended execution semantics of the example are captured by the PSSM execution model.

9.3.19.2 Transition Execution Algorithm Test

Tested state machine

This test is based on the example from [UML], Figure 14.2. The state machine that is executed for the test is presented in Figure 9.115.

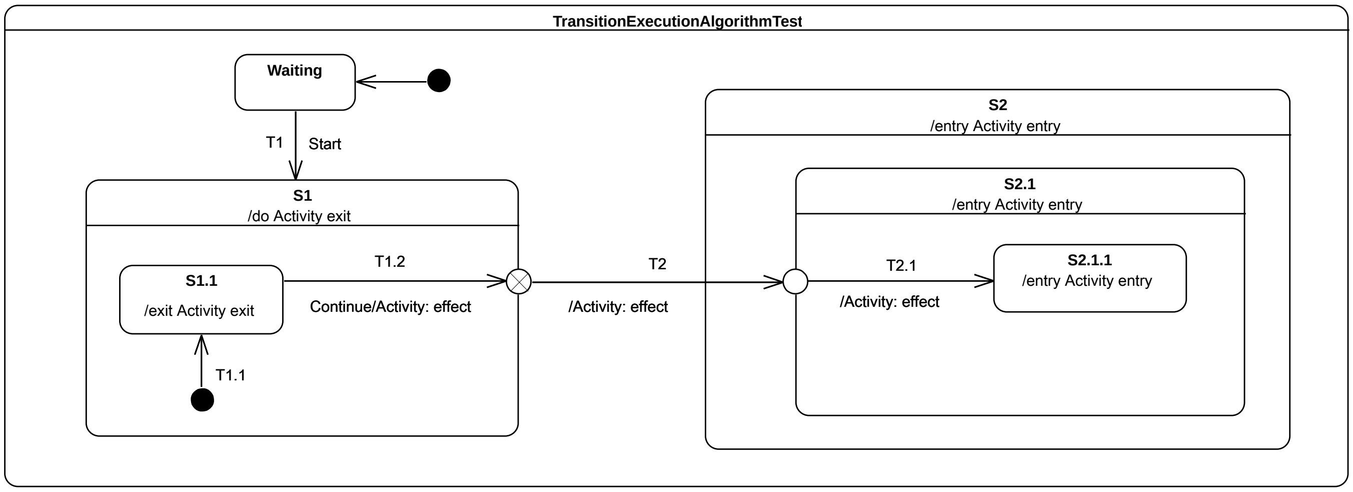


Figure 9.115 - TransitionExecutionAlgorithmTest Classifier Behavior

Test execution

Received event occurrence(s)

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

Generated trace

- S1.1(exit)::T1.2(effect)::S1(exit)::T2(effect)::S2(entry)::S2.1(entry)::T2.1(effect)::S2.1.1(entry)

Note. Consider the situation where the state machine is in configuration *S1[S1.1]*. There are two event occurrences available in the event pool: the first is the completion event occurrence for *S1.1,* and the second is a *Continue* event occurrence. The completion event occurrence gets dispatched first. It does not initiate an RTC step, since *S1.1* has no completion transition and, therefore, the event occurrence is lost. When the *Continue* event occurrence is dispatched, it triggers *T1.2*. State *S1.1* is exited, the *T1.2* effect behavior is executed, and the exit point placed on the edge of S1 is reached. This exit point implies exiting *S1* and traversal of the continuation transition *T2*. This leads the state machine to reach the entry point placed on the edge of *S2.1*. At this point, *S2* is entered first and the continuation transition *T2.1* is traversed next. This means that, at the conclusion of the RTC step, *S2.1.1* is entered and its entry behavior is executed.

RTC Steps

Step	**Event pool**	**State machine configuration**	**Fired transition(s)**
1	[]	[] - Initial RTC step	[InitialTransition]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T1(T1.1)]
4	[Continue, **CE(S1.1)**]	[S1[S1.1]]	[]
5	[**Continue**]	[S1[S1.1]]	[T1.2(T2, T2.1)]
6	[**CE(2.1.1)**]	[S2[S2.1[S2.1.1]]]	[]

9.4 Test Coverage and Traceability

9.4.1 Overview

This subclause presents the complete set of semantic requirements that have been identified for PSSM and describes the coverage of those requirements by the tests in the foregoing test suite (as presented in 9.3). The requirements are grouped according to the same categories as the tests. The requirements for each category are presented in a table that lists, for each requirement, a unique identifier, a description, and references to any related tests (or, alternatively, a note as to why the requirements is not testable).

9.4.2 Behavior

ID	**Description**	**Test(s)**
Behavior 001	*A State may have an associated entry Behavior. This Behavior, if defined, is executed whenever the State is entered through an external Transition.*	See 9.3.2.2
Behavior 002	*A State may also have an associated exit Behavior, which, if defined, is executed whenever the State is exited.*	See 9.3.2.3

ID	Description	Test(s)
Behavior 003	*A State may also have an associated doActivity Behavior. This Behavior commences execution when the State is entered (but only after the State entry Behavior has completed) and executes concurrently with any other Behaviors that may be associated with the State, until it completes (in which case a completion event is generated) or the State is exited, in which case execution of the doActivity Behavior is aborted.*	See 9.3.2.4 and 9.3.2.5.
Behavior 004	*The execution of a doActivity Behavior of a State is not affected by the firing of an internal Transition of that State.*	See 9.3.2.6.

9.4.3 Transition

ID	Description	Test(s)
Transition 001	*A Transition may have an associated effect Behavior, which is executed when the Transition is traversed (executed)*	See 9.3.3.1.
Transition 002	*The duration of a Transition traversal is undefined, allowing for different semantic interpretations, including both "zero" and non-"zero" time.*	See [fUML], 2.4, on the semantics of time in fUML.
Transition 003	*Transitions are executed as part of a more complex compound transition that takes a StateMachine execution from one stable state configuration to another.*	See 9.3.3.11, 9.3.3.15 and 9.3.10.5.
Transition 004	*A transition is said to be reached, when execution of its StateMachine execution has reached its source Vertex (i.e., its source State is in the active state configuration).*	This requirement cannot be tested via the test suite model.
Transition 005	*A transition is said to be traversed, when it is being executed (along with any associated effect Behavior)*	This requirement cannot be tested via the test suite model.
Transition 006	*A transition is said to be completed, after it has reached its target Vertex*	This requirement cannot be tested via the test suite model.
Transition 007	*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 9.3.3.2.
Transition 008	*A Transition trigger is said to be enabled if the dispatched Event occurrence matches its Event type*	See 9.3.3.2, 9.3.3.11, 9.3.3.15 and 9.3.4.8.
Transition 009	*When multiple triggers are defined for a Transition, they are logically disjunctive, that is, if any of them are enabled, the Transition will be triggered*	See 9.3.3.2.

ID	Description	Test(s)
Transition 010	*kind = external means that the Transition exits its source Vertex. If the Vertex is a State, then executing this Transition will result in the execution of any associated exit Behavior of that State*	See 9.3.3.3 and 9.3.3.6. Note that a number of other tests also extensively use external transitions.
Transition 011	*kind = local is the opposite of external, meaning that the Transition does not exit its containing State (and, hence, the exit Behavior of the containing State will not be executed)*	See 9.3.3.4, 9.3.3.5, 9.3.3.7 and 9.3.3.8.
Transition 012	*kind = internal is a special case of a local Transition that is a self-transition (i.e., with the same source and target States), such that the State is never exited (and, thus, not re-entered), which means that no exit or entry Behaviors are executed when this Transition is executed.*	See 9.3.2.6.
Transition 013	*Transitions whose source Vertex is a composite State are called high-level or group Transitions. If they are external, group Transitions result in the exiting of all substates of the composite State, executing any defined exit Behaviors starting with the innermost States in the active state configuration.*	See 9.3.6.2 and 9.3.6.4.
Transition 014	*In case of local Transitions, the exit Behaviors of the source state and the entry Behaviors of the target State will be executed, but not those of the containing State*	See 9.3.3.4 and 9.3.3.7.
Transition 015	*In case of simple States, a completion event is generated when the associated entry and doActivity Behaviors have completed executing*	See 9.3.3.9.
Transition 016	*If no such Behaviors are defined, the completion event is generated upon entry into the State.*	See 9.3.3.10.
Transition 017	*For composite States, a completion event is generated under the following circumstances: All internal activities (e.g., entry and doActivity Behaviors) have completed execution, and all its orthogonal Regions have reached a FinalState*	See 9.3.3.11.
Transition 019	*If two or more completion events corresponding to multiple orthogonal Regions occur simultaneously (i.e., as a result of the same Event occurrence), the order in which such completion occurrences are processed is not defined.*	See 9.3.3.12.
Transition 020	*Completion events have dispatching priority. That is, they are dispatched ahead of any pending Event occurrences in the event pool.*	See 9.3.3.13.
Transition 021	*Completion of all top level Regions in a StateMachine corresponds to a completion of the Behavior of the StateMachine and results in its termination.*	See 9.3.14.1.
Transition 022	*A Transition may have an associated guard Constraint. Transitions that have a guard which evaluates to false are disabled.*	See 9.3.3.14.

ID	Description	Test(s)
Transition 023	*Guards are evaluated before the compound transition that contains them is enabled, unless they are on Transitions that originate from a choice Pseudostate*	See 9.3.3.15, but also 9.3.7.6 9.3.10.5.
Transition 024	*In the latter case, the guards are evaluated when the choice point is reached*	See 9.3.9.6.
Transition 025	*A Transition that does not have an associated guard is treated as if it has a guard that is always true.*	See 9.3.3.2, 9.3.3.11 and 9.3.9.3.
Transition 026	*Branching in a compound transition execution occurs whenever an executing Transition performs a default entry into a State with multiple orthogonal Regions, with a separate branch created for each Region, or when a fork Pseudostate is encountered. The overall behavior that results from the execution of a compound transition is a partially ordered set of executions of Behaviors associated with the traversed elements, determined by the order in which the elements (Vertices and Transitions) are encountered*	See 9.3.3.11, 9.3.4.9 and 9.3.5.6
Transition 027	*If a choice or join point is reached with multiple outgoing Transitions with guards, a Transition whose guard evaluates to true will be taken. If more than one guard evaluates to true, one of these Transitions is chosen for continuing the traversal. The algorithm for making this selection is undefined. (p.329)*	See 9.3.9.3 (for choice) and 9.3.12.4 (for join).

9.4.4 Event

ID	Description	Test(s)
Event 001	*Upon creation, a StateMachine will perform its initialization during which it executes an initial compound transition prompted by the creation, after which it enters a wait point. In case of StateMachine Behaviors, a wait point is represented by a stable state configuration. It remains thus until an Event stored in its event pool is dispatched.*	See 9.3.4.2. Note that all tests start executing as described in this requirement.
Event 002	*This Event is evaluated and, if it matches a valid Trigger of the StateMachine and there is at least one enabled Transition that can be triggered by that Event occurrence, a single StateMachine step is executed.*	See 9.3.4.3, but also 9.3.4.9, 9.3.4.14 and 9.3.10.7
Event 003	*No dedicated test is provided for this requirement. Nevertheless its support is demonstrated through various tests in this suite.*	See 9.3.4.9, 9.3.4.14 and 9.3.8.2.

ID	Description	Test(s)
Event 004	*It is possible for multiple mutually exclusive Transitions in a given Region to be enabled for firing by the same Event occurrence. In those cases, only one is selected and executed. Which of the enabled Transitions is chosen is determined by the Transition selection algorithm described below. The set of Transitions that will fire are the Transitions in the Regions of the current state configuration that satisfy the following conditions: All Transitions in the set are enabled. There are no conflicting Transitions within the set. There is no Transition outside the set that has higher priority than a Transition in the set (that is, enabled Transitions with highest priorities are in the set while conflicting Transitions with lower priorities are left out).*	There is no dedicated test for this requirement, but it is supported by the tests in 9.3.4.7 and 9.3.4.9.
Event 005	*StateMachines can respond to any of the Event types described in Clause 13 as well as to completion events.*	See 9.3.4.10, 9.3.4.11, 9.3.4.13, 9.3.4.14, 9.3.4.15 and 9.3.4.16.
Event 006	*Event occurrences are detected, dispatched, and processed by the StateMachine execution, one at a time.*	This is covered by fUML CommonBehavior semantics that are not changed by PSSM. The required behavior can be observed in all PSSM tests.
Event 007	*Run-to-completion means that, in the absence of exceptions or asynchronous destruction of the context Classifier object or the StateMachine execution, a pending Event occurrence is dispatched only after the processing of the previous occurrence is completed and a stable state configuration has been reached. That is, an Event occurrence will never be dispatched while the StateMachine execution is busy processing the previous one.*	This is covered by fUML CommonBehavior semantics that are not changed by PSSM. The required behavior can be observed in all PSSM tests.
Event 008	*When an Event occurrence is detected and dispatched, it may result in one or more Transitions being enabled for firing. If no Transition is enabled and the corresponding Event type is not in any of the deferrableTriggers lists of the active state configuration, the dispatched Event occurrence is discarded and the run-to-completion step is completed trivially.*	See 9.3.4.4.
Event 009	*It is possible that multiple Transitions (in different Regions) can be triggered by the same Event occurrence. The order in which these Transitions are executed is left undefined.*	See 9.3.4.5. (But see also 9.3.4.9.)
Event 010	*it is possible for multiple mutually exclusive Transitions in a given Region to be enabled for firing by the same Event occurrence. In those cases, only one is selected and executed. Which of the enabled Transitions is chosen is determined by the Transition selection algorithm described below.*	See 9.3.4.6.

ID	Description	Test(s)
Event 011	*When all orthogonal Regions have finished executing the Transition, the current Event occurrence is fully consumed, and the run-to-completion step is completed.*	There is no dedicated test for this requirement, but it is supported by the test in 9.3.4.5.
Event 013	*During a Transition, a number of actions Behaviors may be executed. If such a Behavior includes a synchronous invocation call on another object executing a StateMachine, then the Transition step is not completed until the invoked object method completes its run-to-completion step. (p.330).*	See 9.3.4.14, 9.3.4.15, 9.3.4.16 and 9.3.4.17.
Event 014	*A Transition is enabled if and only if: 1 All of its source States are in the active state configuration. 2 At least one of the triggers of the Transition has an Event that is matched by the Event type of the dispatched Event occurrence. In case of Signal Events, any occurrence of the same or compatible type as specified in the Trigger will match. If one of the Triggers is for an AnyReceiveEvent, then either a Signal or CallEvent satisfies this Trigger, provided that there is no other Signal or CallEvent Trigger for the same Transition or any other Transition having the same source Vertex as the Transition with the AnyReceiveEvent trigger (see also 13.3.1). 3 If there exists at least one full path from the source state configuration to either the target state configuration or to a dynamic choice Pseudostate in which all guard conditions are true (Transitions without guards are treated as if their guards are always true).*	AnyReceiveEvents are not included in PSSM. However for signal event support see 9.3.4.10, for call event support see 9.3.4.17, for static analysis (path analysis) see 9.3.10.4 and 9.3.3.15.
Event 015	*It is possible for more than one Transition to be enabled within a StateMachine. If that happens, then such Transitions may be in conflict with each other. For example, consider the case of two Transitions originating from the same State, triggered by the same event, but with different guards. If that event occurs and both guard conditions are true, then at most one of those Transition can fire in a given run-to-completion step*	See 9.3.4.7.
Event 016	*In situations where there are conflicting Transitions, the selection of which Transitions will fire is based in part on an implicit priority. These priorities resolve some but not all Transition conflicts, as they only define a partial ordering. The priorities of conflicting Transitions are based on their relative position in the state hierarchy. By definition, a Transition originating from a substate has higher priority than a conflicting Transition originating from any of its containing States. The priority of a Transition is defined based on its source State.*	See 9.3.4.8 and 9.3.4.9.
Event 017	*The priority of Transitions chained in a compound transition is based on the priority of the Transition with the most deeply nested source State.*	See 9.3.4.8.

ID	Description	Test(s)
Event 018	*Once a Transition is enabled and is selected to fire, the following steps are carried out in order: 1. Starting with the main source State, the States that contain the main source State are exited according to the rules of State exit (or, composite State exit if the main source State is nested) as described earlier. 2. The series of State exits continues until the first Region that contains, directly or indirectly, both the main source and main target states is reached. The Region that contains both the main source and main target states is called their least common ancestor. At that point, the effect Behavior of the Transition that connects the sub-configuration of source States to the sub-configuration of target States is executed. (A "sub-configuration" here refers to that subset of a full state configuration contained within the least common ancestor Region.) 3. The configuration of States containing the main target State is entered, starting with the outermost State in the least common ancestor Region that contains the main target State. The execution of Behaviors follows the rules of State entry (or composite State entry) described earlier.*	See 9.3.4.12 and 9.3.19.2.

9.4.5 Entering

ID	Description	Test(s)
Entering 001	*The rule for this case is the same as for shallow history except that the target Pseudostate is of type deepHistory and the rule is applied recursively to all levels in the active state configuration below this one.*	See 9.3.15.2.
Entering 002	*If a doActivity Behavior is defined for the State, this Behavior commences execution immediately after the entry Behavior is executed. It executes concurrently with any subsequent Behaviors associated with entering the State, such as the entry Behaviors of substates entered as part of the same compound transition.*	There is no dedicated test for this requirement, but it is supported by the tests in 9.3.2.4 and 9.3.2.5.
Entering 003	*If the incoming Transition terminates on a shallowHistory Pseudostate of a Region of the composite State, the active substate becomes the substate that was most recently active prior to this entry.*	See 9.3.15.5.
Entering 004	*If no initial Pseudostate is defined, there is no single approach defined. One alternative is to treat such a model as ill formed. A second alternative is to treat the composite State as a simple State, terminating the traversal on that State despite its internal parts.*	See 9.3.5.2.
Entering 005	*If the incoming Transition or its continuations terminate on a directly contained substate of the composite State, then that substate becomes active and its entry Behavior is executed after the execution of the entry Behavior of the containing composite State. This rule applies recursively if the Transition terminates on an indirect (deeply nested) substate.*	See 9.3.5.3, Also supported by 9.3.5.5 and 9.3.19.2.

ID	Description	Test(s)
Entering 006	*Rules described in Entering_001 do not apply in the case where the most recently active substate is the FinalState, or this is the first entry into this State. In the latter two cases, if a default shallow history Transition is defined originating from the shallowHistory Pseudostate, it will be taken. Otherwise, default State entry is applied.*	See 9.3.15.8 and 9.3.15.9.
Entering 007	*If a Transition enters a composite State through an entryPoint Pseudostate, then the effect Behavior associated with the outgoing Transition originating from the entry point and penetrating into the State (but after the entry Behavior of the composite State has been executed).*	See 9.3.5.4. Also supported by 9.3.19.2.
Entering 008	*If the composite State is also an orthogonal State with multiple Regions, each of its Regions is also entered, either by default or explicitly.*	See 9.3.5.5.
Entering 009	*If the Transition terminates on the edge of the composite State (i.e., without entering the State), then all the Regions are entered using the default entry rule above.*	See 9.3.5.6. Also supported by 9.3.4.9 and 9.3.3.11.
Entering 010	*If the Transition explicitly enters one or more Regions (in case of a fork), these Regions are entered explicitly and the others by default.*	See 9.3.11.2.
Entering 011	*Regardless of how a State is entered, the StateMachine is deemed to be "in" that State even before any entry Behavior or effect Behavior (if defined) of that State start executing.*	This requirement cannot be tested via the test suite model.

9.4.6 Exiting

ID	Description	Test(s)
Exiting 001	*When exiting a State, regardless of whether it is simple or composite, the final step involved in the exit, after all other Behaviors associated with the exit are completed, is the execution of the exit Behavior of that State.*	See 9.3.6.2.
Exiting 002	*If the State has a doActivity Behavior that is still executing when the State is exited, that Behavior is aborted before the exit Behavior commences execution*	See 9.3.6.3.
Exiting 003	*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 9.3.6.4.
Exiting 004	*If the exit occurs through an exitPoint Pseudostate, then the exit Behavior of the State is executed after the effect Behavior of the Transition terminating on the exit point.*	See 9.3.6.5.

ID	Description	Test(s)
Exiting 005	*When exiting from an orthogonal State, each of its Regions is exited. After that, the exit Behavior of the State is executed*	See 9.3.6.6.
Exiting 006	*Regardless of how a State is exited, the StateMachine is deemed to have "left" that State only after the exit Behavior (if defined) of that State has completed execution.*	This requirement cannot be tested via the test suite model.

9.4.7 Encapsulated

ID	Description	Test(s)
Encaps 001	*Entry points represent termination points (sources) for incoming Transitions and origination points (targets) for Transitions that terminate on some internal Vertex of the composite State. In effect, the latter is a continuation of the external incoming Transition, with the proviso that the execution of the entry Behavior of the composite State (if defined) occurs between the effect Behavior of the incoming Transition and the effect Behavior of the outgoing Transition.*	See 9.3.7.4
Encaps 002	*If there is no outgoing Transition inside the composite State, then the incoming Transition simply performs a default State entry.*	See 9.3.7.5.
Encaps 003	*Exit points are the inverse of entry points. That is, Transitions originating from a Vertex within the composite State can terminate on the exit point. In a well-formed model, such a Transition should have a corresponding external Transition outgoing from the same exit point, representing a continuation of the terminating Transition. If the composite State has an exit Behavior defined, it is executed after any effect Behavior of the incoming inside Transition and before any effect Behavior of the outgoing external Transition.*	See 9.3.8.2.

9.4.8 Entry

ID	Description	Test(s)
Entry 001	*If the owning State has an associated entry Behavior, this Behavior is executed before any behavior associated with the outgoing Transition.*	See 9.3.7.4 and 9.3.7.7.
Entry 002	*In addition to Entry 001, if multiple Regions are involved, the entry point acts as a fork Pseudostate.*	See 9.3.7.2.

9.4.9 Exit

ID	Description	Test(s)
Exit 001	*Transitions terminating on an exit point within any Region of the composite State implies exiting of this composite (with execution of its associated exit Behavior).*	See 9.3.8.2.
Exit 002	*If multiple Transitions from orthogonal Regions within the State terminate on this Pseudostate, then it acts like a join Pseudostate.*	See 9.3.8.3.

9.4.10 Choice

ID	Description	Test(s)
Choice 001	*The guard Constraints on all outgoing Transitions are evaluated dynamically, when the compound transition traversal reaches this Pseudostate.*	See 9.3.9.2.
Choice 002	*If more than one guard evaluates to true, one of the corresponding Transitions is selected. The algorithm for making this selection is not defined.*	See 9.3.9.3.
Choice 003	*If none of the guards evaluates to true, then the model is considered ill formed. To avoid this, it is recommended to define one outgoing Transition with the predefined "else" guard for every choice Pseudostate.*	See 9.3.9.4.

9.4.11 Junction

ID	Description	Test(s)
Junction 001	*Junction pseudo state can be used to split an incoming Transition into multiple outgoing Transition segments with different guard Constraints. Such guard Constraints are evaluated before any compound transition containing this Pseudostate is executed*	See 9.3.10.3 and 9.3.10.5.
Junction 002	*It may happen that, for a particular compound transition, the configuration of Transition paths and guard values is such that the compound transition is prevented from reaching a valid state configuration. In those cases, the entire compound transition is disabled even though its Triggers are enabled*	See 9.3.10.4 and 9.3.10.5.
Junction 003	*If more than one guard evaluates to true, one of these is chosen. The algorithm for making this selection is not defined.*	See 9.3.10.4.

9.4.12 Join

ID	Description	Test(s)
Join 001	*All incoming Transitions have to complete before execution can continue through an outgoing Transition.*	See 9.3.12.2 and 9.3.12.3.

9.4.13 Terminate

ID	Description	Test(s)
Terminate 001	*Entering a terminate Pseudostate implies that the execution of the StateMachine is terminated immediately. The StateMachine does not exit any States nor does it perform any exit Behaviors.*	See 9.3.13.2 and 9.3.13.4.
Terminate 002	*Any executing doActivity Behaviors are automatically aborted. Entering a terminate Pseudostate is equivalent to invoking a DestroyObjectAction.*	See 9.3.13.3.

9.4.14 Final

ID	Description	Test(s)
Final 001	*FinalState is a special kind of State signifying that the enclosing Region has completed. Thus, a Transition to a FinalState represents the completion of the behaviors of the Region containing the FinalState.*	See 9.3.14.2.

9.4.15 History

ID	Description	Test(s)
History 001	*Deep history (deepHistory) represents the full state configuration of the most recent visit to the containing Region. The effect is the same as if the Transition terminating on the deepHistory Pseudostate had, instead, terminated on the innermost State of the preserved state configuration, including execution of all entry Behaviors encountered along the way*	See 9.3.15.2. Also supported by 9.3.15.3.
History 002	*In cases where a Transition terminates on a history Pseudostate when the State has not been entered before (i.e., no prior history) or it had reached its FinalState, there is an option to force a transition to a specific substate, using the default history mechanism. This is a Transition that originates in the history Pseudostate and terminates on a specific Vertex (the default history state) of the Region containing the history Pseudostate. This Transition is only taken if execution leads to the history Pseudostate and the State had never been active before. Otherwise, the appropriate history entry into the Region is executed (see above)*	See 9.3.15.3 and 9.3.15.9.

ID	Description	Test(s)
History 003	*If no default history Transition is defined, then standard default entry of the Region is performed*	See 9.3.15.8.
History 004	*A Transition terminating on this Pseudostate implies restoring the Region to that same state configuration, but with all the semantics of entering a State (see the Subclause describing State entry). The entry Behaviors of all States in the restored state configuration are performed in the appropriate order starting with the outermost State*	See 9.3.15.2 and 9.3.15.3.
History 005	*Represents the most recent active substate of its containing Region, but not the substates of that substate. A Transition terminating on this Pseudostate implies restoring the Region to that substate with all the semantics of entering a State. A single outgoing Transition from this Pseudostate may be defined terminating on a substate of the composite State. This substate is the default shallow history state of the composite State.*	See 9.3.15.6 and 9.3.15.9.

9.4.16 Deferred

The PSSM semantics covers the usual functionality of States with `deferrableTriggers`, as captured in requirements Deferred 001 to Deferred 003, below (see the discussion under StateActivation in 8.5.5). However, PSSM also adopts some special functionality to allow `deferredTriggers` to be used to permit `doActivities` executed by a State to consume Event occurrences without competing with the execution of the containing StateMachine (see the discussion under DoActivityContextObject in 8.5.6). This functionality is captured in the additional requirements Deferred 004 and Deferred 005, below, in order to show the traceability to certain tests that validate support for the required capabilities.

ID	Description	Test(s)
Deferred 001	*A State may specify a set of Event types that may be deferred in that State. This means that Event occurrences of those types will not be dispatched as long as that State remains active. Instead, these Event occurrences remain in the event pool until a state configuration is reached where these Event types are no longer deferred.*	See 9.3.16.2 for SignalEvent deferral and 9.3.16.11 for CallEvent deferral. Also supported by 9.3.16.5, 9.3.16.6 and 9.3.16.7.
Deferred 002	*An otherwise deferrable Event occurrence will instead be dispatched if a deferred Event type is used explicitly in a Trigger of a Transition whose source is the deferring State.*	See 9.3.16.3.
Deferred 003	*An Event may be deferred by a composite State, in which case it remains deferred as long as the composite State remains in the active configuration*	See 9.3.16.4.
Deferred 004	*An otherwise deferrable Event occurrence will instead be dispatched if there is an executing doActivity invoked from the deferring State that is able to accept the Event occurrence.*	See 9.3.16.8.

ID	Description	Test(s)
Deferred 005	*If an Event occurrence has been deferred by a State, but an executing doActivity invoked by that State becomes able to accept that Event occurrence, then the Event occurrence will be dispatched to the doActivity, even though it was previously deferred.*	See 9.3.16.9.

9.4.17 Region

ID	Description	Test(s)
Region 001	*A Region becomes active (i.e., it begins executing) either when its owning State is entered or, if it is directly owned by a StateMachine (i.e., it is a top level Region), when its owning StateMachine starts executing.*	See 9.3.5.6 and 9.3.14.2. Also supported by 9.3.4.9 and 9.3.3.11.
Region 002	*A default activation of a Region occurs if the Region is entered implicitly, that is, it is not entered through an incoming Transition that terminates on one of its component Vertices (e.g., a State or a history Pseudostate), but either through a (local or external) Transition that terminates on the containing State or, in case of a top level Region, when the StateMachine starts executing. Default activation means that execution starts with the Transition originating from the initial Pseudostate of the Region, if one is defined. no specific approach is defined if there is no initial Pseudostate exists within the Region. One possible approach is to deem the model ill defined. An alternative is that the Region remains inactive, although the State that contains it is active. In other words, the containing composite State is treated as a simple (leaf) State.*	No dedicated test is provided for this requirement. Nevertheless support for it is demonstrated by 9.3.5.6 9.3.7.5, 9.3.5.2, 9.3.3.11 and 9.3.4.17.
Region 003	*An explicit activation occurs when a Region is entered by a Transition terminating on one of the Region's contained Vertices. When one Region of an orthogonal State is activated explicitly, this will result in the default activation of all of its orthogonal Regions, unless those Regions are also entered explicitly (multiple orthogonal Regions can be entered explicitly in parallel through Transitions originating from the same fork Pseudostate).*	See 9.3.7.3, 9.3.11.2, 9.3.11.3 and 9.3.5.5.

9.4.18 Configuration

ID	Description	Test(s)
Config 001	*A particular "state" of an executing StateMachine instance is represented by one or more hierarchies of States, starting with the topmost Regions of the StateMachine and down through the composition hierarchy to the simple, or leaf, States. Similarly, we can talk about such a hierarchy of substates within a composite State. This complex hierarchy of States is referred to as a state configuration (of a State or a StateMachine)*	This requirement cannot be tested via the test suite model.
Config 002	*An executing StateMachine instance can only be in exactly one state configuration at a time, which is referred to as its active state configuration*	This requirement cannot be tested via the test suite model.

ID	Description	Test(s)
Config 003	*A State is said to be active if it is part of the active state configuration.*	This requirement cannot be tested via the test suite model.
Config 004	*A state configuration is said to be stable when no further Transitions from that state configuration are enabled and all the entry Behaviors of that configuration, if present, have completed (but not necessarily the doActivity Behaviors of that configuration, which, if defined, may continue executing). A configuration is deemed stable even if there are deferred, completion, or any other types of Event occurrences pending in the event pool of that StateMachine*	This requirement cannot be tested via the test suite model.
Config 005	*After it has been created and completed its initial Transition, a StateMachine is always "in" some state configuration. However, because States can be hierarchical and because there can be Behaviors associated with both Transitions and States, "entering" a hierarchical state configuration involves a dynamic process that terminates only after a stable state configuration (as defined above) is reached.*	This requirement cannot be tested via the test suite model.

9.4.19 Redefinition

ID	Description	Test(s)
Redefinition 001	*A specialized StateMachine will have all the elements of the general StateMachine, and it may include additional elements. Regions may be added. Inherited Regions may be redefined by extension: States and Vertices are inherited, and States and Transitions of the Regions of the StateMachine may be redefined.*	See 9.3.17.2, 9.3.17.3 and 9.3.17.4 . Note that other tests in 9.3.17 also demonstrate support for this requirement.
Redefinition 002	*A simple State may be redefined (extended) to become a composite State by one or more Regions. A composite State can be redefined (extended) by: adding new Regions, adding Vertices and Transitions to inherited Regions, adding entry/exit/doActivity Behaviors, if the general State does not have any, redefining States and Transitions.*	See 9.3.17.2, 9.3.17.3 and 9.3.17.4 . Note that other tests in 9.3.17 also demonstrate support for this requirement.
Redefinition 003	*The effective set of triggers for a redefining transition is the set of triggers owned by that transition plus the set of triggers owned by directly or indirectly redefined transitions.*	See 9.3.17.5.
Redefinition 004	*If a redefining transition does not declare an effect behavior, then if a directly or indirectly redefined transition declares an effect, this effect is used for the redefining transition.*	See 9.3.17.3.
Redefinition 005	*If a redefining transition declares an effect behavior, then this behavior is used (i.e., it replaces any effect specified by the transition it redefined).*	See 9.3.17.5 and 9.3.17.6.

ID	Description	Test(s)
Redefinition 006	*A state can have entry, doActivity and exit behaviors. If the redefining state does not declare such a behavior, then the behavior from the directly or indirectly redefined state is used for the redefining state.*	See 9.3.17.6.
Redefinition 007	*A state can have entry, doActivity and exit behaviors. If the redefining state has such a behavior defined, then this behavior is used instead of the corresponding behavior of the redefined state.*	See 9.3.17.6.
Redefinition 008	*The set of deferrable triggers of a redefining state is the set of deferrable triggers declared by that state plus the set of deferrable triggers declared in its directly or indirectly redefined states.*	See 9.3.17.7.

9.4.20 Data Passing

A dispatched event occurrence can be either a SignalEvent occurrence or a CallEvent occurrence. Both kinds of Event occurrences can hold data. UML does not specify whether this data can be accessed during the execution of an RTC step by the Behaviors of a StateMachine (`entry`, `doActivity`, `exit`, `effect`) or by `guards` on Transitions. PSSM clarifies this point by describing how data embedded in event occurrences is passed to behaviors and guards (see 7.6.2 and 8.5.10).

The adopted approach for event data passing essentially places additional requirements on a conforming PSSM execution tool. These requirements are listed below, in order to show how the traceability of certain tests to the required capabilities for data passing. The following terms are used in the statement of the requirements:

- A *signature* is the ordered sequence of Parameters owned by an Operation or Behavior.
- One signature *conforms* to another if the first signature has the same number of Parameters as the second signature, and each Parameter of the first signature has a `type` that conforms to the `type` of the corresponding Parameter (in order) from the second signature, a multiplicity that is a superset of that of the second Parameter and the same ordering and uniqueness as the second Parameter. (Note: An "empty type" is considered to conform to any other type, including "empty", while no non-empty type conforms to an "empty type".)
- A signature *input-conforms* to another if the first signature conforms to the signature that results from including only the `in` Parameters from the second signature.

ID	Description	Test(s)
DataPassing 001	*If the trigger is for a SignalEvent, then all executed Behaviors must have either one Parameter or no Parameters. If a Behavior has a Parameter, then the Signal instance corresponding to the SignalEvent occurrence is passed into the Behavior execution as the value of its Parameter.*	See 9.3.4.11 and 9.3.10.7.
DataPassing 002	*If the trigger is for a CallEvent, then all executed Behaviors must have either no Parameters or they must have signatures that conform or input-conform to the signature of the Operation being called. If a Behavior has Parameters, then the values of the input Parameters of the call are passed into the Behavior execution as the values of the corresponding input Parameters of that Behavior.*	See 9.3.4.13, 9.3.4.14 and 9.3.4.15.

ID	Description	Test(s)
DataPassing 003	*If the trigger is for a CallEvent, then all executed Behaviors must have either no Parameters or they must have signatures that conform or input-conform to the signature of the Operation being called. If an effect, entry or exit Behavior is not just input-conforming, then the values of its output Parameters are passed out of its Behavior execution on its completion as values for the output Parameters of the called Operation.*	See 9.3.4.16.
DataPassing 004	*If the trigger is for a CallEvent, then all executed Behaviors must have either no Parameters or must have signatures that conform or input-conform to the signature of the Operation being called. The (synchronous) call ends at the end of the triggered RTC step. If the called Operation has output Parameters, then the values returned for those parameters are those produced by the last effect, entry or exit Behavior to complete its execution during the RTC step. (Since some or all of those Behaviors may execute concurrently, which one completes "last" may be only partially determined by the specified semantics. The values returned may legally be those produced by any Behavior that produces potential output values and is the last one to complete in any valid execution trace for the RTC step.)*	See 9.3.4.17.

Annex A Protocol State Machines
(informative)

A.1 Overview

ProtocolStateMachines are intended to specify some constraints on sequences of interactions supported by an associated classifier behavior together with their expected outcomes.

According to the UML specification [UML], violation of a constraint specified by a ProtocolStateMachine at run time shall result in an exception to be raised. However, since the fUML version upon which this specification is built [fUML] does not support exceptions, it is not possible to define an executable semantics for ProtocolStateMachines. Instead, this annex provides a precise but non-normative interpretation of the UML semantics for ProtocolStateMachines.

This interpretation assumes the following restrictions:

- ProtocolConformance is excluded since the real conformance of one protocol to another depends the valid interaction sequences actually allowed be each of them and cannot simply be claimed.
- Protocols specify contracts constraining all the involved entities. ProtocolsStateMachines are given semantics only in the case where they control binary interactions. This specification constrains them to be associated with an Interface
- There can be more than one protocol defined for given Classifier. The precise semantics specified below assumes that only one protocol is controlling a given interaction. ProtocolsStateMachines are constrained to be associated with a Port, which identifies an interaction point where the protocol applies.
- Neither Operation::precondition nor Operation::postcondition are derived. Therefore, it is not possible to compute them according to the preconditions and the postconditions of enabled ProtocolTransitions they are associated with. Instead, this specification assumes that the constraint implied by an enabled ProtocolTransition is the result of a logical "and" between the preconditions and the postconditions, respectively, of both the protocol transition and its associated operation.

A.2 Abstract Syntax

Figure A.1 shows classes related to protocol state machines in the StateMachines package from the UML abstract syntax. ProtocolConformance has been excluded from this subset, since it is a declarative statement that can be derived from the actual definition of the involved ProtocolStateMachine.

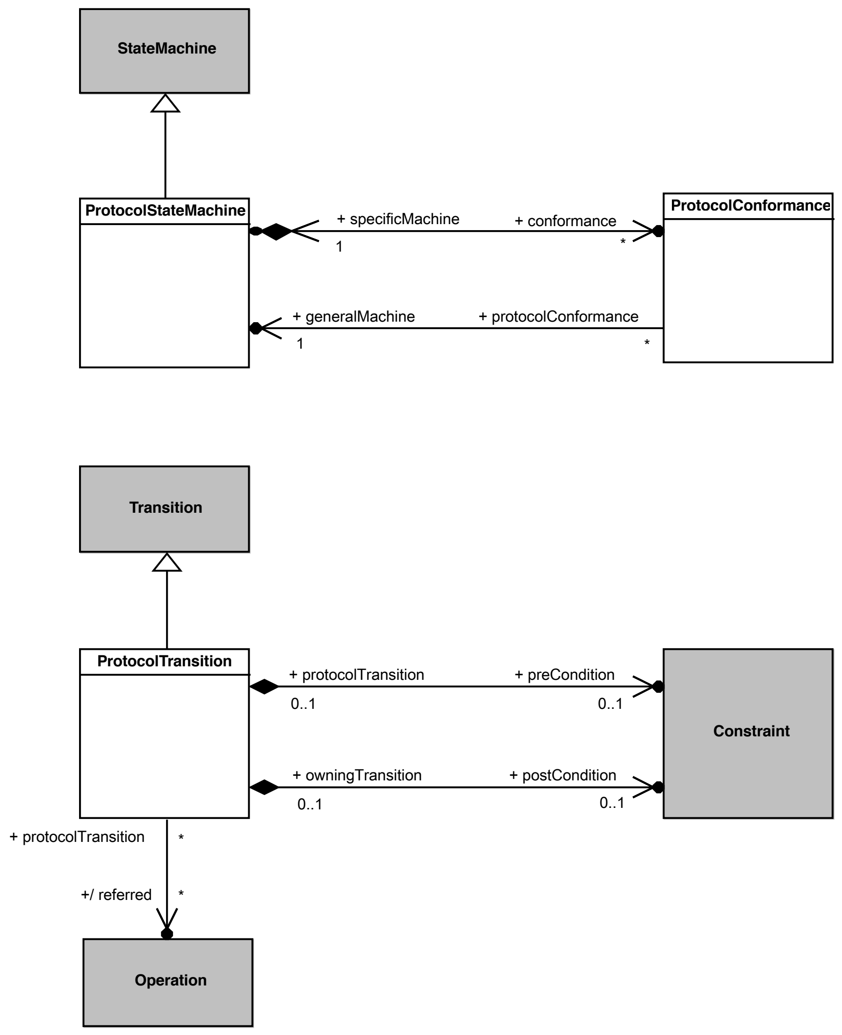


Figure A.1 - ProtocolStateMachines

A.3 Semantics

A.3.1 Controlled Events

Interactions controlled by a ProtocolStateMachine are restricted to event occurrences for which this state machine has at least one trigger defined. An occurrence of such a controlled event violates the protocol specified by a ProtocolStateMachine if it is not explicitly allowed according to the current state of the protocol.

A.3.2 Protocol States Configuration

The initial state configuration of the protocol is defined according the initial Pseudostate of each active Region within the ProtocolStateMachine. For each occurrence of an event controlled by the ProtocolStateMachine which is not invalid, the corresponding ProtocolTransition is fired, which result in the target State to become the active protocol state.

A.3.3 Protocol Violation

A protocol violation shall result in an exception being raised. This occurs in the following cases:

- An occurrence of a controlled event is received while it is invalid. That is, there is no enabled ProtocolTransition for that event for which the precondition is satisfied. In cases where that event is a CallEvent, this precondition is computed as a logical “and” between the ProtocolTransition::precondition and the CallEvent::operation::precondition.
- The postcondition of the ProtocolTransition activated following the occurrence of a controlled event linked o the invocation of a BehavioralFeature is not met when the execution of the corresponding method ends. In cases where that event is a CallEvent, this postcondition is computed as a logical “and” between the ProtocolTransition:: postcondition and the CallEvent::operation::postcondition.

Annex B State Machines for Passive Classes
(Informative)

B.1 Background and Rationale

The precise execution semantics for StateMachines in the main body of this specification covers the cases in which a StateMachine is either used as the classifierBehavior of an active Class or executes itself as a "standalone" active Behavior. However, StateMachines have also been used to specify the behavior of *passive* Classes, and support for this can be found in existing UML tools. This annex discusses the semantics of this usage, which are different than the semantics of StateMachines used with active Classes.

To help understand how the behavior of passive classes can be described using StateMachines, it is useful to recall that an essential characteristic of StateMachine behavior is that a response to a particular stimulus (e.g., a CallEvent occurrence) depends on the object's history; that is, the nature and order of preceding stimuli received by that object. In StateMachines for active Classes, this information is captured concisely by the current State of an object's classifierBehavior. However, when dealing with passive objects, which do not have a classifierBehavior, this means that, in the general case, each method of the Class of the object needs to include a conditional branch to handle the different responses based on some internal value that, in effect, represents the history of the object.

Consider, for example, the simplest case of a Stack Class shown in Figure B.1. Note that the response to a "pop" Operation will depend on whether the stack is empty or not. Similarly, assuming that the stack is of limited capacity, the response to a "push" Operation will differ when the stack is full compared to when it is not full.

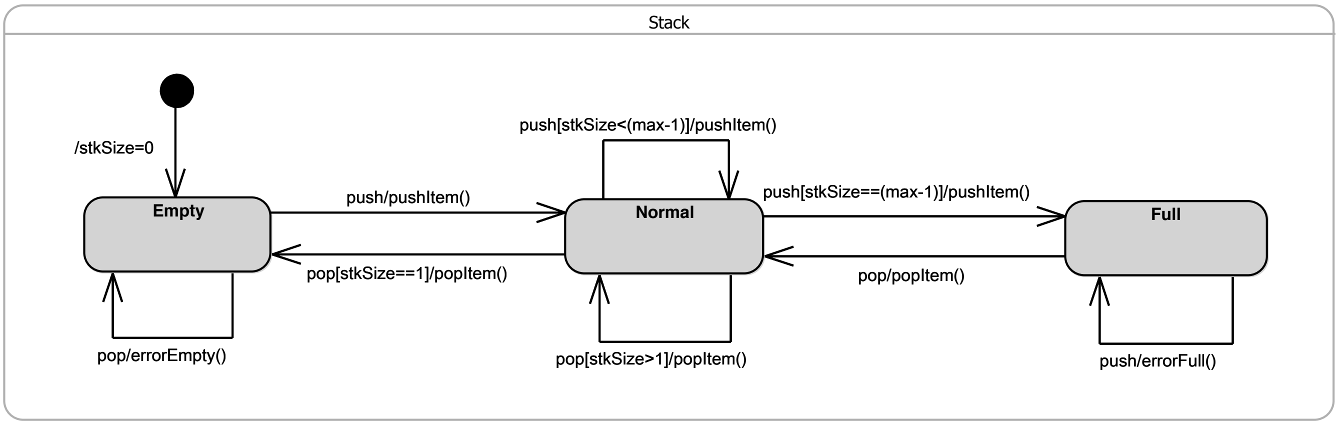


Figure B.1 - Stack Example

Of course, this can be coded explicitly by defining a suitable local variable of the object (e.g., "stack size") and using appropriate action language conditional statements. However, this not only obscures the true nature of the behavior in question, but, because it relies on relatively low-level (i.e., "manual") coding, it is also more error prone and requires more effort by the modeler. This approach becomes increasingly more problematic as the complexity of the behavior grows.

Hence, the motivations behind supporting StateMachine specifications of passive Class behaviors are to reduce the burden on the modeler, to more clearly describe an object's behavior using a higher-level formalisms, and to increase both reliability and productivity.

B.2 Semantics

To avoid gratuitous differences from the familiar semantics of active StateMachines, the general strategy taken here is to be fully consistent with those semantics wherever possible. Note that this approach covers both passive Classes as well as stand-alone *passive* StateMachines (which are, after all, Classes as well).

The core idea behind the approach is straightforward: map the StateMachine specification into an equivalent set of behavioral fragments and conditional statements distributed across the appropriate methods. For example, all three transitions triggered by the “pop” CallEvent in the Stack example above, would be mapped to appropriate conditional statement cases of a single “pop” Operation method. The control variable of such a statement would correspond to the current state of the StateMachine[6]. This is illustrated by the following pseudocode for the method of the “pop” Operation[7]:

```
operation pop(): Item {
   case (state) {
        'Empty':  errorEmpty();
        'Normal':      if (stkSize == 1) then
                            {popItem();
                             nextState('Empty');}
                       else
                            popItem();
        'Full':        {popItem();
                        nextState('Normal');}
        };
   }
```

Furthermore, any action associated with the initial Transition would be mapped to the method of the Class constructor.

Of course, in addition to the lack of a classifierBehavior, one key difference between active and passive Class semantics is in how the Transition triggering mechanism works. For active Classes, triggering is realized by a dedicated scheduling and dispatch mechanism, which is external to the StateMachine instance. Among other responsibilities, this mechanism also ensures that run-to-completion semantics are enforced. In contrast, no such mechanism exists for passive Classes; the methods of a passive Class are executed synchronously when some calling behavior invokes the corresponding Operation. Consequently, if two or more concurrently executing behaviors make overlapping calls to the same passive object, there is a possibility of concurrency conflicts that would violate the run-to-completion semantics. (Note that this can occur even if all of the Operations of the passive class are declared as “guarded”, since that only prevents a given Operation being invoked concurrently. However, it would still be possible to concurrently invoke two or more different Operations of the passive class.)

Therefore, to ensure run-to-completion semantics of a passive-Class StateMachine, it is necessary that, for any passive Class whose behavior is defined by a StateMachine, at most one Operation call can be executed (to completion) at a time. This restriction avoids unsafe and error prone designs, and it is consistent the core semantic tenets of UML StateMachines.

6 The exact type and format of such a variable are of no concern here; implementers are free to chose their own.

7 To simplify the example, we assume here that there are no entry, exit, or doActivity behaviors associated with any of the states.