Date: February 2016

Precise Semantics of UML State Machines (PSSM)

Initial Submission

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

OMG Document Number: ad/2016-02-xx

Machine Readable File(s):

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

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

Table of Contents

0 Submission Introduction....1
- 0.1 Overview....1
- 0.2 Submitters....1
- 0.3 Mandatory Requirements....1
- 0.4 Non-Mandatory Features....1
- 0.5 Issues To Be Discussed....1

1 Scope....2

2 Conformance....2

3 Normative References....2

4 Terms and Definitions....2

5 Symbols....2

6 Additional Information....3
- 6.1 Changes to Adopted OMG Specifications....3
- 6.2 Acknowledgments....3

7 Abstract Syntax....4
- 7.1 Overview....4
- 7.2 Values....4
  - 7.2.1 Overview....4
  - 7.2.2 Expressions....4
    - 7.2.2.1 Overview....4
    - 7.2.2.2 Class Descriptions....4
- 7.3 Structured Classifiers....4
  - 7.3.1 Overview....4
  - 7.3.2 Classes....4
- 7.4 State Machines....4
  - 7.4.1 Overview....4
  - 7.4.2 Behavior State Machines....4
  - 7.4.3 State Machine Redefinition....4

8 Execution Model......5
8.1 Overview......5
8.2 Loci......5
8.2.1 Overview......5
8.2.2 Class Descriptions......5
8.3 Values......5
8.3.1 Overview......5
8.3.2 Expressions......5
8.3.2.1 Overview......5
8.3.2.2 Class Descriptions......5
8.4 Structured Classifiers......5
8.4.1 Overview......5
8.4.2 Classes......5
8.5 State Machines......5
8.5.1 Overview......5
8.5.2 Behavior State Machines......5
8.5.3 State Machine Redefinition......5
9 Test Suite......6
9.1 Overview......6
9.1.1 UML Semantics Requirements......6
9.1.2 Test Suite Organization......6
9.1.3 Test Coverage and Limits......6
9.1.4 Constraint on Vendors implementing the Semantic Model......6
9.2 Utilities......7
9.2.1 Overview......7
9.2.2 Architecture......7
9.2.2.1 Relation with UML testing profile......7
9.2.2.2 Base Semantic Test......8
9.2.2.2.1 Tester......8
9.2.2.2.2 Target......9
9.2.2.2.3 SemanticTest......10
9.2.2.3 Semantic Test and Test Suite......11
9.2.2.3.1 SemanticTestSuite......11
9.2.3 Protocol......14
9.2.4 Tracing......14
9.3 Tests......14
9.3.1 Behaviors......14
9.3.1.1 Overview......14
9.3.1.2 Test Behavior 001......14
9.3.1.3 Test Behavior 002......14
9.3.1.4 Test Behavior 004......14

9.3.2 Transitions.......... 14
9.3.3 Region.......... 14
9.3.4 Event.......... 14
9.3.5 Entering.......... 14
9.3.6 Exiting.......... 14
9.3.7 Exit.......... 14
9.3.8 Entry.......... 14
9.3.9 Choice.......... 14
9.3.10 Other Tests.......... 14
9.4 Test Coverage and Limits.......... 14
A Protocol State Machines.......... 15
A.1 Overview.......... 15
A.2 Abstract Syntax.......... 15
A.3 Execution Model.......... 15

0 Submission Introduction

0.1 Overview

0.2 Submitters

0.3 Mandatory Requirements

0.4 Non-Mandatory Features

0.5 Issues To Be Discussed

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

1 Scope

2 Conformance

3 Normative References

4 Terms and Definitions

5 Symbols

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

6 Additional Information

6.1 Changes to Adopted OMG Specifications

6.2 Acknowledgments

7 Abstract Syntax

7.1 Overview

7.2 Values

7.2.1 Overview

7.2.2 Expressions

7.2.2.1 Overview

7.2.2.2 Class Descriptions

7.3 Structured Classifiers

7.3.1 Overview

7.3.2 Classes

7.4 State Machines

7.4.1 Overview

7.4.2 Behavior State Machines

7.4.3 State Machine Redefinition

8 Execution Model

8.1 Overview

8.2 Loci

8.2.1 Overview

8.2.2 Class Descriptions

8.3 Values

8.3.1 Overview

8.3.2 Expressions

8.3.2.1 Overview

8.3.2.2 Class Descriptions

8.4 Structured Classifiers

8.4.1 Overview

8.4.2 Classes

8.5 State Machines

8.5.1 Overview

8.5.2 Behavior State Machines

8.5.3 State Machine Redefinition

9 Test Suite

9.1 Overview

PSSM specification is delivered with a test suite. This test suite is designed as a UML model. It captures a set of tests cases that when executed enables us to assess the semantics that is captured within the semantic model for UML state machines that is presented in section [XX].

9.1.1 UML Semantics Requirements

The definition of the test suite is strongly coupled to the semantics requirements that were identified within the UML chapter related to state machines semantics (see annex XX). Indeed each test case verify (or not) one particular requirement. A requirement takes the form of textual description of particular part of the semantics related to state machines. A test case is defined as an executable UML model that assesses this particular part of the semantics. Each time a test will be presented in section [XX its related requirement will also be reminded.

9.1.2 Test Suite Organization

The test suite is separated in two parts. The first one 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 section [XX].

The second part of the test suite defines a set of packages where each package refers to a particular test category. As an example a test category in the test suite captures all tests cases related to transition semantics. Each test case in this category assert a specific part (identified in the requirements) of the transition semantics.

9.1.3 Test Coverage and Limits

The purpose of having a strong coupling between the semantics requirements for state machines is to be able (as in usual software development) to identify quickly and precisely what is covered by the semantic model in terms of semantics and what is currently not covered. Statistics and limitations about current coverage of UML state machines semantics are presented in section [REF].

9.1.4 Constraint on Vendors implementing the Semantic Model

Any tool implementing implementing the semantic model must be able to execute and pass all tests of the test suite to claim that it conforms to what is in this specification. As an additional requirement, it must also be able to execute and pass all tests provided in the PSCS (Precise Semantics for Composite Structures) test suite. This constraint is implied by the fact that PSSM is built on top PSCS but does not modify composite structure semantics.

9.2 Utilities

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 section [XX]. The communications that take place between the different elements of the architecture are presented in section [XX]. Finally section [XX] explains the process of generating a trace that capture 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.

9.2.1 Architecture~~Overview~~

This section presents the architecture that was defined to describe test cases to assess PSSM semantic model. In section 9.2.1.1 concepts provided by UTP (UML Testing Profile) which are reused in the context of the test suite architecture are identified. In section 9.2.1.2 the structure and the behavior of a base semantic test are presented. Finally section 9.2.1.3 identifies the structure of the semantic test container as well its behavior.

9.2.1.1 Relation with UML testing profile

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”.

The base architecture of the PSSM test suite is inspired by concepts identified by the UML testing profile. These concepts are:

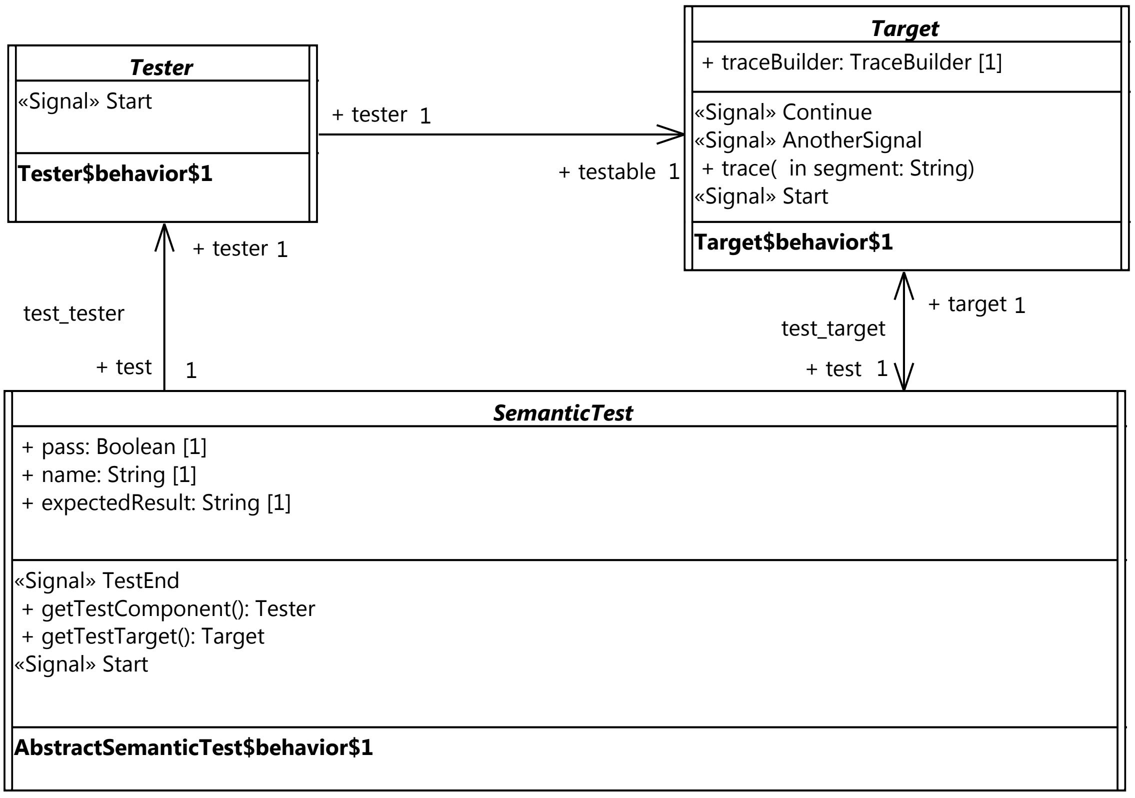
- **TestComponent:** “Test components are part of the test environment and are used to communicate with the system under test (SUT) and other test components. The main function of test components is to drive a test case by stimulating the system under test through its provided interfaces and to evaluate whether the actual responses of the system under test comply with the expected ones.” (see section 8.2.2.2 of UTP specification 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 section 9.2.2.4 of UTP specification 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 section 8.2.2.3 of UTP specification to read the complete description of this concept).

The next section shows how these concepts are used in the context of the definition of the abstract architecture of a test.

9.2.1.2 Base Semantic Test



The figure [XX] illustrates the architecture of an abstract semantic test.

9.2.1.2.1 Tester

9.2.1.2.1.1 Description

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

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

9.2.1.2.1.2 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.

9.2.1.2.1.3 Receptions

- Start – A tester can receive Start signal

9.2.1.2.1.4 Classifier Behavior

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

9.2.1.2.2 Target

9.2.1.2.2.1 Description

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

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

All along 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.

9.2.1.2.2.2 Attributes

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

9.2.1.2.2.3 Association Ends

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

9.2.1.2.2.4 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.

9.2.1.2.2.5 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.

9.2.1.2.2.6 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.

9.2.1.2.3 SemanticTest

9.2.1.2.3.1 *Descriptions*

A SemanticTest is the main artifact of a semantic test case. It is in charge of instantiating and controlling the tester and the target (i.e. the SUT). When the execution of the SUT is done then the execution trace that was produced is provided to the semantic test case for analysis. If the trace matches one of the expected trace for the executed state machine the test pass otherwise it fails.

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

9.2.1.2.3.2 *Attributes*

- name: String [1] – The name of the test case name.
- pass: Boolean [1] – The current status (pass or fail) of the test.
- expectedResult: String [1] – The execution trace that is expected for SUT.

9.2.1.2.3.3 *Operations*

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

9.2.1.2.3.4 *Receptions*

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

9.2.1.2.3.5 *Classifier Behavior*

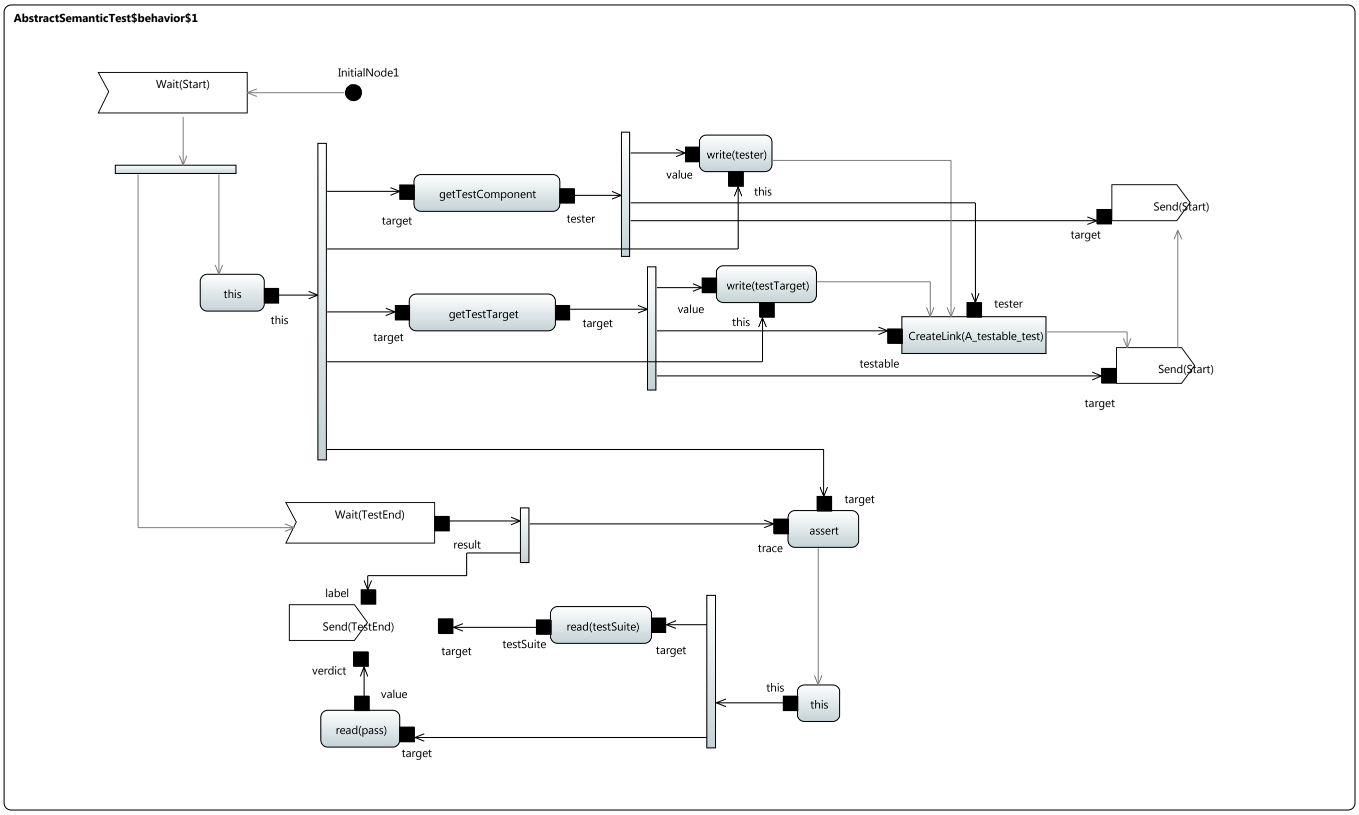
The classifier behavior of a semantic test is defined as UML activity (which conforms to the fUML subset). It is presented in figure [XX].

The principle of the behavior is the following:

1. When the classifier behavior starts it blocks waiting for the dispatching of a Start signal
2. On the reception of the Start signal it creates and instantiates both a the tester and the target.
    - The links (instances of associations) are created.
    - The tester and the target receive a Start signal.

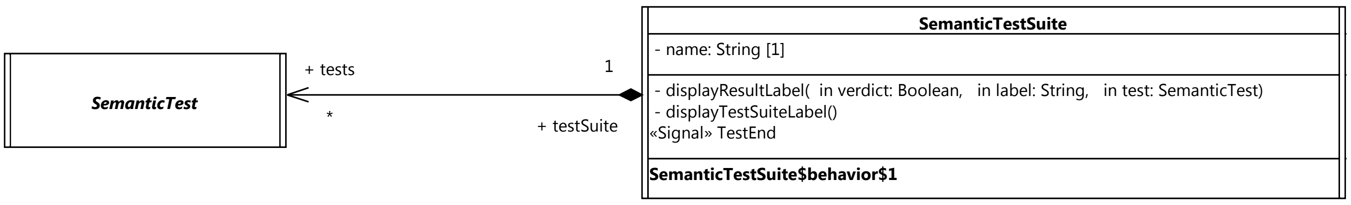
- The semantic test finally blocks again waiting for the dispatching of a TestEnd signal.

3. When the TestEnd signal is received then it computes the verdict of the test and notifies the test suite that controls it.



Note that specializations of a semantic test are not intended to override the dynamic that is captured in this classifier behavior.

9.2.1.3 Semantic Test and Test Suite



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

9.2.1.3.1 SemanticTestSuite

9.2.1.3.1.1 Description

A SemanticTestSuite owns a set of SemanticTest. 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 charge of displaying the results. This is active class and is not intended to be specialized.

One can notice that it matches the concept of TestContext proposed by UTP. This class has the stereotype TestContext applied.

9.2.1.3.1.2 Attributes

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

9.2.1.3.1.3 Association Ends

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

9.2.1.3.1.4 Receptions

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

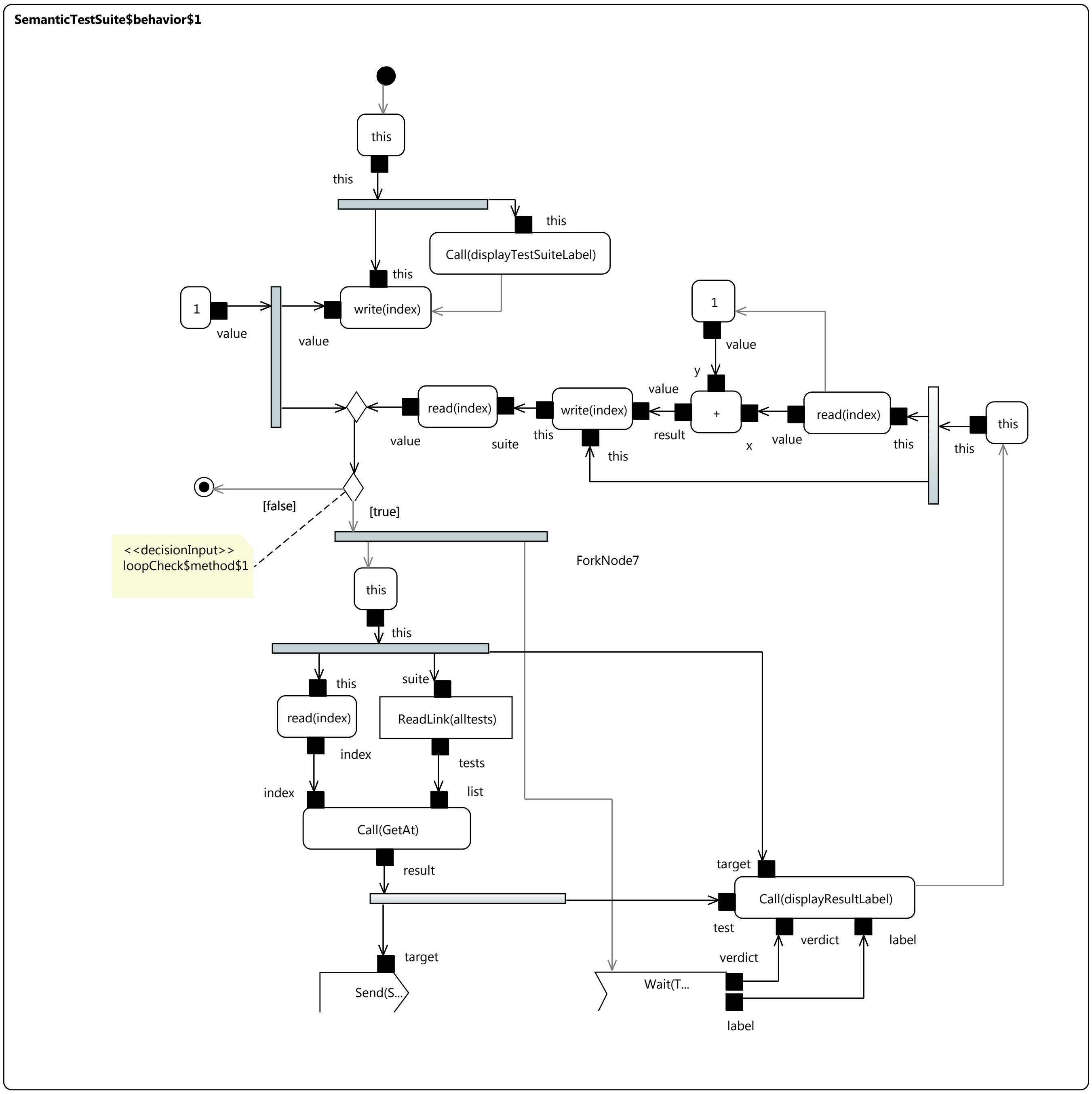
9.2.1.3.1.5 Operations

- displayResultLabel(in verdict: Boolean [1], in label: String[1], in test: SemanticTest [1]) – A convenience operation to display the test result on a output stream.

- displayTestSuiteLabel() - A convenience operation to display the name of the name of the suite on an output stream.

9.2.1.3.1.6 Classifier Behavior

The classifier behavior of the suite encodes a sequential execution of the test known by the test suite. Each time a test (that was started by the test suite) terminates then the test suite gets notified. As soon as the notification (i.e., a TestEnd signal) is dispatched then the test result is displayed in the output stream chosen by the user. The classifier behavior of the semantic test suite is presented in figure [XX].



9.2.2 Protocol

The Protocol package of the test suite in two packages: Messages and Events.

1. Messages contains all signals used to communicate between the different active classes (see the model below).

```
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 {}

        /* -- Stimulations --*/

}
```

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

The following sub-sections present the different signals and there usages in the context of the test suite.

9.2.2.1 Synchronization

These signals are used to synchronize executions of different active objects.

9.2.2.1.1 Start

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 Testert and the Target is that they must all register an accepter for the Start signal at the beginning of the execution of their classifier behaviors. Note that Start signal does not encapsulate data (since it has no attributes).

9.2.2.1.2 End

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.

9.2.2.1.2.1 Attributes

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

9.2.2.1.3 TestEnd

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

9.2.2.1.3.1 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.

9.2.2.2 Stimulation sequence

These signals (i.e.Continue, Pending and AnotherSignal) are used by the tester to stimulate the target (i.e. the system under test). None of these signals encapsulate data.

9.2.3 Tracing

At runtime 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.) which must be part of the trace are up to the designer of the test. To add a new information in the trace the designer must call the trace operation provided by the target. This latter will delegate to the trace builder. Such call to the trace operation must take place while the state machine is currently running. Consequently there 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
5.

9.3 Tests

9.3.1 Behaviors

9.3.1.1 Overview

9.3.1.2 Test Behavior 001

9.3.1.3 Test Behavior 002

9.3.1.4 Test Behavior 004

9.3.2 Transitions

9.3.3 Region

9.3.4 Event

9.3.5 Entering

9.3.6 Exiting

9.3.7 Exit

9.3.8 Entry

9.3.9 Choice

9.3.10 Other Tests

9.4 Test Coverage and Limits

Annex A Protocol State Machines
(informative)

A.1 Overview

A.2 Abstract Syntax

A.3 Execution Model