FROM RESEARCH TO INDUSTRY

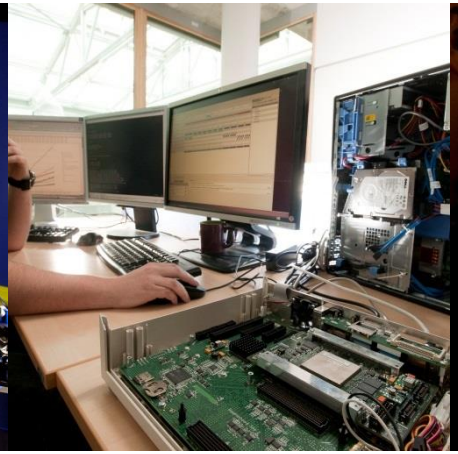
cea tech

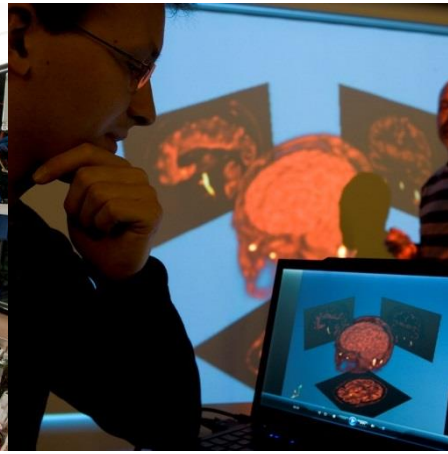
list

[PSSM] – SAN DIEGO MEETING DECEMBER 10TH

Jérémie TATIBOUET (CEA LIST)

Arnaud CUCCURU (CEA LIST)





INSTITUT CARNOT CEA LIST

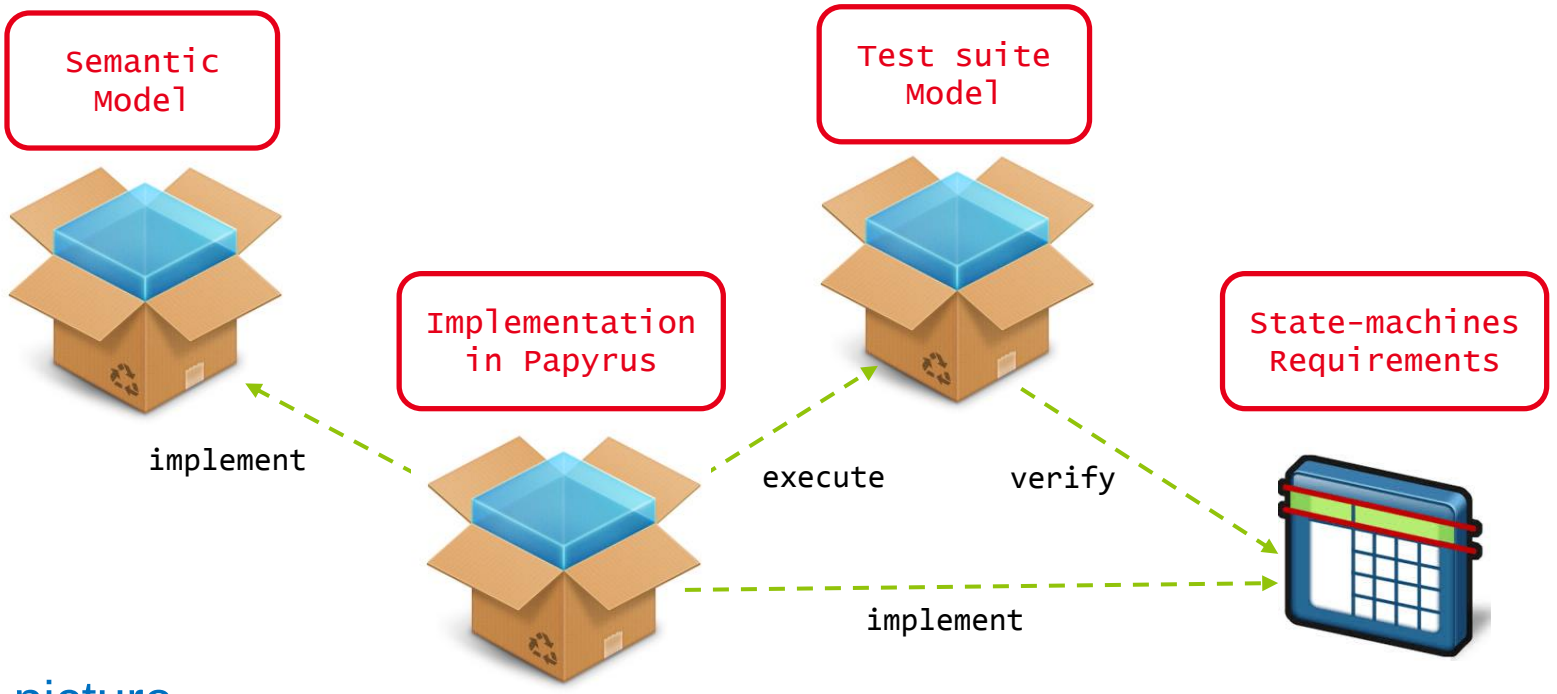
digiteo

OUTLINE

A. Semantic model
   - Model architecture
   - Core
     - Base semantic visitors
     - State-machine execution
     - State-machine configuration
   - State
     - Simple, composites and Final
   - Communication
     - Event accepter and completion events
   - Selection
     - Transition selection
   - Pseudo states
     - Entry point / Exit point / Choice

B. Test suite
   - Reminder on the structure
   - Requirements verification review

C. Assessment
   - Progress compare to last meeting
   - What is not supported

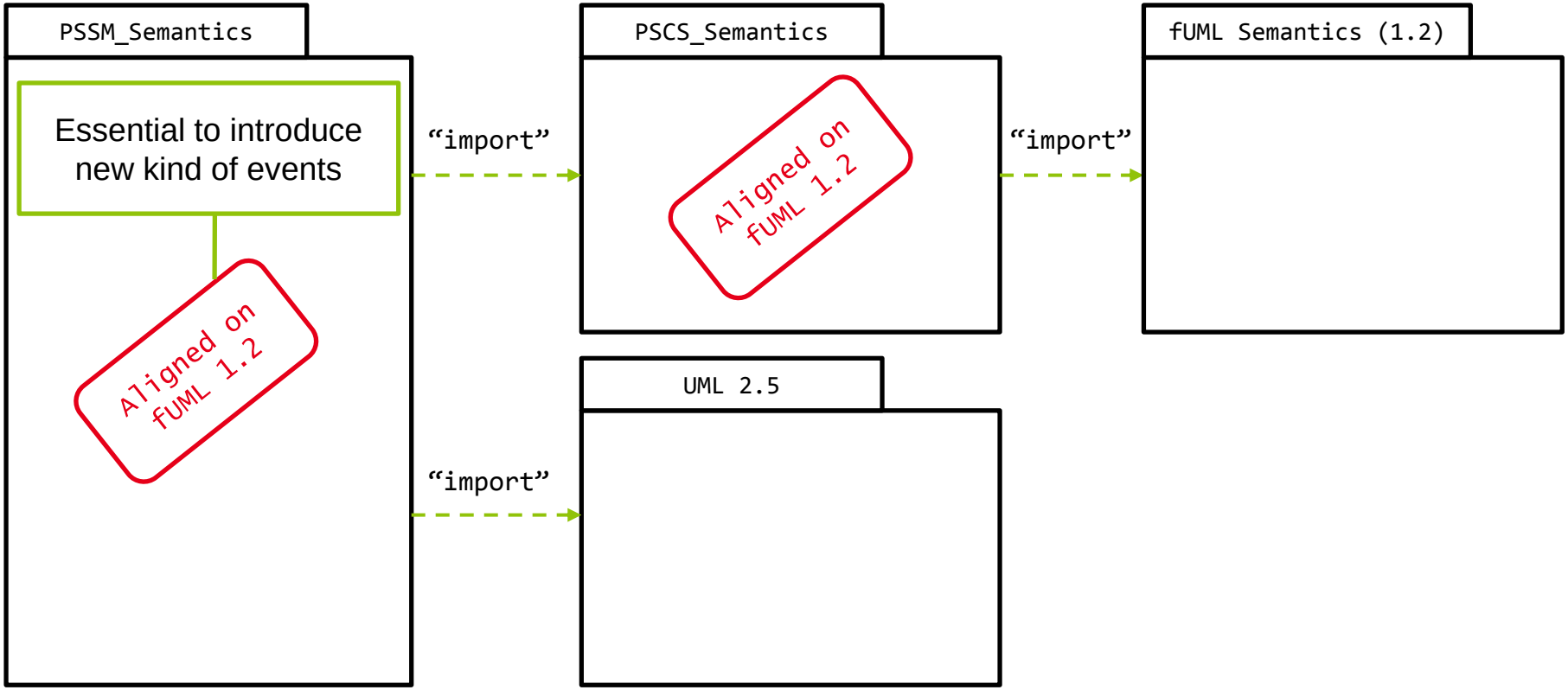


Big picture

- Prototype implementation
  - An implementation of the semantic model
  - Capture the requirements for UML state-machines
  - Capable of executing the test suite
- Test suite
  - Define a set of executable state-machines
  - Each test enables the verification of one requirement
  - The “verified” status is calculated based on trace comparison

SEMANTIC MODEL ARCHITECTURE

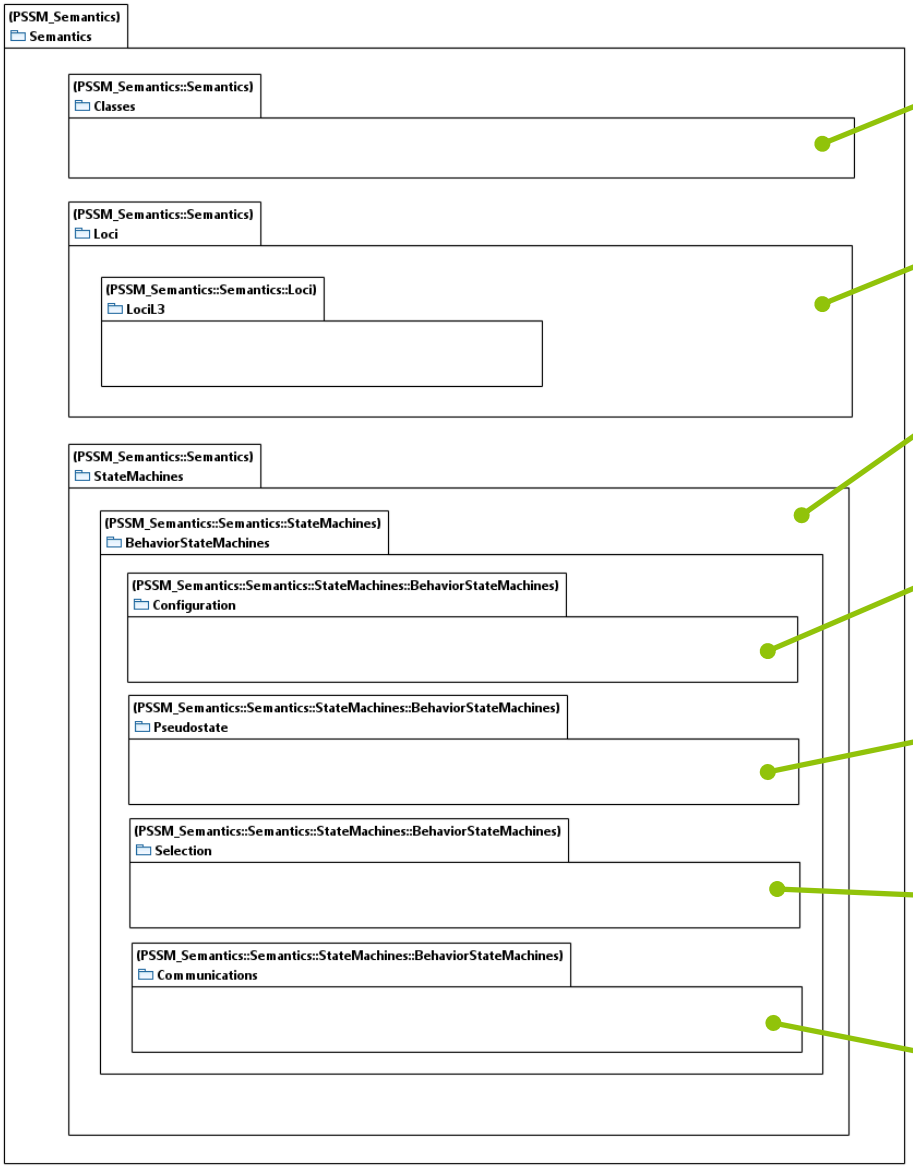
SEMANTIC MODEL DEPENDENCIES



Dependencies

- UML 2.5
- PSCS semantics
  - In Papyrus, Moka (model execution tool) is aligned on fUML 1.2. As PSCS is built on top of fUML it is aligned with fUML 1.2. **Note: fUML 1.2 has no impact on PSCS.**

SEMANTIC MODEL PACKAGES



Extended visitors related to structure

Extended elements related to the instantiation

Semantic visitors defined to capture UML state-machines components

Elements to build the state-machine configuration

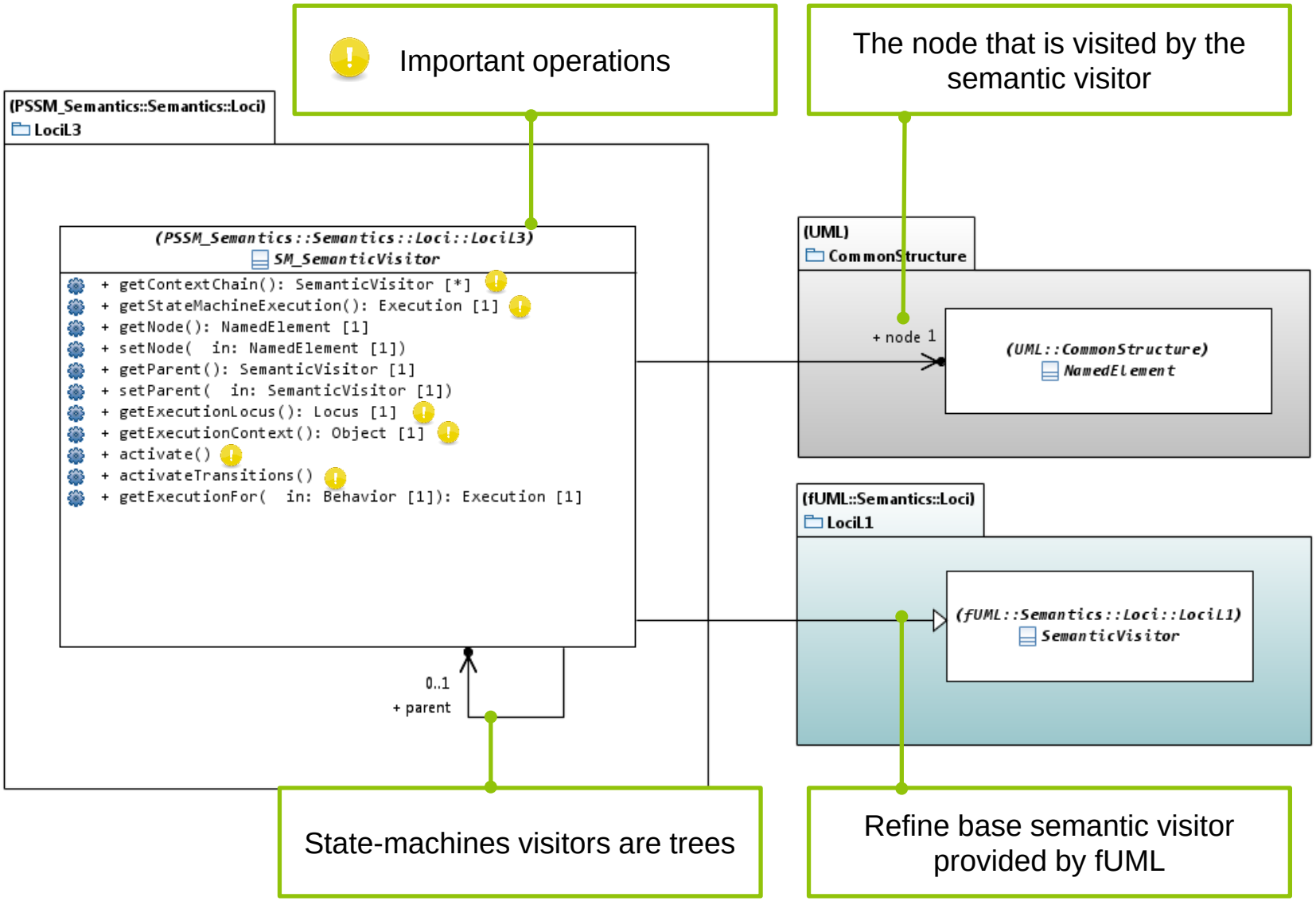
Visitors implementing pseudo state semantics

Elements implementing transition selection

Elements related to communications that can be received (accepter, new event occurrence)

CORE OF PSSM

BASE SEMANTIC VISITOR

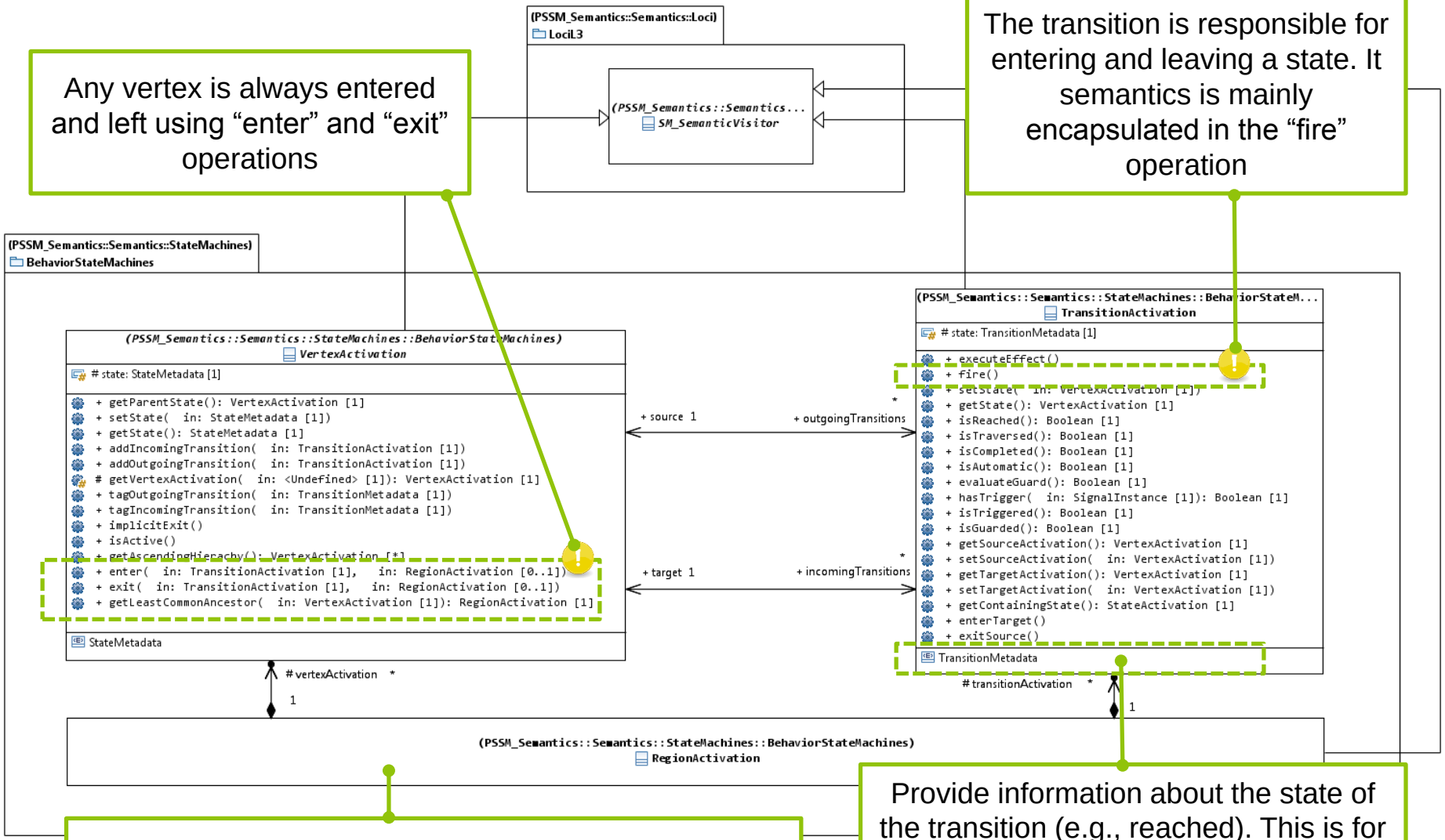


CORE VISITORS

STATE-MACHINE EXECUTION: FIRE(ENTER()-EXIT())

Any vertex is always entered and left using “enter” and “exit” operations

The transition is responsible for entering and leaving a state. It semantics is mainly encapsulated in the “fire” operation



Semantic visitor for a region. It references a set of visitors for its contained transition and vertices

Provide information about the state of the transition (e.g., reached). This is for example used in the context of the ExitPoint

FIRE OPERATION

EXCERPT FROM THE PROTOTYPE IMPLEMENTATION

Propagation of the execution

```
protected void enterTarget(){
    Transition node = (Transition) this.getNode();
    RegionActivation leastCommonAncestor = null;
    // A target state is always entered except when the transition reaching this latter
    // has the internal kind
    if(node.getKind()!=TransitionKind.INTERNAL_LITERAL){
        if(this.vertexSourceActivation.getParentState()!=this.vertexTargetActivation.getParentState()){
            leastCommonAncestor = this.vertexSourceActivation.getLeastCommonAncestor(this.vertexTargetActivation);
        }
        this.vertexTargetActivation.enter(this, leastCommonAncestor);
    }
}
```

1

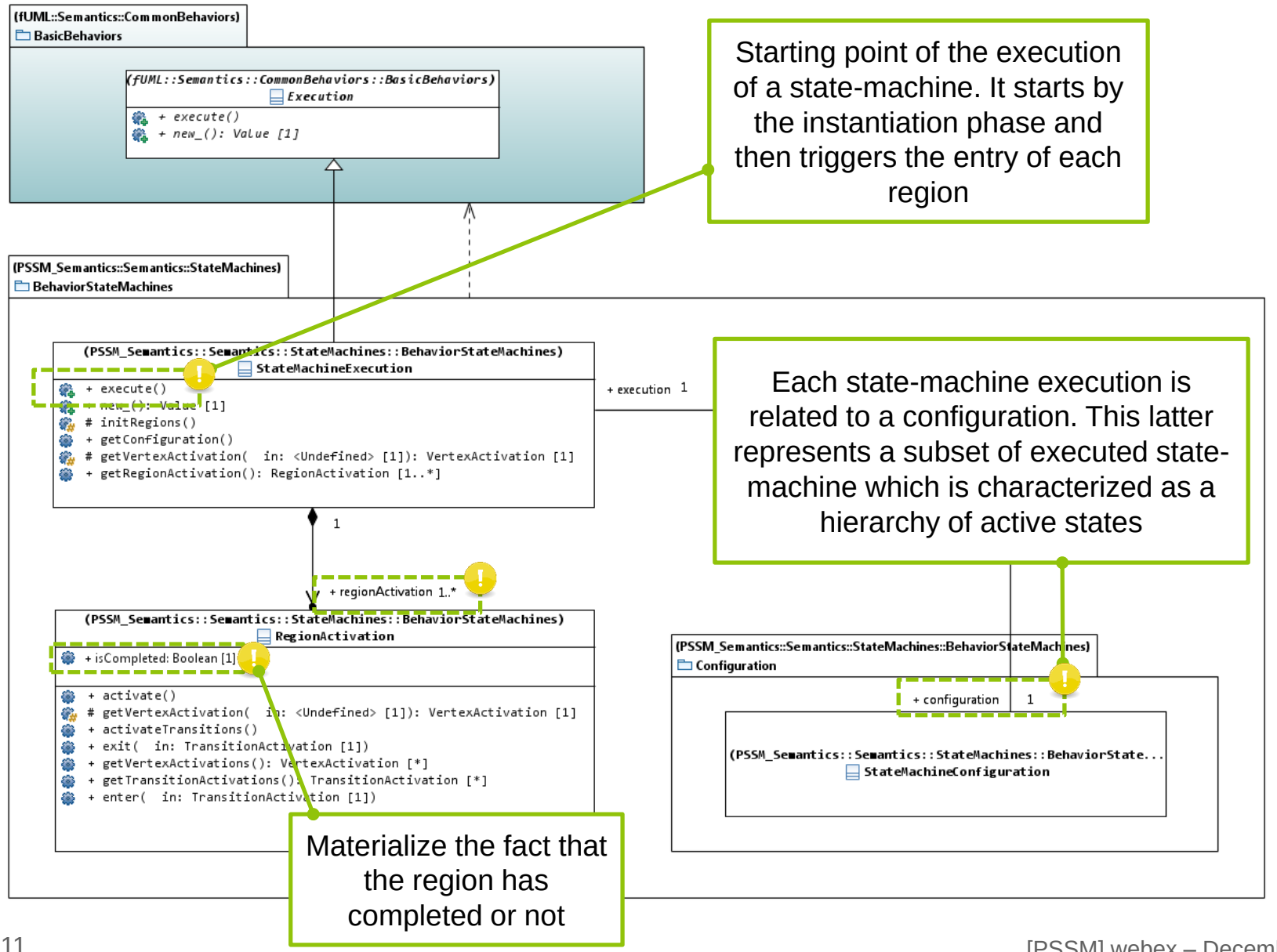
```
public void executeEffect(){
    // Execute the effect that is on the transition if it exists one
    Transition transition = (Transition) this.getNode();
    Execution execution = this.getExecutionFor(transition.getEffect());
    if(execution!=null){
        execution.execute();
    }
}
```

2

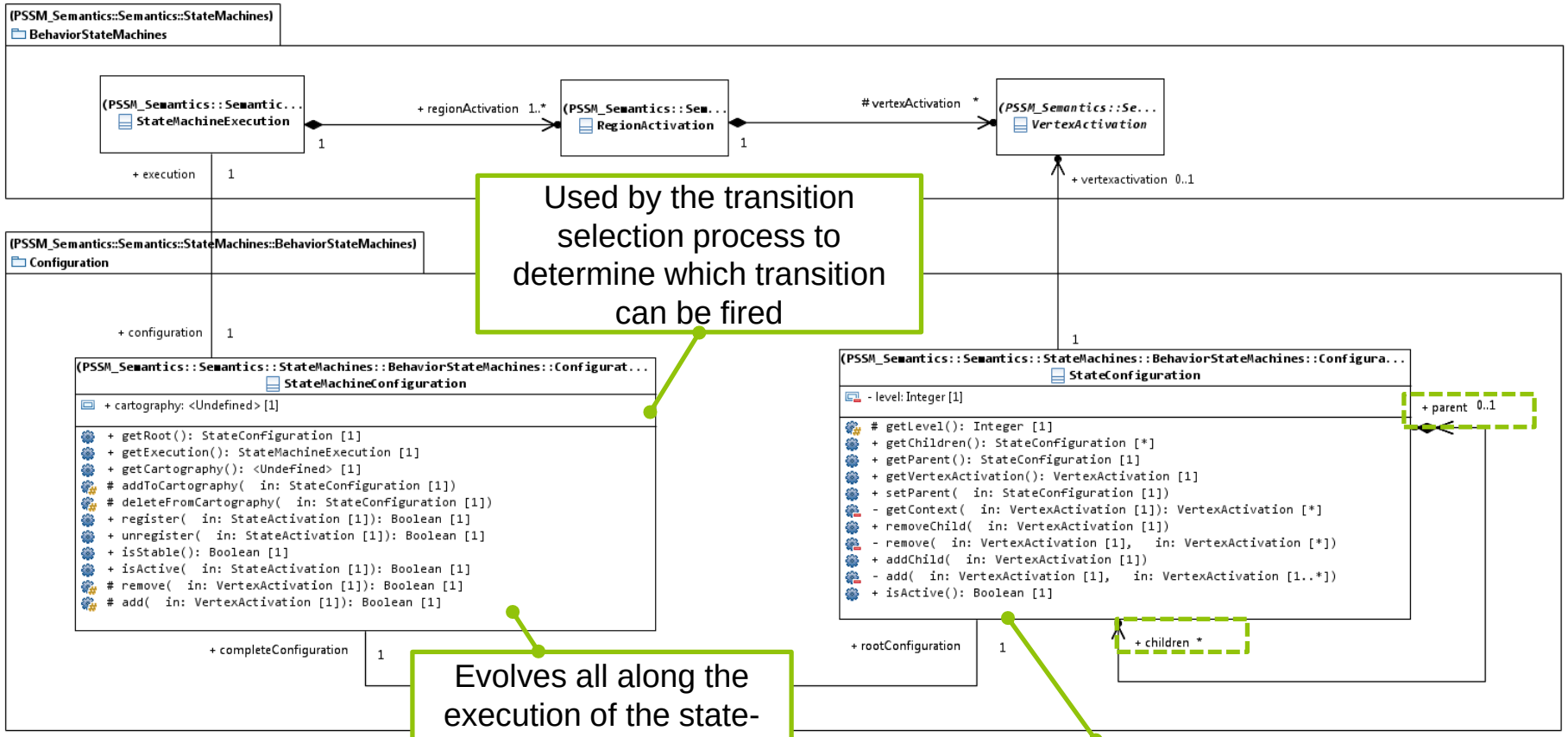
```
protected void exitSource(){
    // A source state is exited when the transition fires under the following conditions:
    // 1 - The transition leaving the source state is external
    // 2 - The transition leaving the source state is local but the source state is not the
    //     state which contains the transition
    Transition node = (Transition) this.getNode();
    boolean exitSourceState = false;
    RegionActivation leastCommonAncestor = null;
    if(node.getKind()==TransitionKind.EXTERNAL_LITERAL){
        exitSourceState = true;
    }else if(node.getKind()==TransitionKind.LOCAL_LITERAL){
        StateActivation stateActivation = this.getContainingState();
        exitSourceState = stateActivation!=null && node.getSource()!=stateActivation.getNode();
    }
    if(exitSourceState){
        if(this.vertexSourceActivation.getParentState()!=this.vertexTargetActivation.getParentState()){
            leastCommonAncestor = this.vertexSourceActivation.getLeastCommonAncestor(this.vertexTargetActivation);
        }
        this.vertexSourceActivation.exit(this, leastCommonAncestor);
    }
}
```

Propagation of the execution

MAIN CLASS FOR STATEMACHINE EXECUTION



MAINTAINING THE ACTIVE CONFIGURATION



Role

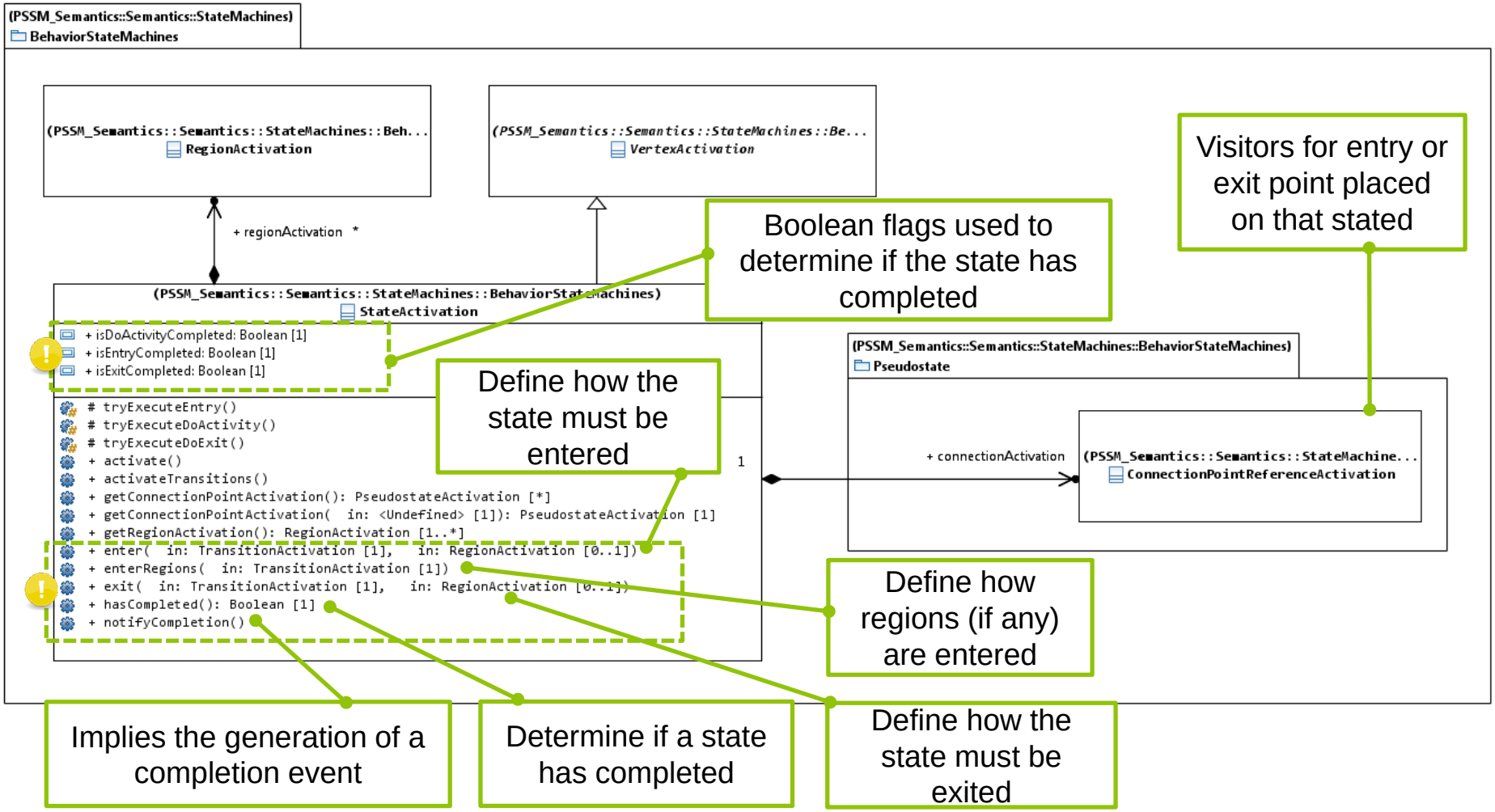
- View of active states in a stable situation

Principles

- Each state that is entered registers within the configuration
- Each state that is exited leaves the configuration

STATES: SIMPLE, COMPOSITE AND FINAL

SIMPLE AND COMPOSITES



Note

- Simple and composite state semantics is defined in a single visitor

ENTER(…)

EXCERPT FROM THE PROTOTYPE IMPLEMENTATION

If parent need to be entered first

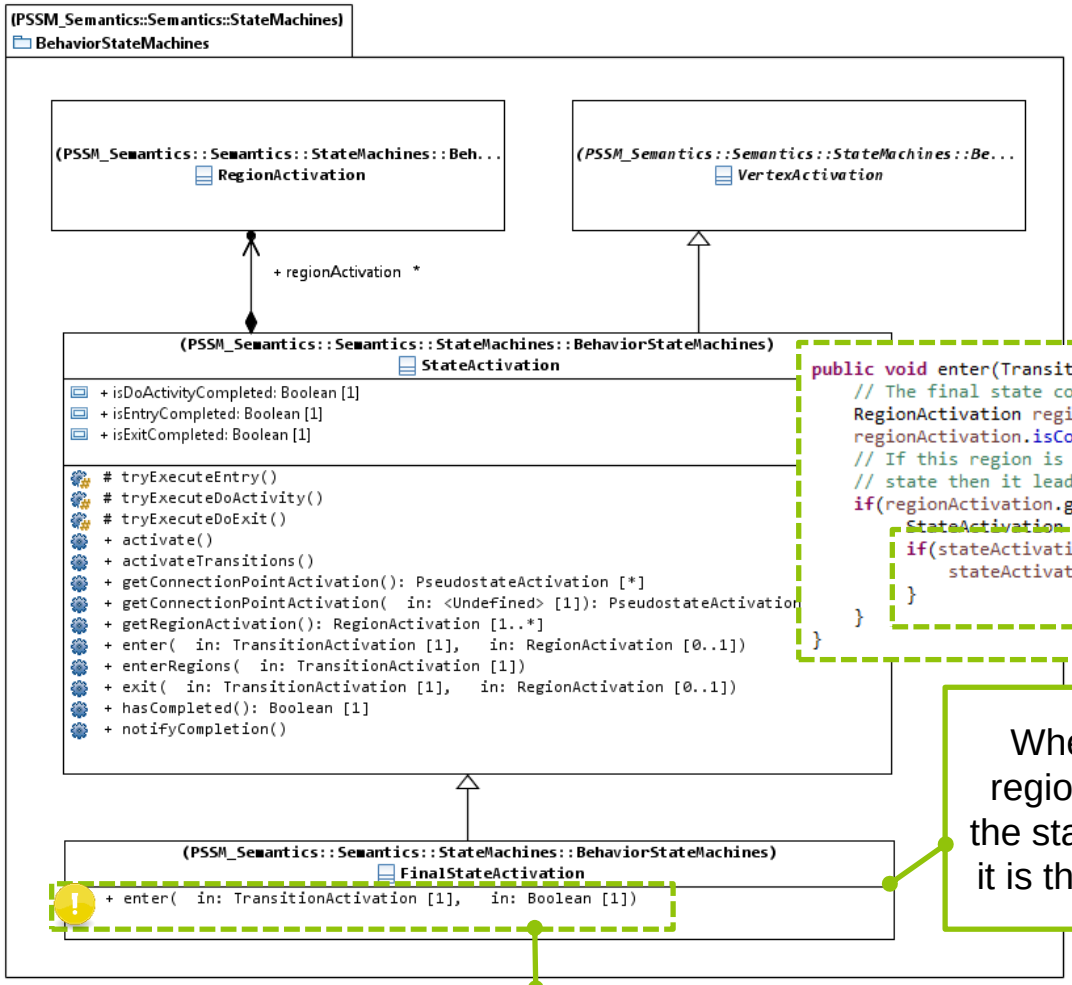
Calculate initial completion status

Complete if this is allowed

Execute entry, then enter region(s), finally execute doActivity

```
public void enter(TransitionActivation enteringTransition, RegionActivation leastCommonAncestor) {
    if(this.state.equals(StateMetadata.IDLE)){
        // The state is entered via an explicit transition
        // The impact on the execution is that the parent state
        // of the current state is not active then it must be entered
        // the rule applies recursively
        if(leastCommonAncestor!=null){
            RegionActivation parentRegionActivation = (RegionActivation) this.getParent();
            if(leastCommonAncestor!=parentRegionActivation){
                StateActivation stateActivation = (StateActivation) parentRegionActivation.getParent();
                if(stateActivation!=null){
                    stateActivation.enter(enteringTransition, leastCommonAncestor);
                }
            }
        }
        // Initialization
        State state = (State) this.getNode();
        super.enter(enteringTransition, leastCommonAncestor);
        this.isEntryCompleted = state.getEntry()==null;
        this.isDoActivityCompleted = state.getDoActivity()==null;
        this.isExitCompleted = state.getExit()==null;
        // When the state is entered it is registered in the current
        // state-machine configuration
        StateMachineExecution smExecution = (StateMachineExecution)this.getStateMachineExecution();
        smExecution.getConfiguration().register(this);
        // If state has completed then generate a completion event*/
        if(this.hasCompleted()){
            this.notifyCompletion();
        }else{
            // Execute the entry behavior if any
            this.tryExecuteEntry();
            // If the state is not completed, then try to start its owned regions.
            // A region is entered implicitly since the is not the
            this.enterRegions(enteringTransition);
            // Execute the doActtivity if any
            this.tryExecuteDoActivity();
        }
    }
}
```

FINAL STATE



The owner generates a completion event

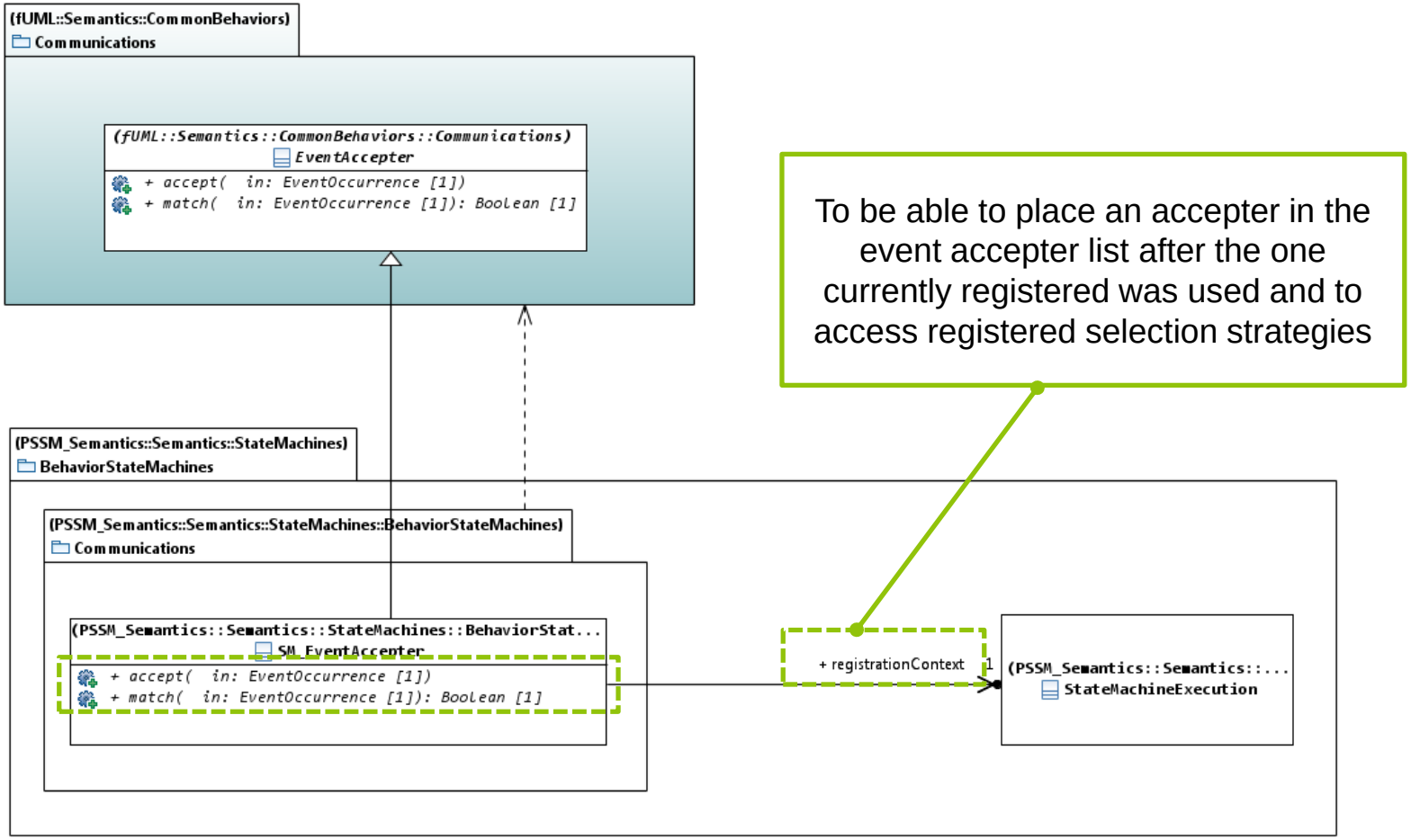
```
public void enter(TransitionActivation enteringTransition, RegionActivation leastCommonAncestor) {
    // The final state completes the region in which it is located*/
    RegionActivation regionActivation = (RegionActivation) this.getParent();
    regionActivation.isCompleted = true;
    // If this region is the last of this state to complete through its final
    // state then it leads to the generation of a completion event
    if(regionActivation.getParent() instanceof StateActivation){
        StateActivation stateActivation = (StateActivation) regionActivation.getParent();
        if(stateActivation.hasCompleted()){
            stateActivation.notifyCompletion();
        }
    }
}
```

When the final state is reached then its owning region terminates. If the region is terminated then the state owning that region may have completed. If it is the case then a completion event is generated.

The visitor for the FinalState simply extends the one defined for regular (i.e., simple and composite) state. It redefines the way the final state is entered.

COMMUNICATIONS: ACCEPTER AND COMPLETION EVENTS

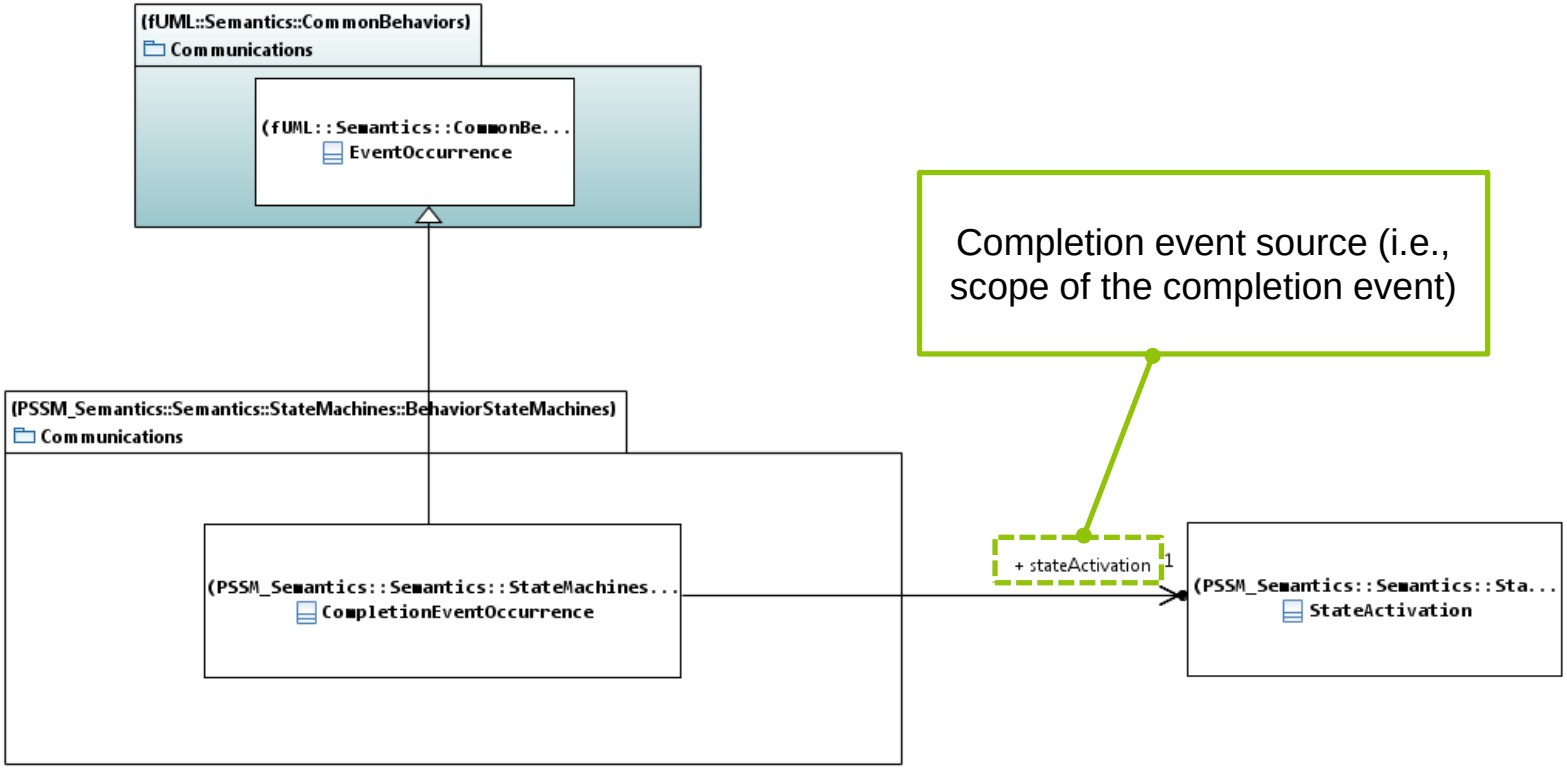
EVENT ACCEPTER



Principle

- A single accepter is registered for the executed state-machine
- The match delegates to the transition selection process

COMPLETION EVENT (1/2)

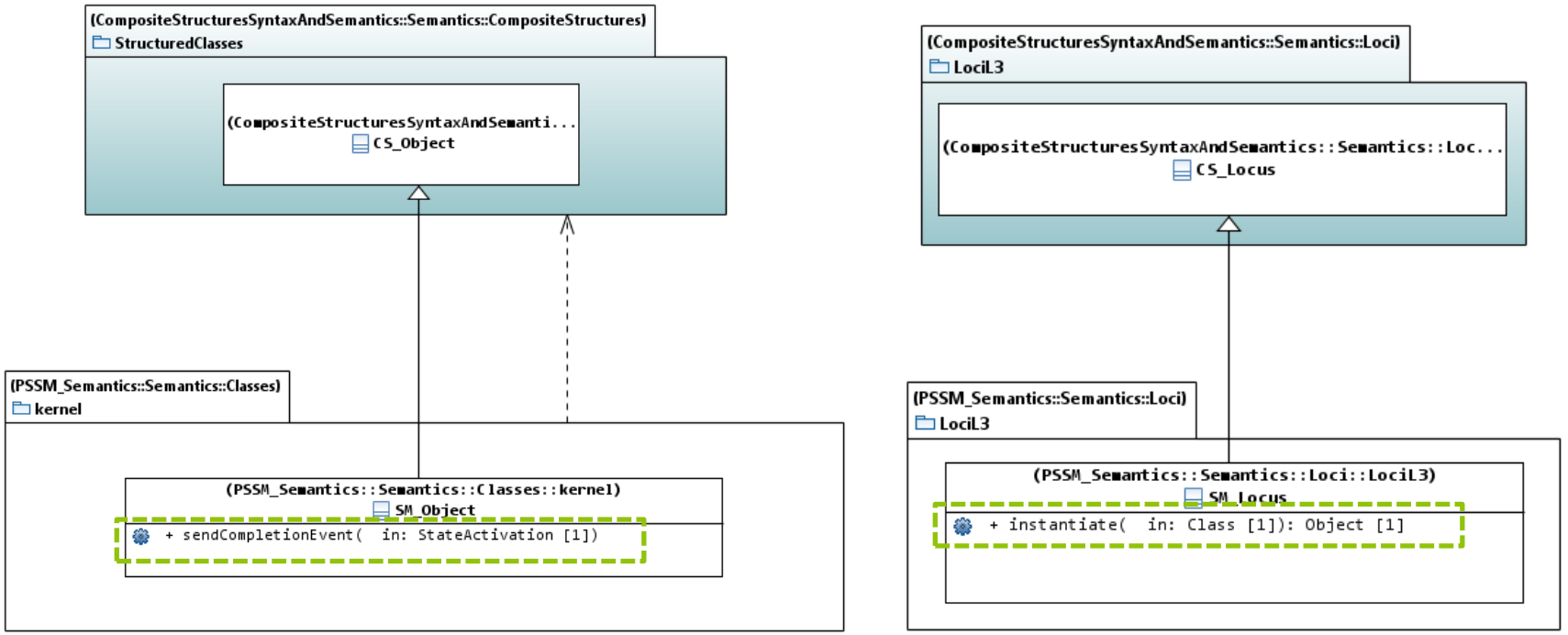


Completion event

- Have priorities over events that are not completion event
- Dispatched in the order in which are they are placed in the pool
- Scope of the completion event is the event that generated it

COMPLETION EVENT (2/2)

IMPLIED EXTENSIONS

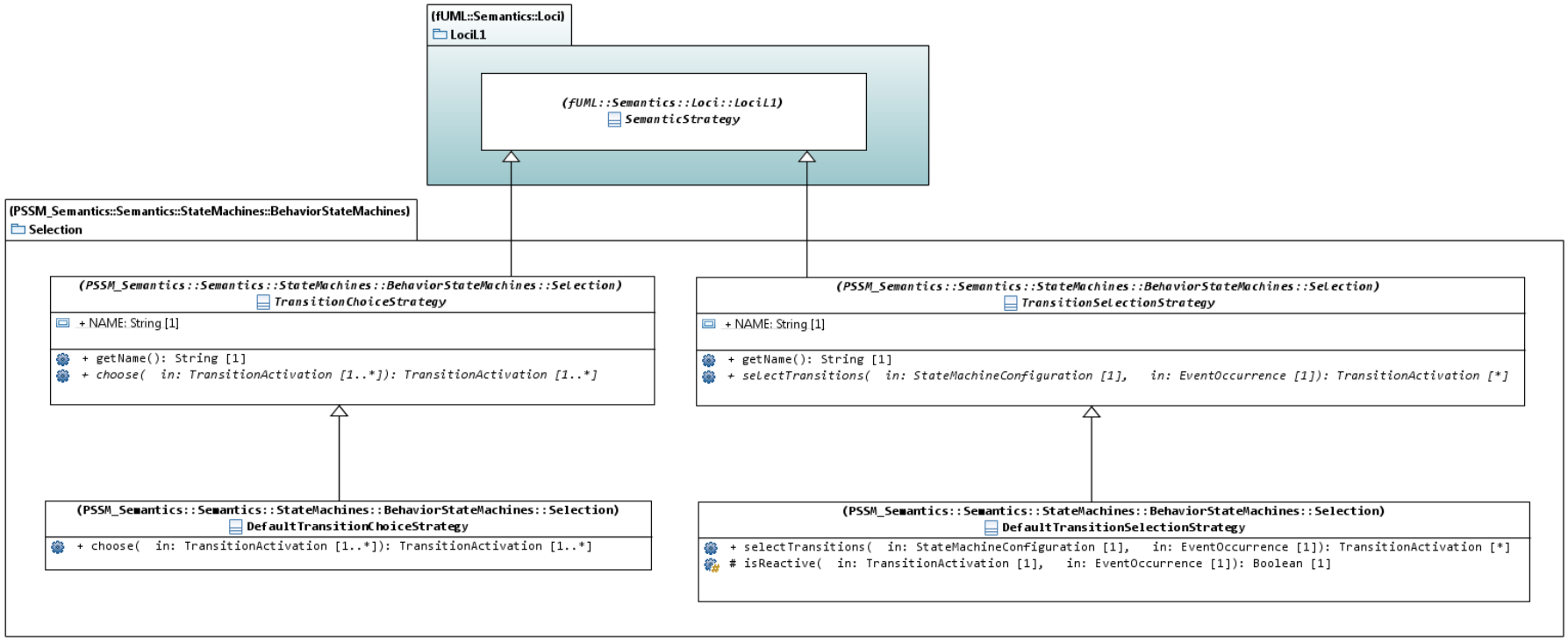


Implied extensions

- *"SM_Object"* has the possibility to place completion events in its pool
- *"SM_Locus"* has its "instantiate" operation overridden to make sure that any class that is instantiated by PSSM leads to get an "SM_Object"

TRANSITIONS SELECTION

TRANSITION SELECTION



A. Transition selection
   - Select a set of transitions that can be triggered by an event occurrence
   - Selection is made based on the current stable configuration

B. Transition choice
   - Reduce the set of transitions that can be fired
   - A single transition or a set of transitions located in orthogonal regions

SELECTION PROCESS

EXCERPT FROM THE IMPLEMENTATION


```
public List<TransitionActivation> selectTransitions(StateMachineConfiguration configuration, EventOccurrence eventOccurrence) {
    // Find for the given configuration the set of transition that can fire.
    // The search starts from the deepest level. If no transition enabled are found at this level
    // then the search continue in the upper level. The search stops when at given level a not empty
    // set of enabled transition is calculated. Transition returned are only those that are not triggered by an event
    // (i.e. Transitions that are automatic or only guarded).
    List<TransitionActivation> fireableTransition = new ArrayList<TransitionActivation>();
    Map<Integer, List<VertexActivation>> cartography = configuration.getCartography();
    int i = cartography.size();
    boolean nextLevel = true;
    while(i >= 1 && nextLevel){
        for(VertexActivation vertexActivation : cartography.get(i)){
            for(TransitionActivation transitionActivation : vertexActivation.getOutgoingTransitions()){
                if(this.isReactive(transitionActivation, eventOccurrence)){
                    fireableTransition.add(transitionActivation);
                }
            }
        }
        if(!fireableTransition.isEmpty()){
            nextLevel = false;
        }else{
            i--;
        }
    }
    return fireableTransition;
}
```

Principle

- Analysis starts from the lowest level of the hierarchy
- Collect all fireable transitions located at this level
- If no fireable transition then move one level up
- Otherwise stop and returns the list of fireable transitions

REDUCTION PROCESS

EXCERPT FROM THE IMPLEMENTATION

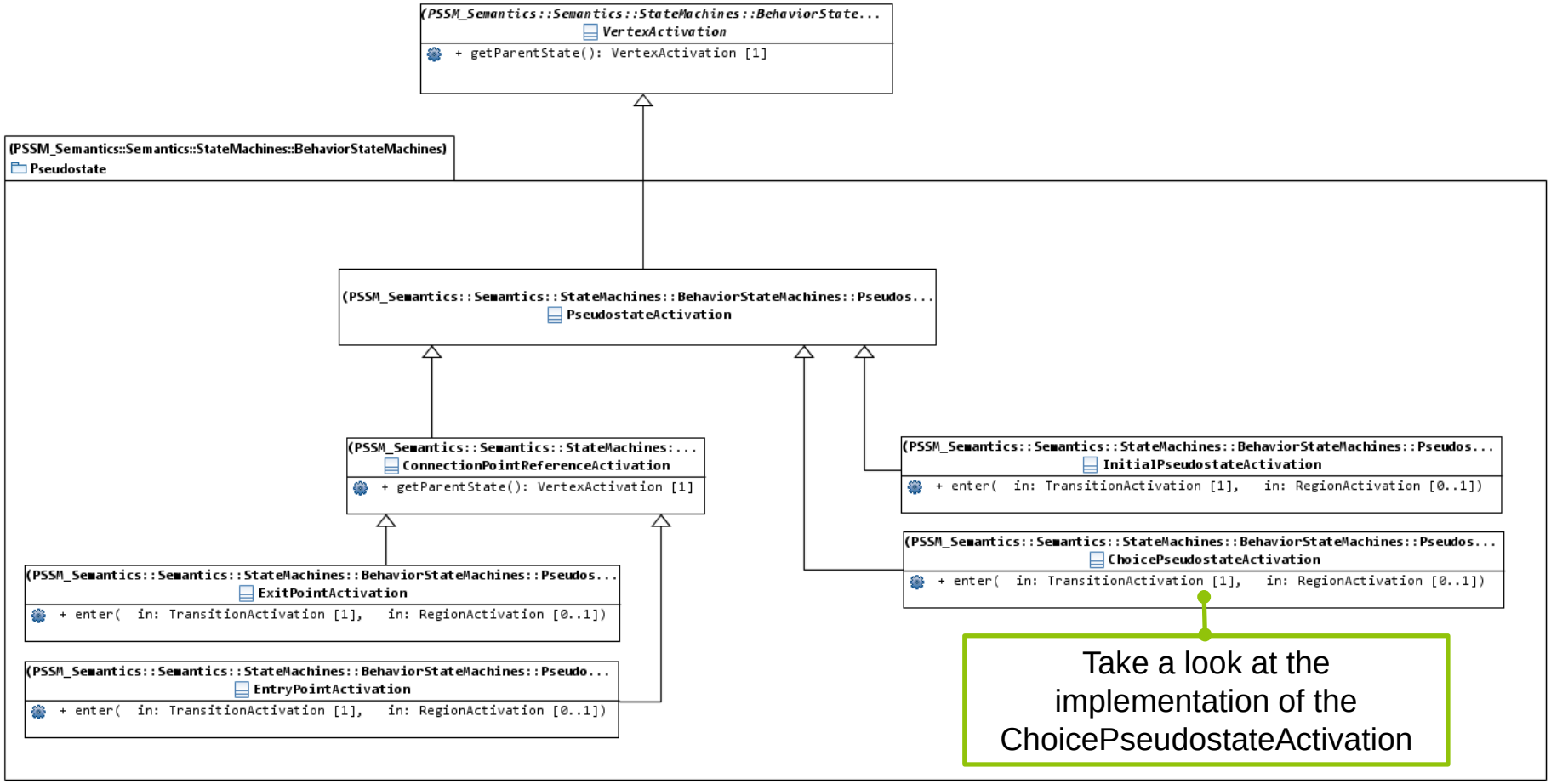

```
@Override
public List<TransitionActivation> choose(List<TransitionActivation> transitionActivations) {
    // The purpose here is to choose among a set of transition that a subset that will
    // effectively be fired. This subset can contain in the most trivial case a single
    // transition. In the case of orthogonal region it can contain several transitions
    // that will be fired using the same event occurrence
    if(transitionActivations.isEmpty()){
        return new ArrayList<TransitionActivation>();
    }else{
        List<List<TransitionActivation>> chosenTransitions = new ArrayList<List<TransitionActivation>>();
        int i = 0;
        while(i < transitionActivations.size()){
            TransitionActivation transition = transitionActivations.get(i);
            List<TransitionActivation> targetSet = this.getTargetSet(transitionActivations.get(i), chosenTransitions);
            if(targetSet==null){
                targetSet = new ArrayList<TransitionActivation>();
                targetSet.add(transition);
                chosenTransitions.add(targetSet);
            }else{
                targetSet.add(transition);
            }
            i++;
        }
        ChoiceStrategy choiceStrategy = new FirstChoiceStrategy();
        int choice = choiceStrategy.choose(chosenTransitions.size()) - 1;
        return chosenTransitions.get(choice);
    }
}
```

Principle

- Based on the selection, find out which transitions will actually be fired
- Transitions in different orthogonal regions are grouped in the same set.
- The final choice is delegated to ChoiceStrategy provided by fUML

PSEUDO STATES: ENTRY / EXIT / CHOICE

PSEUDO STATES

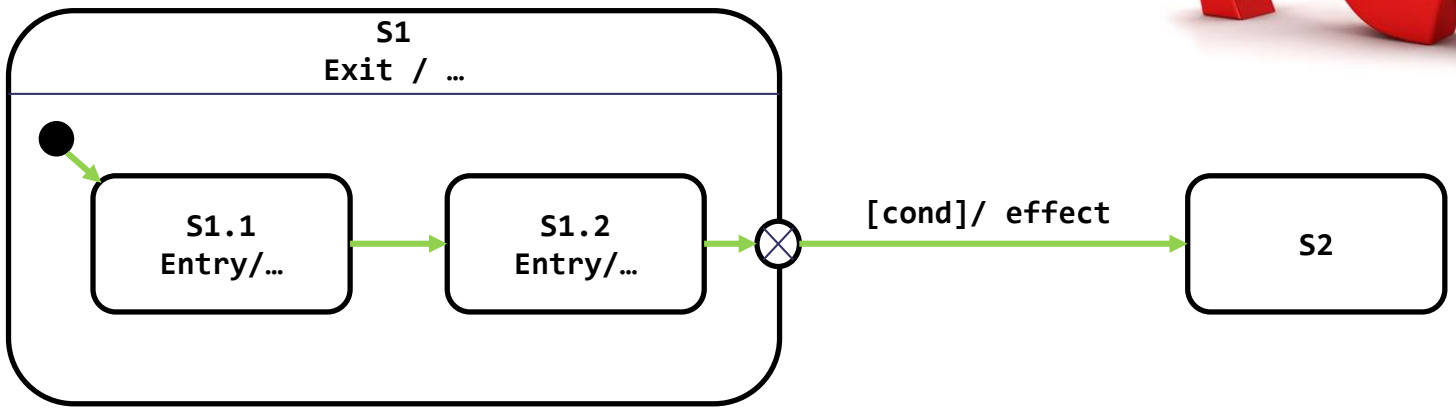


Note

- Recurrent pattern to define a visitor for a pseudo-state
- Each pseudo-state visitor must override the “enter” operation

Transitions originating from entry and exit points

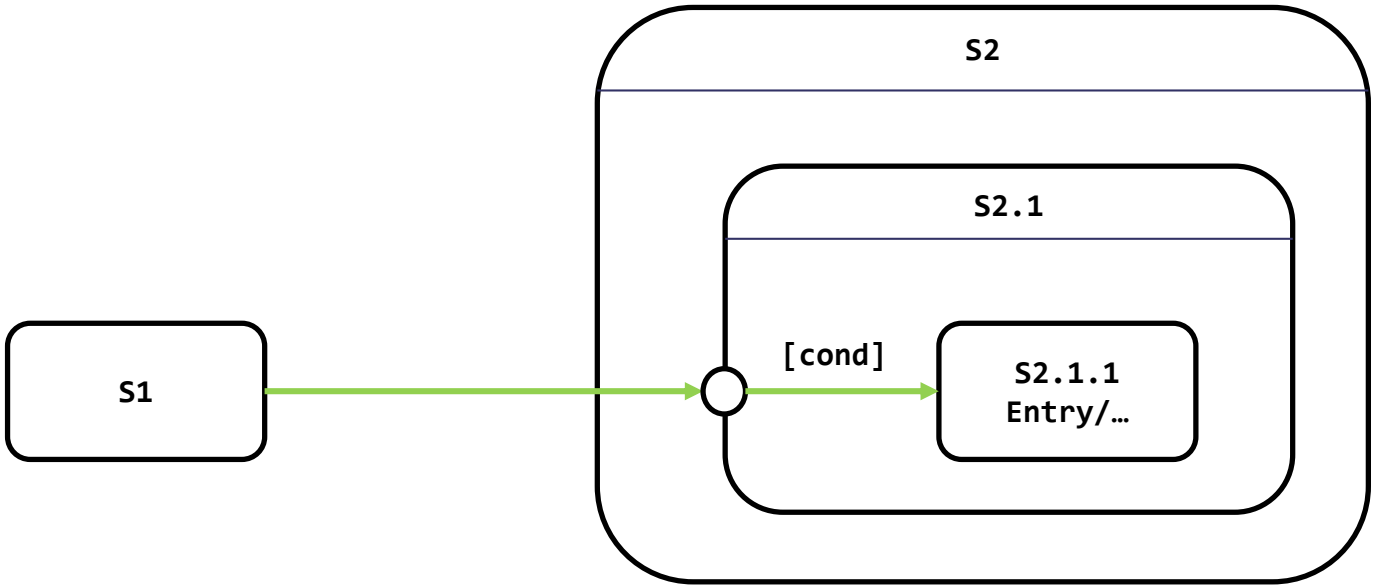
- Can have guards
- Can have an effect



- What should occur in the situation where “cond” is false ?
  - S1 remains the in the configuration
  - The propagation of the execution stops
- Do we need to add modeling constraints ?
  - At least one of the of the guard of the outgoing transitions must be true. If it is not then the model should be considered as being ill-formed.

ENTRY AND EXIT PSEUDO STATES

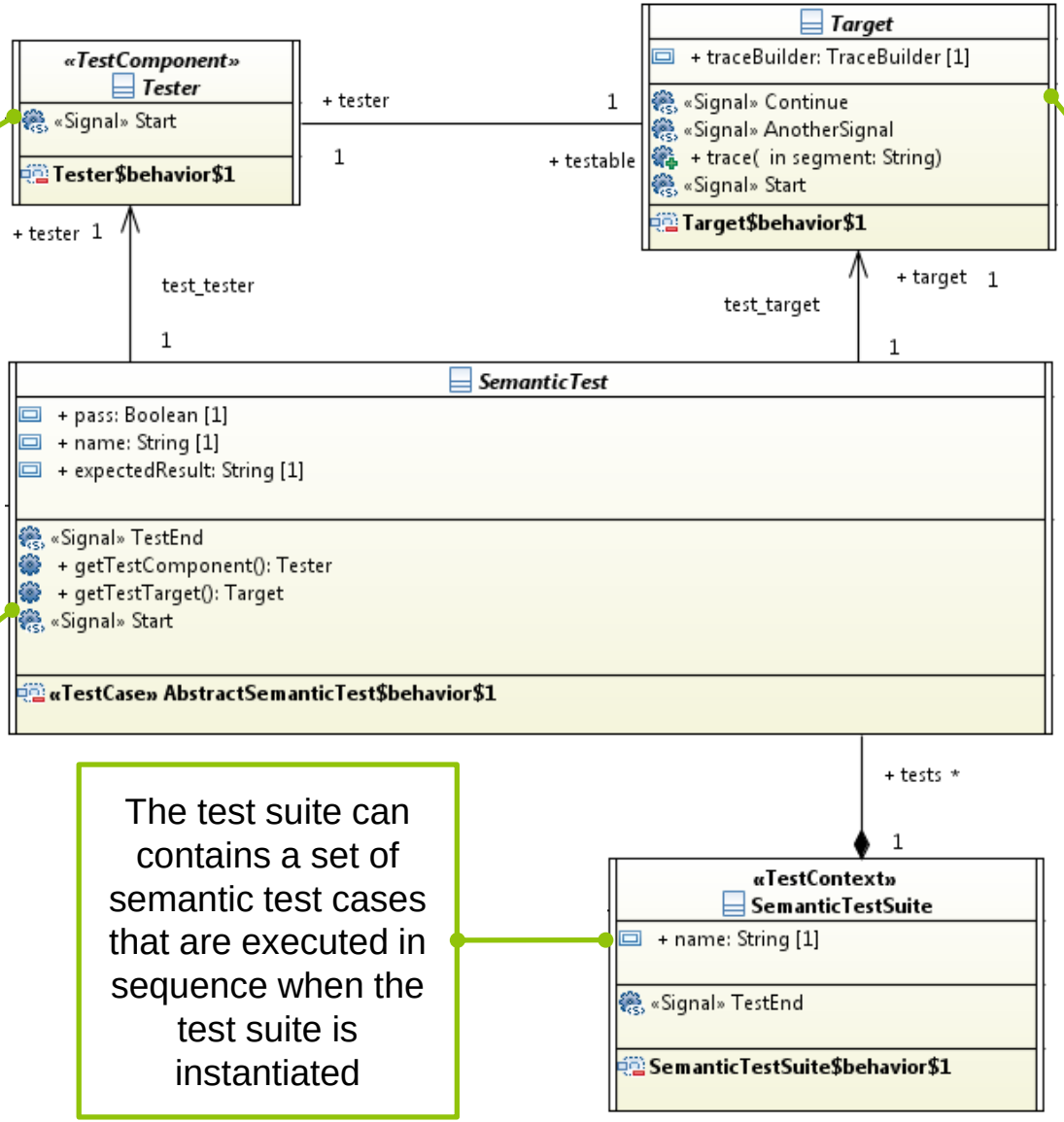
QUESTION ABOUT THE SEMANTICS



- What should occur in the situation in which “cond” is false ?
  - S2 is entered
  - S2.1 is entered
  - S2.1 is treated a simple state

TEST SUITE

ARCHITECTURE REMINDER



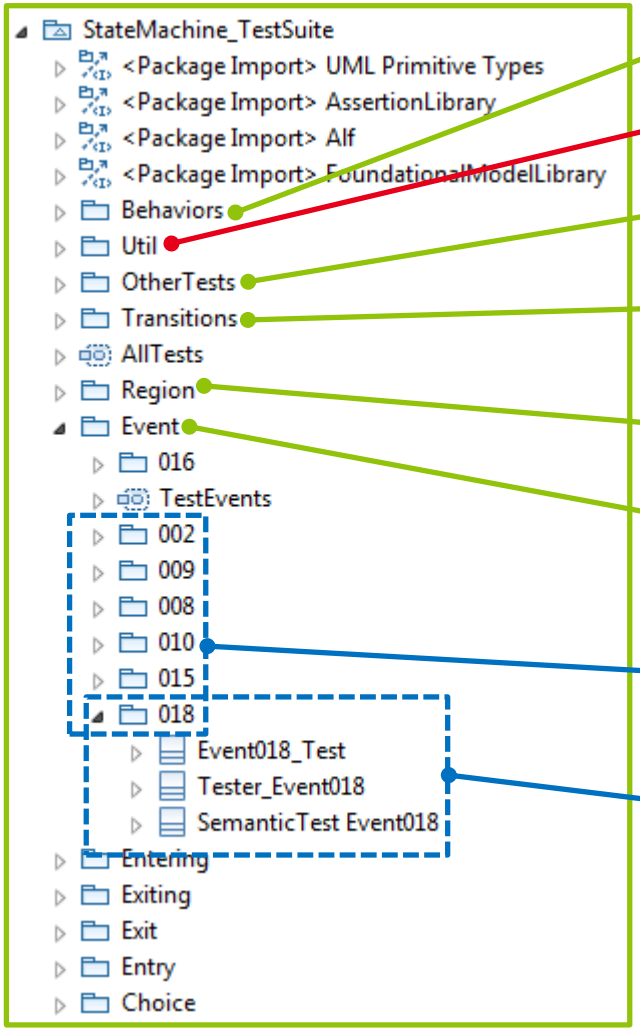
The test component formalizes the stimulation sequence that is received by the test target

Formalizes the test case that controls both the test component and the test target. It also computes the test verdict using trace comparison

The test suite can contains a set of semantic test cases that are executed in sequence when the test suite is instantiated

The test target. This part contains the state-machine that is actually executed to validate part of the semantics

TEST SUITE STRUCTURE (1/2)



Tests related to behaviors executed when a state is entered or left

Base architecture of the test suite (see previous slide)

Additional tests (examples presented in the specification)

Tests related to transition interpretation (effect execution, transition states, triggers, guards, transition types, completion event generation,…)

Tests related to entry of regions (explicit or by default)

Tests related to event consumption (conflicts, priorities, enabled transitions,…)

Each package contains the definition of a semantic test

A semantic test is composed of three classes : an extension of the “Target”, an extension of the “Tester” as well an extension of the the “Semantic Test” class. This can be automatically generated.

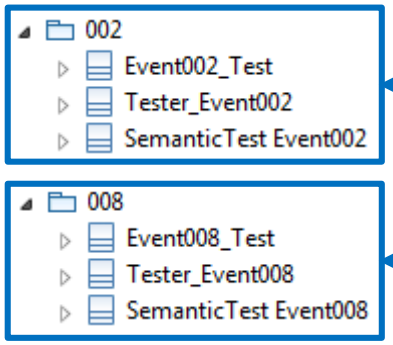
TEST SUITE STRUCTURE (2/2)

LOW LEVEL BEHAVIORS ARE SPECIFIED IN ALF

Main entry point

```
namespace StateMachine_TestSuite;

activity AllTests() {
        Behaviors::Behaviors();
        Transitions::Transitions();
        OtherTests::OtherTests();
        Event::TestEvents();
        Entering::TestsEntering();
        Exiting::TestsExiting();
        Exit::TestsExit();
        Entry::TestsEntry();
        Choice::TestsChoice();
}
```

```
namespace StateMachine_TestSuite::Event;

private import StateMachine_TestSuite::Util::Architecture::SemanticTestSuite;
private import StateMachine_TestSuite::Util::Architecture::SemanticTest;

private import '002'::'SemanticTest Event002';
private import '008'::'SemanticTest Event008';

activity TestEvents() {
        let name : String = "Events";
        let tests : SemanticTest = new SemanticTest[]{};
        /*Event002*/
        e002 = new 'SemanticTest Event002'();
        e002.name = "Event002";
        e002.expectedResult ="S1(entry)::S1.1(entry)::S1(exit)";
        tests->add(e002);
        /*Event008*/
        e008 = new 'SemanticTest Event008'();
        e008.name = "Event008";
        e008.expectedResult ="T2(effect)::T3(effect)";
        tests->add(e008);
        /*Test suite*/
        suite = new SemanticTestSuite(tests, name);
}
```



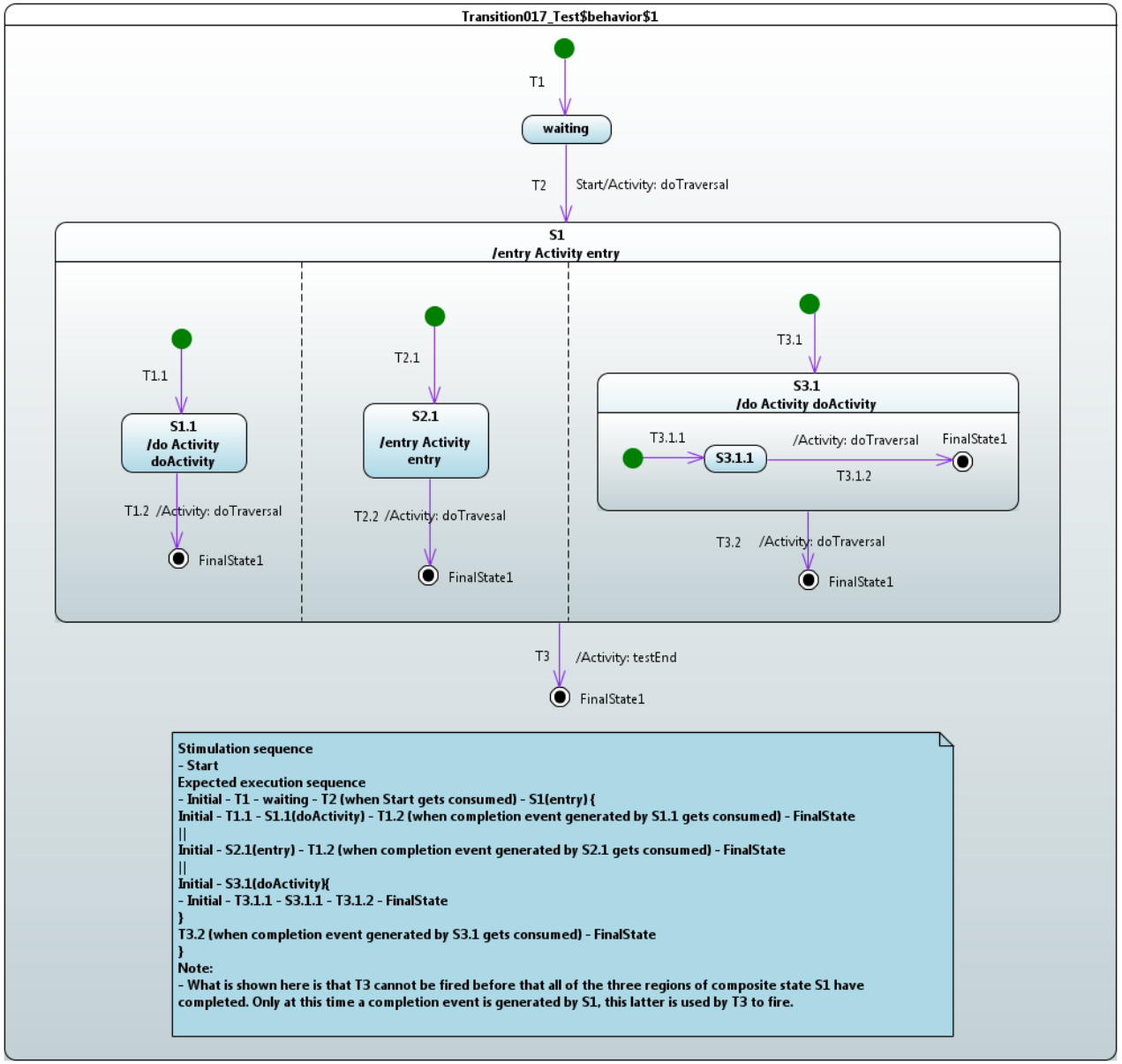
"**expectedResult**" defines the trace that is expected as result of the test. **Note**: in the future it will be changed to be a set defining possible traces.

TESTS REVIEW

ONE TEST PER CATEGORY - ILLUSTRATE HOW IT WORKS

Requirement

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. (p.328 - 329)

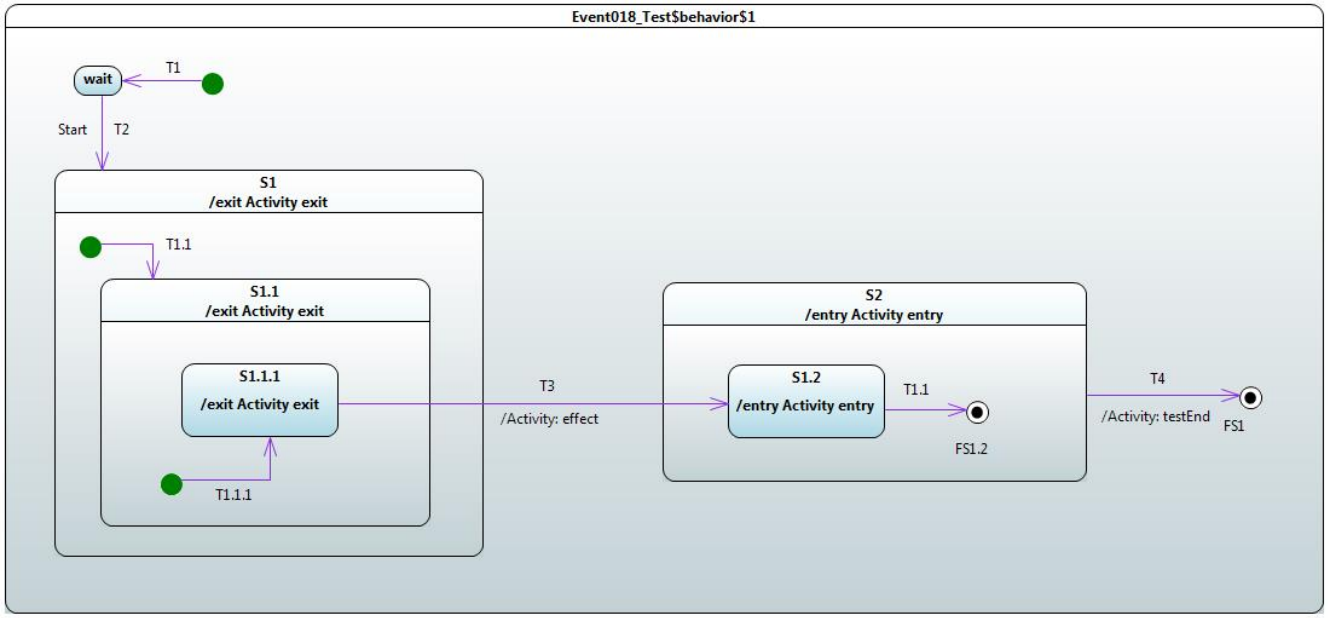


TEST 2

EXPLICIT ENTRY AND EXIT

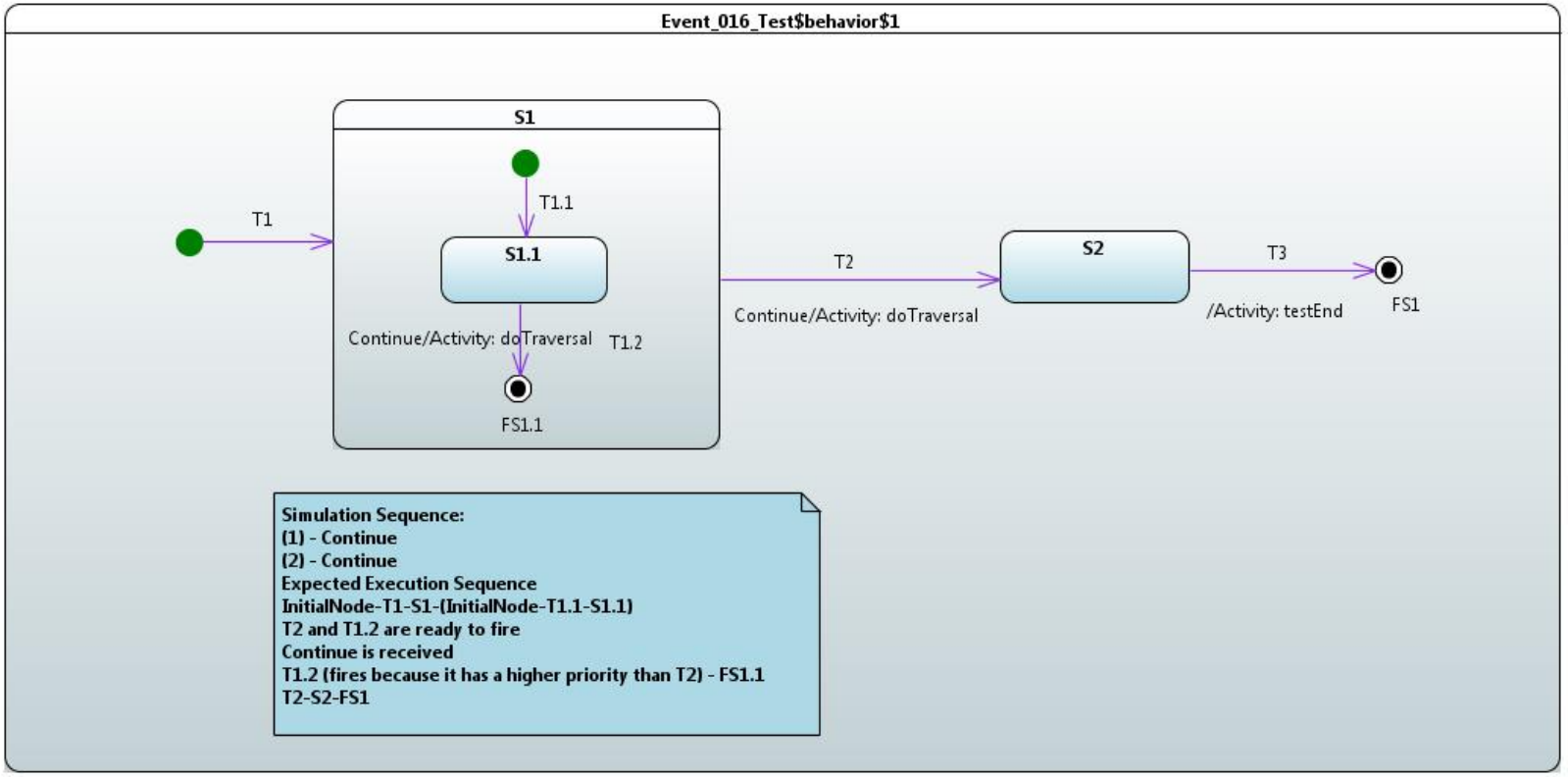
Requirement

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. (p.331)



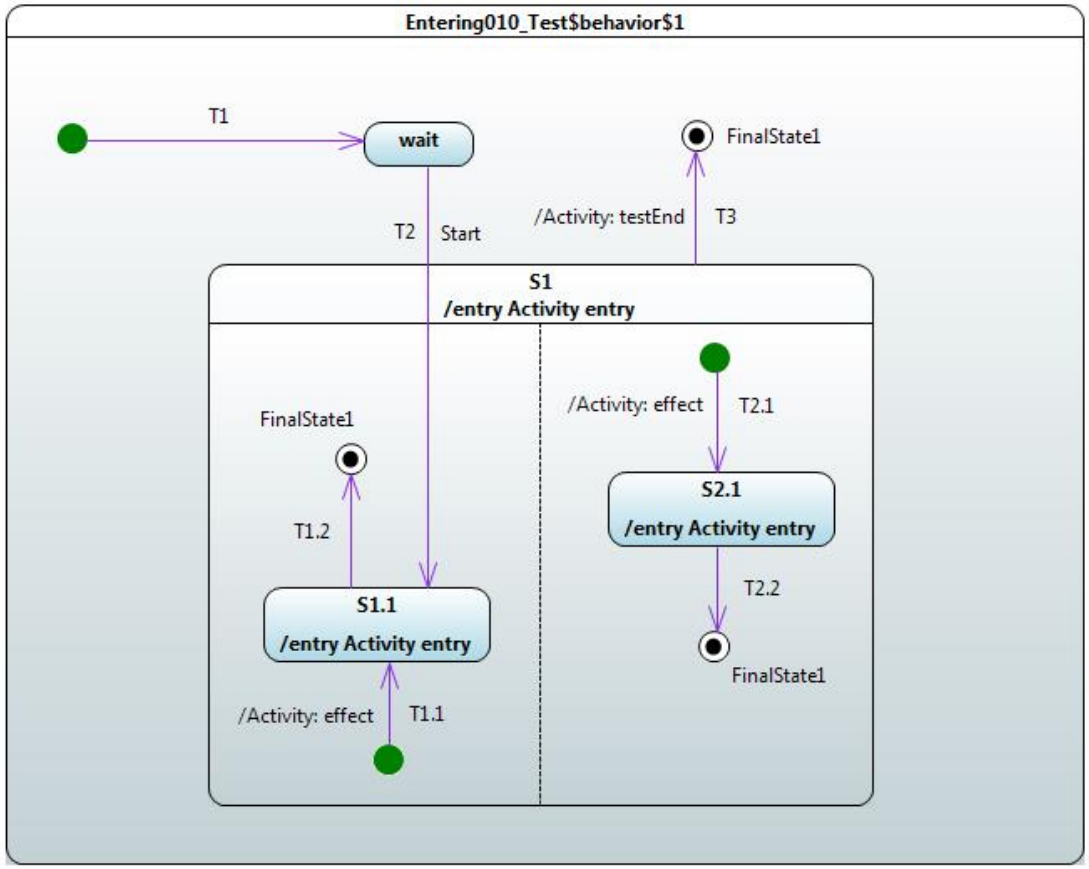
Requirement

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. (p.331)



Requirement

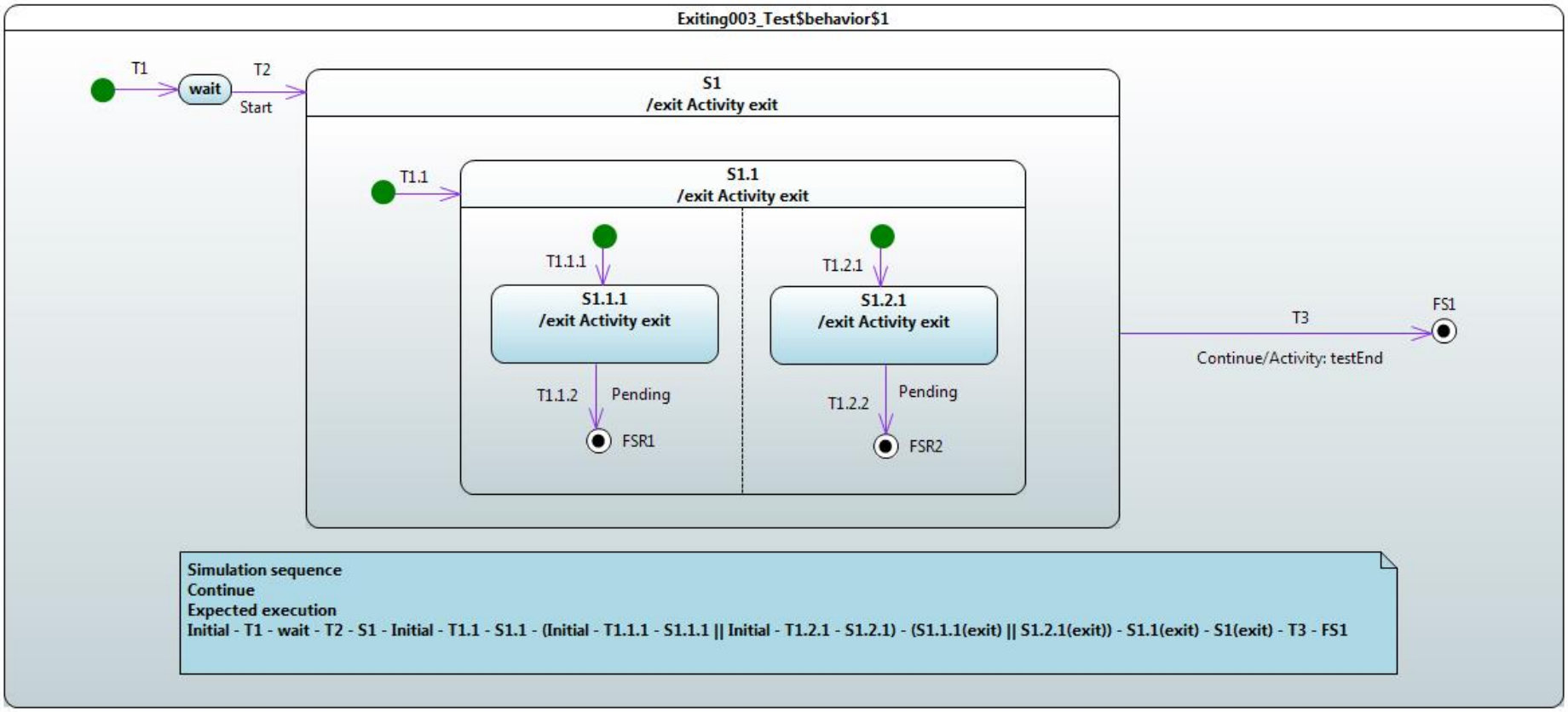
If the composite State is also an orthogonal State with multiple Regions, each of its Regions is also entered, either by default or explicitly. (p.324)



TEST 5
EXITING A COMPOSITE TEST WITH ORTHOGONAL REGIONS

Requirement

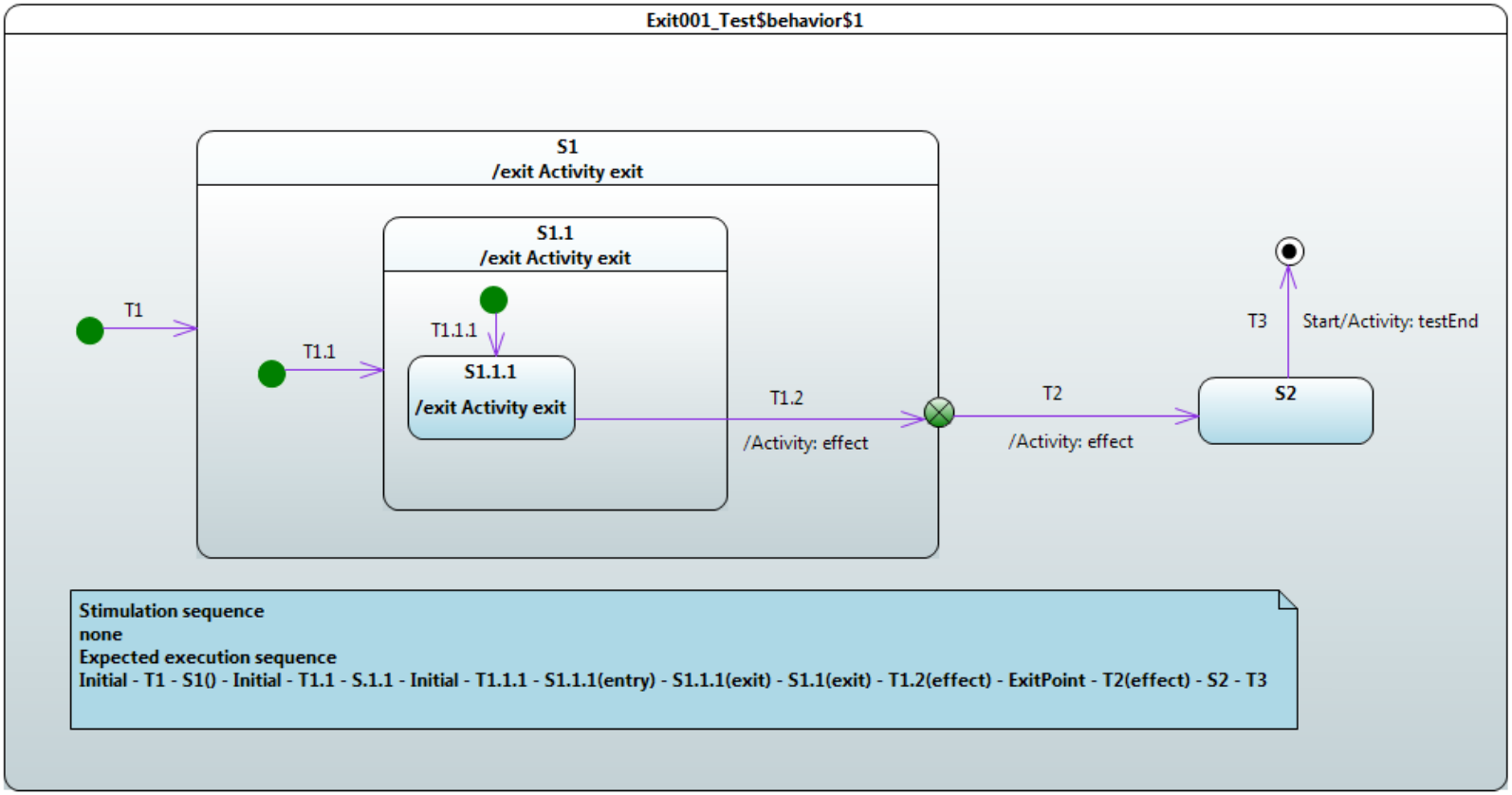
When exiting from a composite State, exit commences with the innermost State in the active state configuration. This means that exit Behaviors are executed in sequence starting with the innermost active State. (p.324)



TEST 6
EXIT A COMPOSITE USING A EXIT POINT

Requirement

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). (p.327)



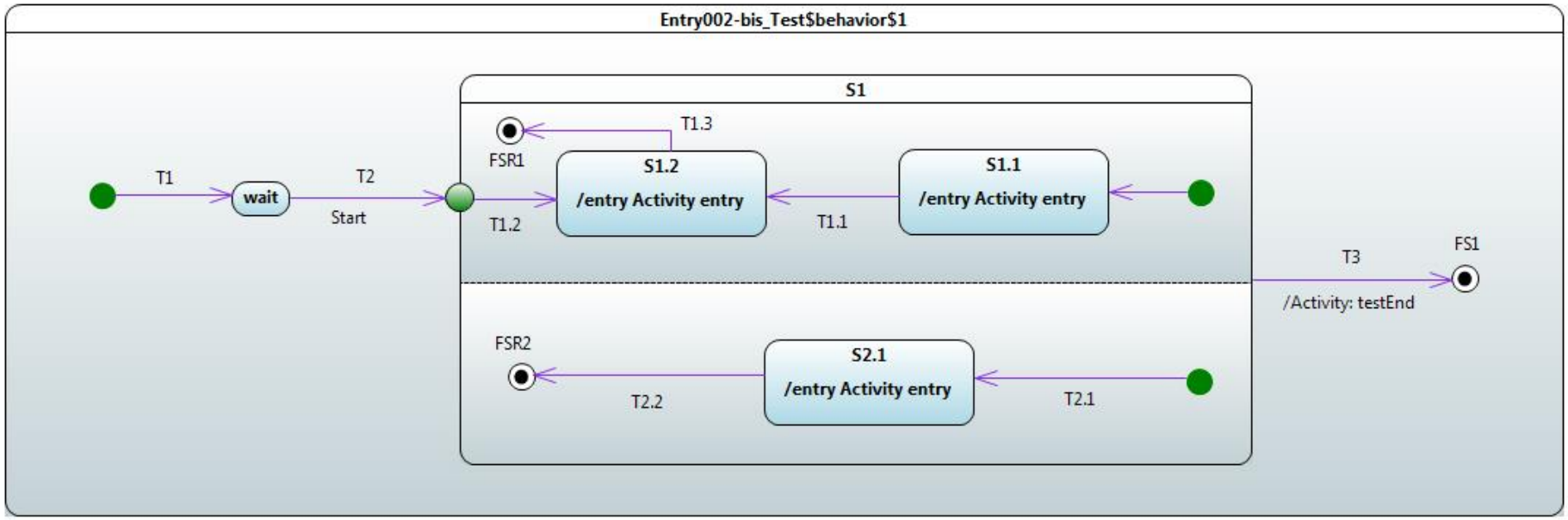
TEST 8

ENTER A COMPOSITE STATE USING A ENTRY POINT

Requirement

If the owning State has an associated entry Behavior, this Behavior is executed before any behavior associated with the outgoing Transition.

If multiple Regions are involved, the entry point acts as a fork Pseudostate

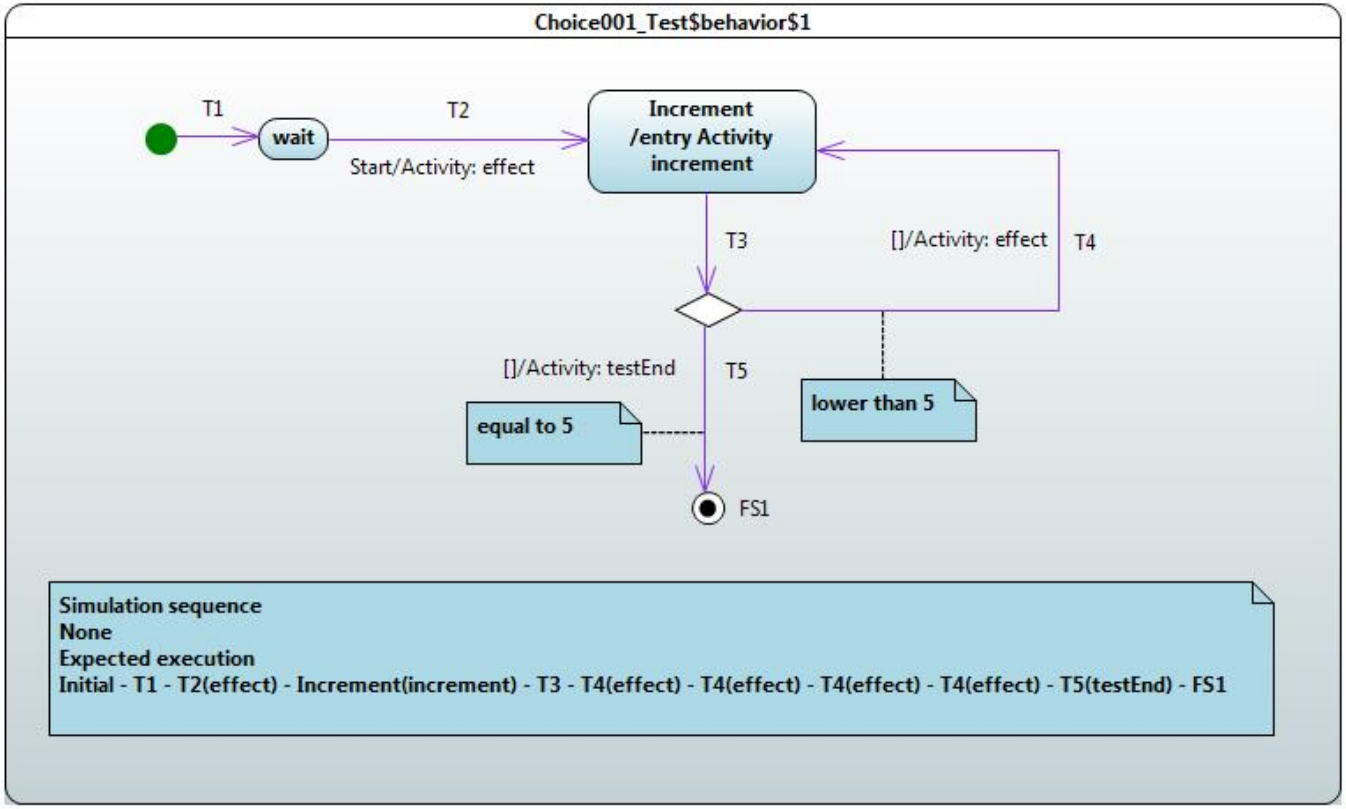


TEST 9

TRAVERSE A CHOICE PSEUDO STATE

Requirement

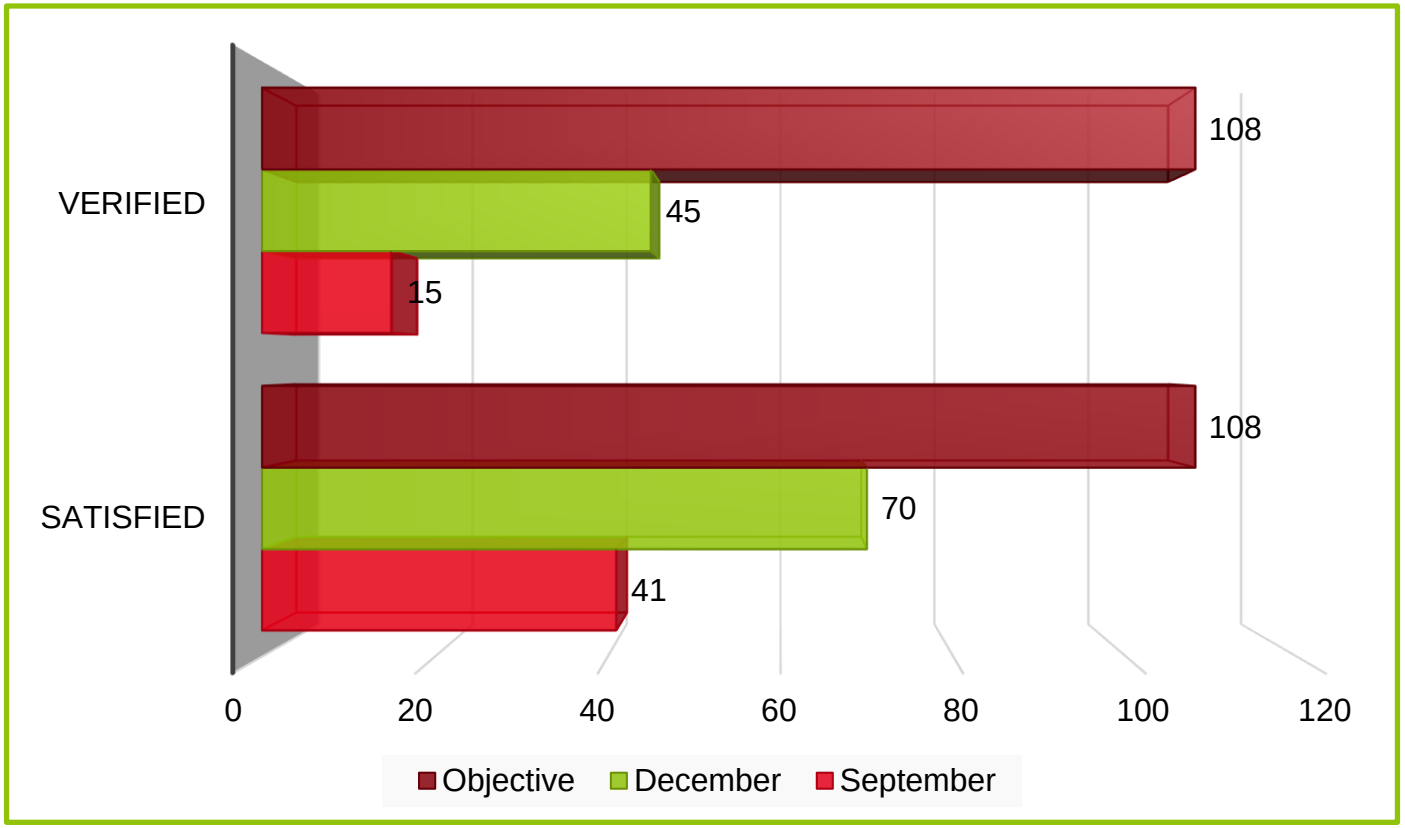
The guard Constraints on all outgoing Transitions are evaluated dynamically, when the compound transition traversal reaches this Pseudostate (p.327)



DEMO

GENERAL ASSESSEMENT

PROGRESS EVALUATION FROM LAST MEETING & OBJECTIVES FOR SUBMISSION



Notes

- Tested requirements: +30
  - Effort to consolidate existing prototype
- Satisfied requirements: + 29
- Some time allocated to move from fUML 1.1 to fUML 1.2

WHAT REMAINS TO BE SUPPORTED ?

A. From mandatory requirements point of view

- Local transitions
- Deferred events
- History (shallow and deep)
  - Require a significant effort
- Fork
- Join
- Terminate
- Junction
  - Require a significant effort

Scope – Initial submission deadline

B. From the extra requirements point of view

- Call events
- Sub-machines

After initial submission

C. From the test suite point of view

After initial submission

- Test suite can be execute using an asynchronous implementation
  - Test results must be a set of possible traces
  - Require a significant effort to refactor


Papyrus