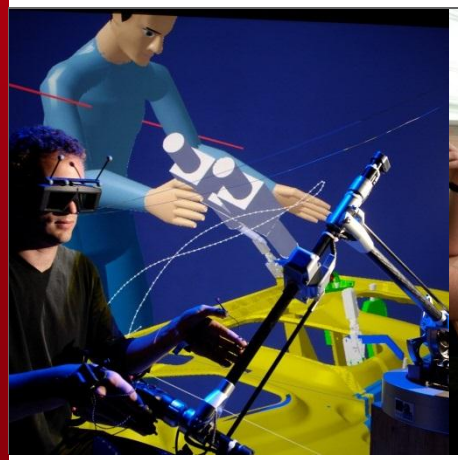
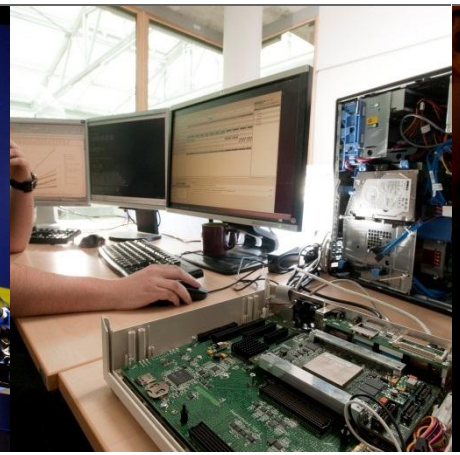
FROM RESEARCH TO INDUSTRY

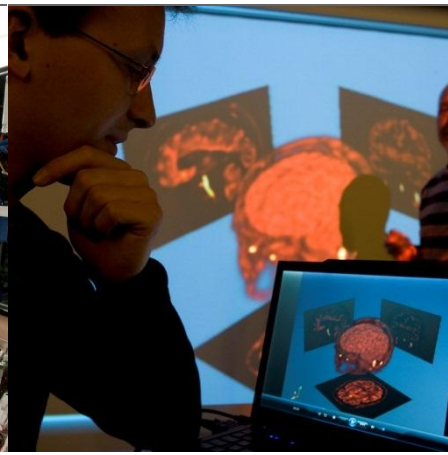
cea tech

[PSSM] – SEMANTIC MODEL

Jérémie TATIBOUET (CEA LIST)
Arnaud CUCCURU (CEA LIST)







list

INSTITUT CARNOT CEA LIST digiteo

OUTLINE

A. What is currently covered

B. Detailed overview of the semantic model

- Package structure
- Base (i.e. abstract) semantic visitors
- State semantics
- Pseudo states semantics
- Transition semantics
- State machine configuration
- Event dispatching and transition selection

C. What needs to be supported by December?

WHAT IS COVERED

UML State machines

Mandatory Requirements

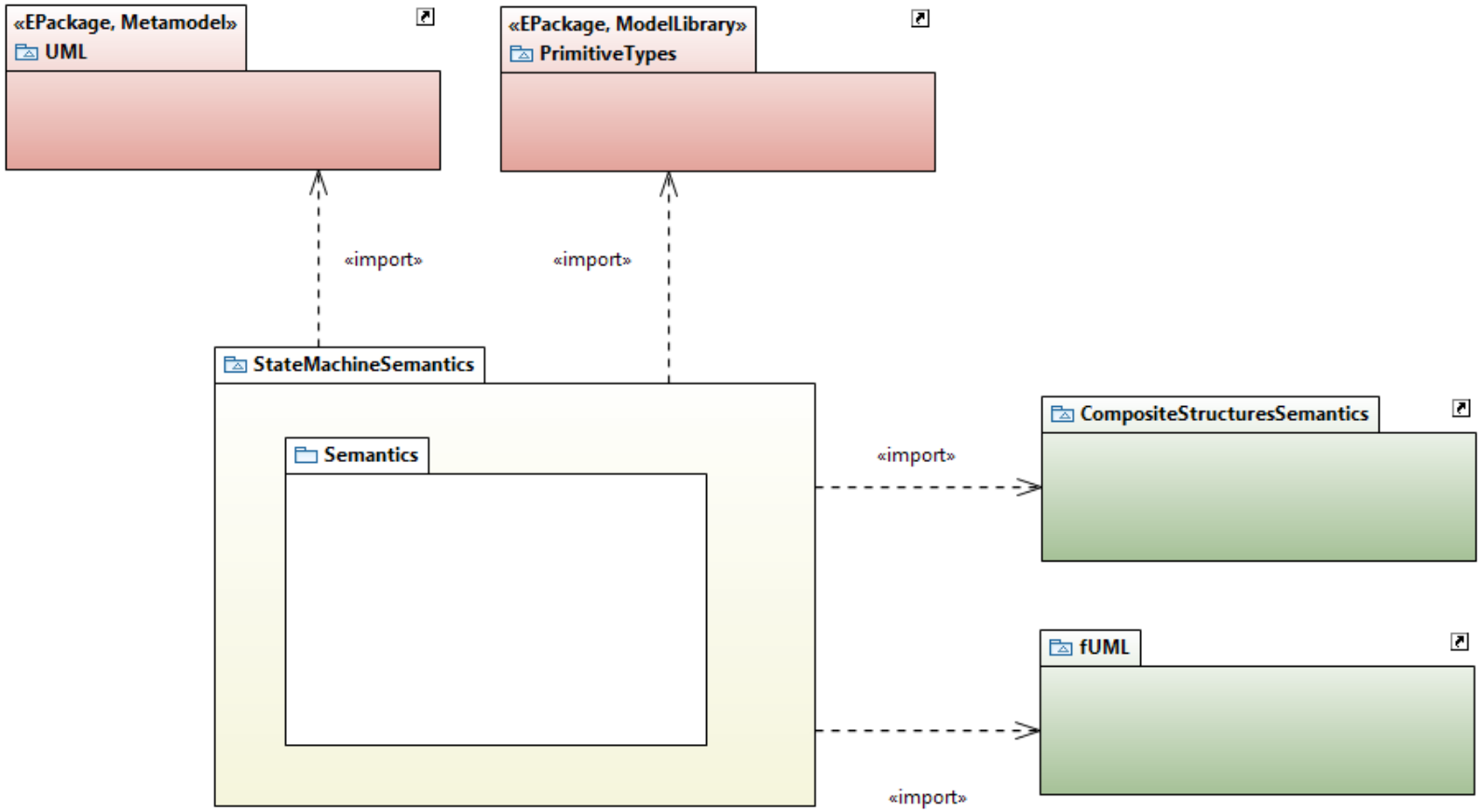
- State
- Guarded transition
- FinalState
- Triggered transition (SignalEvent)
- State-machine as classifier behavior
- Completion transition

- Triggered Transition (CallEvent)
- Orthogonal regions
- Local Transition
- doActivity of a State
- History, Fork, Join, Choice pseudostates
- Standalone State-machine

- Submachines (cf. section 6.6.1)
- Protocol State-machines (cf. section 6.6.2)
- State machines redefinition (cf. section 6.6.3)
- State-machine as method

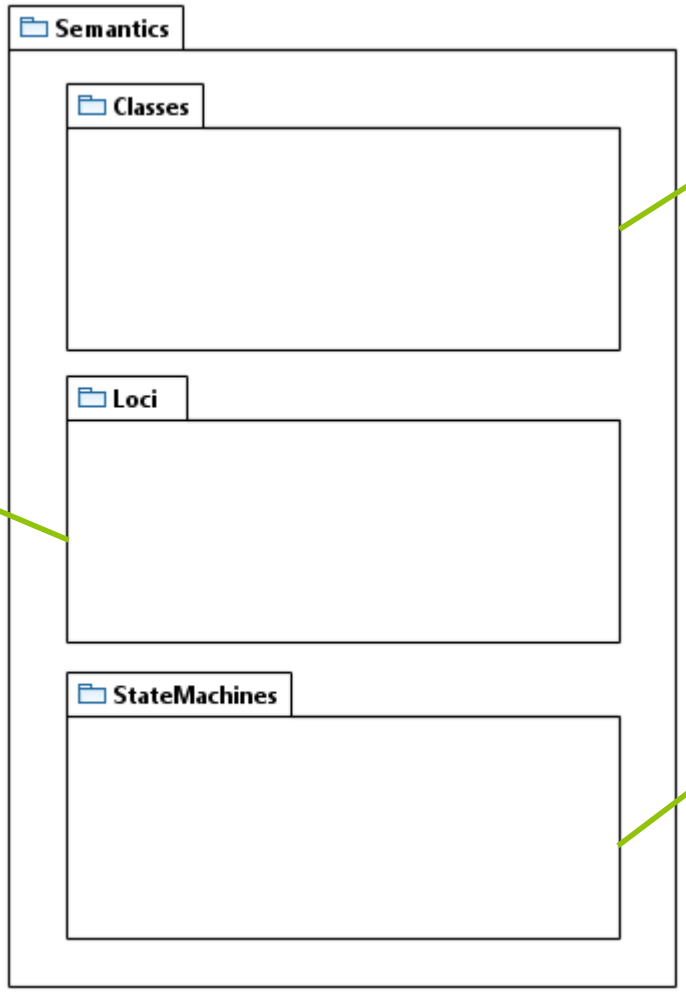
SEMANTIC MODEL

DEPENDENCIES



SEMANTIC MODEL

PACKAGE STRUCTURE



Provides extensions to values semantics

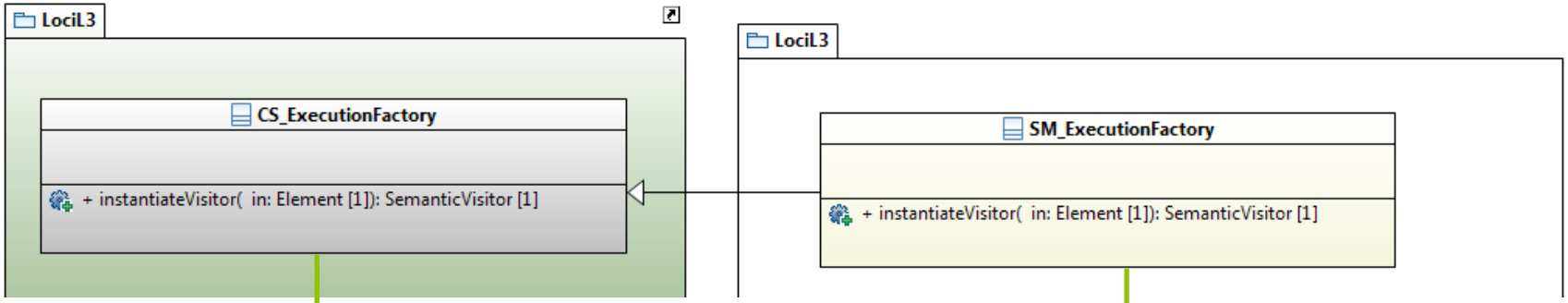
Provides extensions to deal with newly introduced visitors

Provides a set of visitors that capture the semantics of meta-classes targeted by the mandatory requirements

INSTANTIATION

SEMANTIC MODEL

LOCI::LOCIL3 (INSTANTIATION)



Visitor factory provided by PSCS

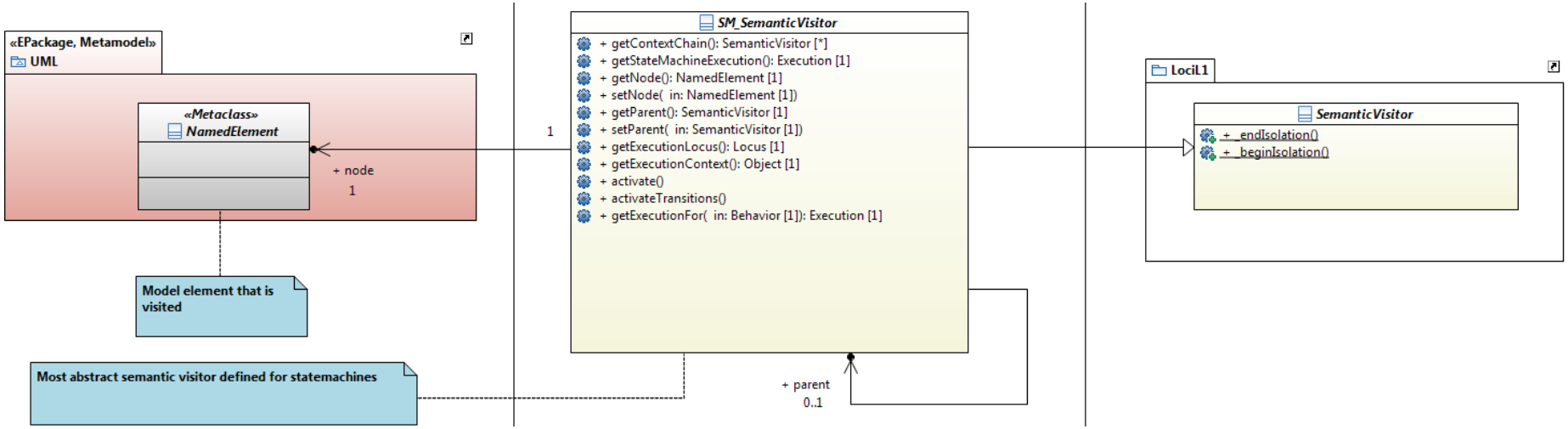
Its extension in the context of PSSM

The implementation is done in the usual philosophy (i.e. a set of if conditions)

```
public SemanticVisitor instantiateVisitor(Element element) {
    SemanticVisitor visitor = null ;
    if(element instanceof StateMachine){
        visitor = new StateMachineExecution();
    }else if (element instanceof Pseudostate) {
        Pseudostate pseudostate = (Pseudostate) element;
        switch(pseudostate.getKind()){
            case INITIAL_LITERAL: visitor = new InitialPseudostateActivation(); break;
            case ENTRY_POINT_LITERAL: visitor = new EntryPointActivation(); break;
            case EXIT_POINT_LITERAL: visitor = new ExitPointActivation(); break;

        }
    }else if (element instanceof State) {
        if(element instanceof FinalState){
            visitor = new FinalStateActivation();
        }else{
            visitor = new StateActivation() ;
        }
    }else if (element instanceof Transition) {
```

SEMANTIC MODEL

LOCI::LOCIL3 (TOP COMMON ANCESTOR)



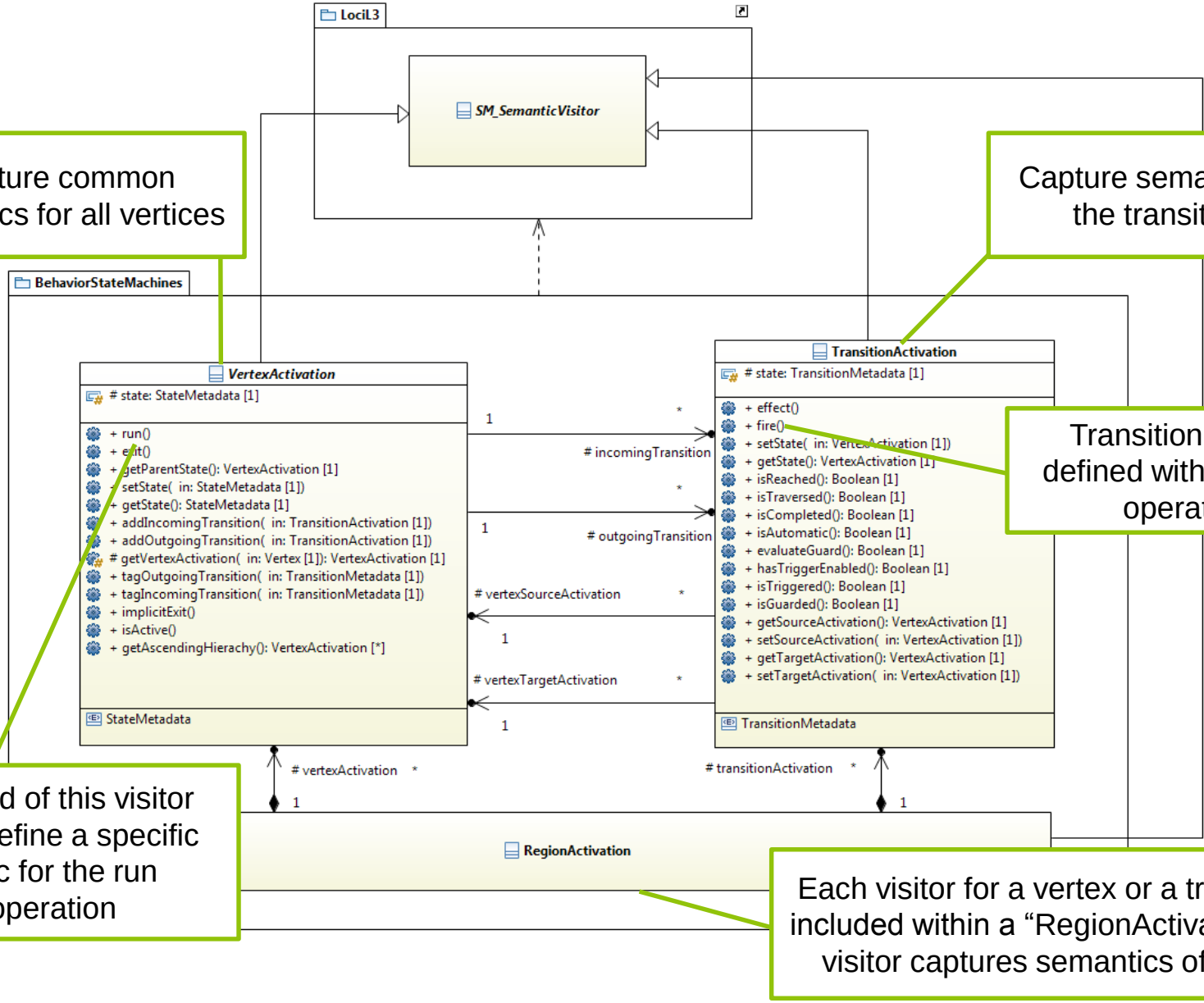
SM_SemanticVisitor

- Specialization of most basic visitor provided by fUML
- Each visitor extending this one can access its parent
- Group useful operations
  - "*getContextChain*" - Provide the path to access the top of the state-machine
  - "*getStateMachineExecution*" – The execution governing the interpretation
  - "*getExecutionContext*" – Return the context of the state-machine execution
  - "*activate*" – Encode owned node instantiation
  - "*activateTransition*" – Encode transition instantiation

BEHAVIOR STATE MACHINES
[BASE VISITORS]

SEMANTIC MODEL

BASE VISITORS



Capture common semantics for all vertices

Capture semantics for the transitions

Transition logic is defined within the fire operation

All child of this visitor must define a specific logic for the run operation

Each visitor for a vertex or a transition is included within a “RegionActivation”. This visitor captures semantics of regions

SEMANTIC MODEL

IMPLEMENTATION OF FIRE

TransitionActivation

- # state: TransitionMetadata [1]
- + effect()
- + fire()
- + setState(in: VertexActivation [1])
- + getState(): VertexActivation [1]
- + isReached(): Boolean [1]
- + isTraversed(): Boolean [1]
- + isCompl
- + isAutom
- + evaluate
- + hasTrigg
- + isTrigger
- + isGuarde
- + getSourc
- + setSourc
- + getTarge
- + setTarge

TransitionM

Should be completed to support local transitions

```
public void fire(){
    Transition node = (Transition) this.getNode();

    this.setState(TransitionMetadata.TRAVERSED);
    /*1. Exit the source state (if transition is local or external)*/
    if(node.getKind()!=TransitionKind.INTERNAL_LITERAL){
        this.vertexSourceActivation.exit(this);
    }

    /*2. Execute the effect on the transition if present*/
    this.effect();
    /*3. Run the target state (if transition is local or external)*/

    if(node.getKind()!=TransitionKind.INTERNAL_LITERAL){
        this.vertexTargetActivation.run();
    }
}
```

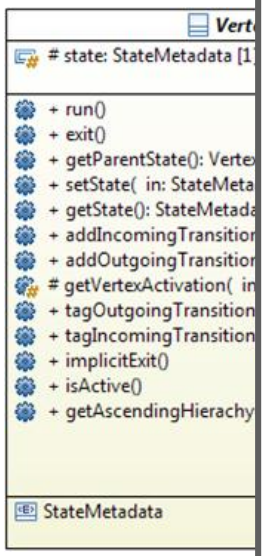
Exit source state

Execute the effect

Propagate execution flow to the target vertex

SEMANTIC MODEL

BASE IMPLEMENTATION OF RUN AND EXIT

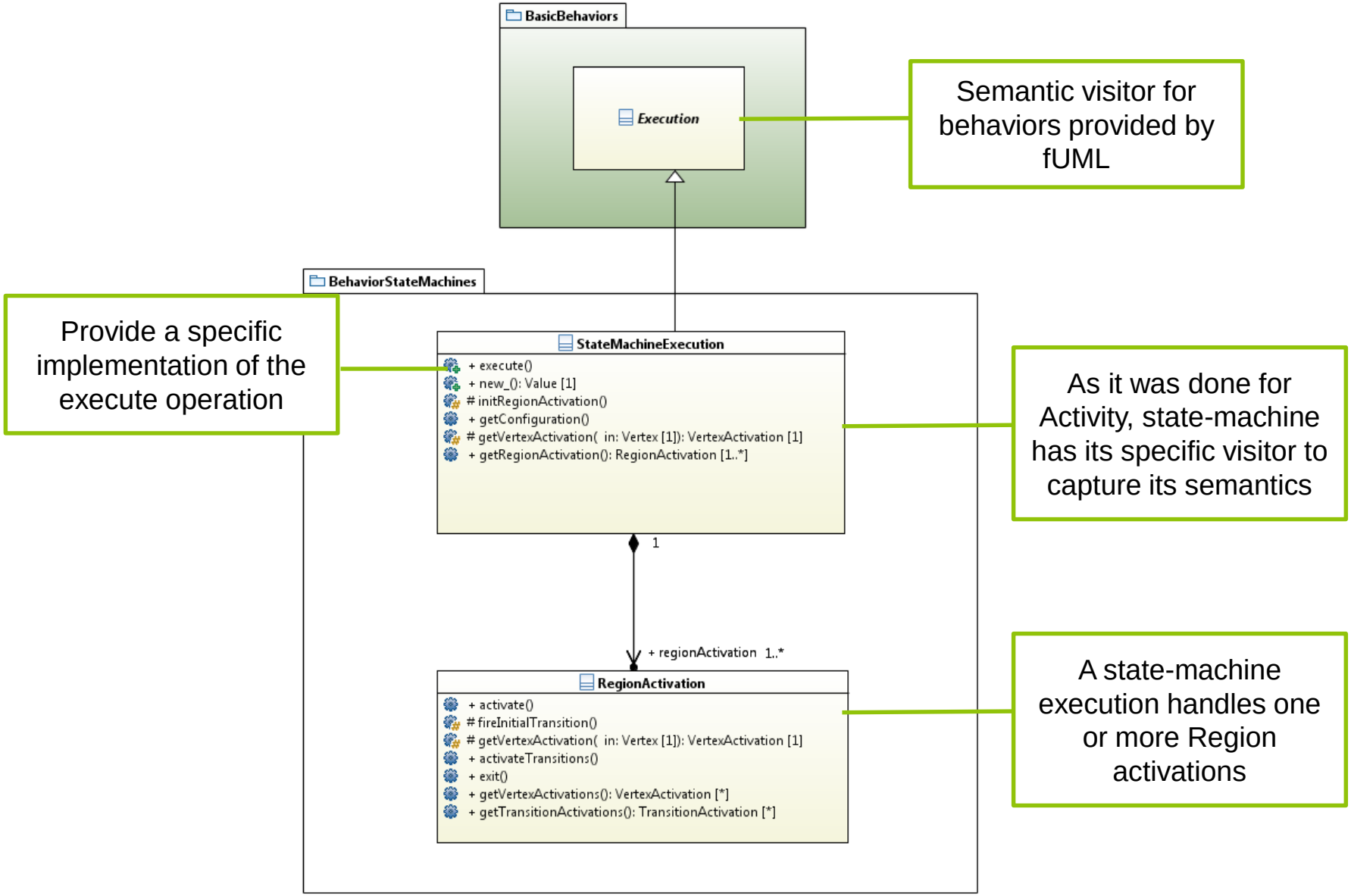



```java
/**
 * Describes the semantics of a vertex
 */
public void run(){

    /*1. The vertex becomes active*/
    this.setState(StateMetadata.ACTIVE);
    /*2. Vertex outgoing transitions are tag as REACHED*/
    this.tagOutgoingTransitions(TransitionMetadata.REACHED);


}

/**
 * Describes the semantics of a vertex when exited
 */
public void exit(TransitionActivation exitingTransition){


    /*2. The incoming transitions of this vertex get back to the NONE status*/
    this.tagIncomingTransitions(TransitionMetadata.NONE);
    /*3. The vertex becomes IDLE*/
    this.setState(StateMetadata.IDLE);

}
```

- Update the meta-data about the state
- Update the state of outgoing/incoming transitions
- Some Moka implementation specific statements for animation (hidden here)

SEMANTIC MODEL

AN EXECUTION FOR A STATE-MACHINE



SEMANTIC MODEL

IMPLEMENTATION OF THE EXECUTION OPERATION


```
@Override
public void execute() {
    /*0. Initialization*/
    if(this.context!=null && this.context.objectActivation!=null){
        this.context.register(new SM_EventAccepter(this));
    }
    this.initRegions();
    /*1. Create visitors for all vertices*/
    for(RegionActivation activation: this.regionActivation){
        activation.activate();
    }
    /*2. Create visitors for all transitions*/
    for(RegionActivation activation: this.regionActivation){
        activation.activateTransitions();
    }
    /*3. Fire "concurrently" all initial transition in the different regions*/
    for(RegionActivation activation: this.regionActivation){
        activation.fireInitialTransition();
    }
    /*4. Try to find another transition available to fire*/
    TransitionSelectionStrategy selectionStrategy = (TransitionSelectionStrategy) this.locus.factory.getStrategy(TransitionSelectionStrategy.NAME);
    List<TransitionActivation> fireableTransition = selectionStrategy.selectTransitions(this.configuration);
    while(!fireableTransition.isEmpty()){
        TransitionChoiceStrategy choiceStrategy = (TransitionChoiceStrategy)this.locus.factory.getStrategy(TransitionChoiceStrategy.NAME);
        TransitionActivation transitionActivation = choiceStrategy.choose(fireableTransition);
        transitionActivation.fire();
        fireableTransition = selectionStrategy.selectTransitions(this.configuration);
    }
}
```

State-machine instantiation phase (i.e. construction of semantic visitor tree).

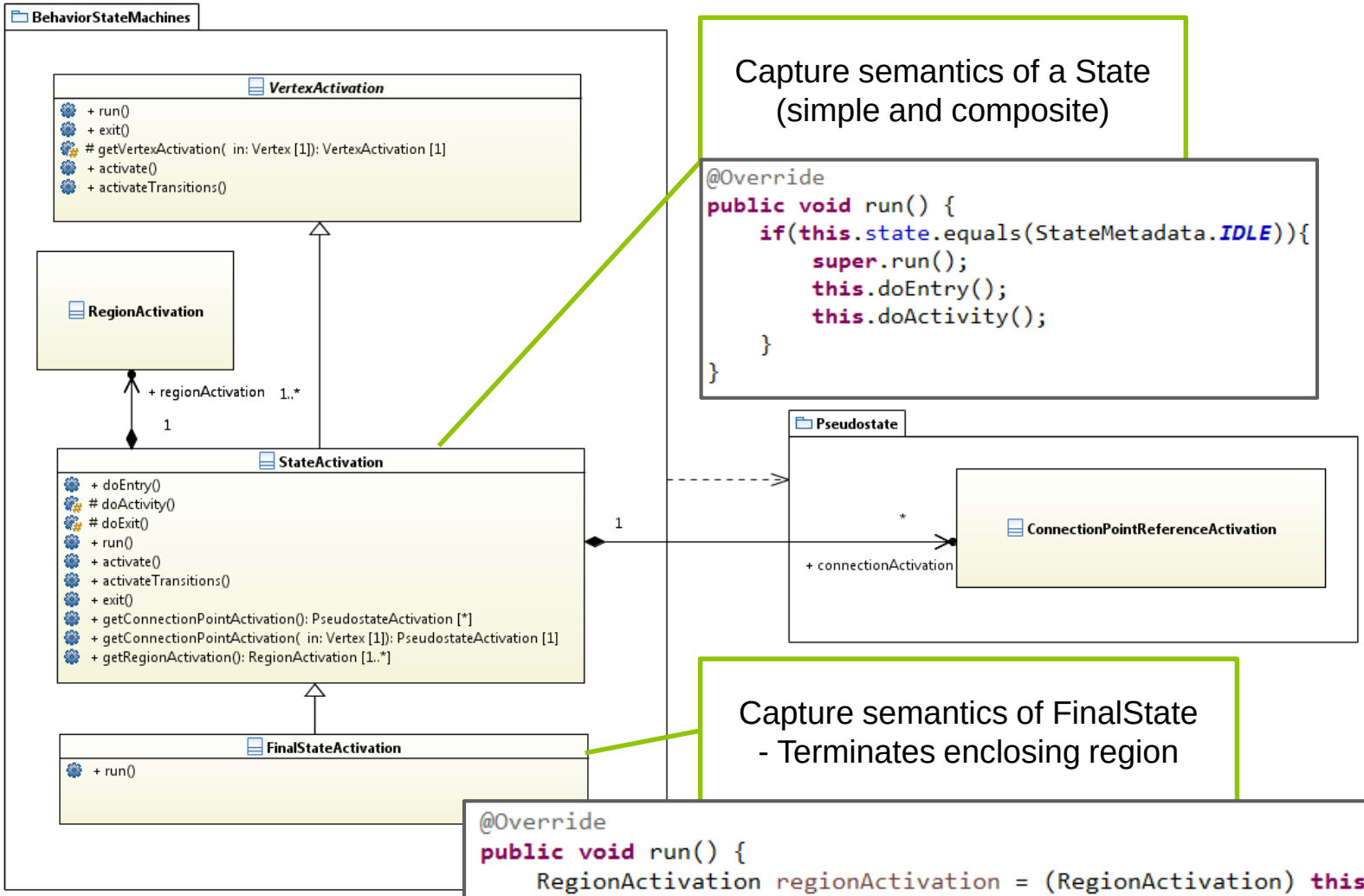
Start each regions of the state-machine. This should be done concurrently

Search transitions ready to fire. If none of them is ready then “execute” terminates.

BEHAVIOR STATE MACHINES [SPECIALIZED VISITORS]

SEMANTIC MODEL

STATE AND FINAL STATE



Capture semantics of a State (simple and composite)

```
@Override
public void run() {
    if(this.state.equals(StateMetadata.IDLE)){
        super.run();
        this.doEntry();
        this.doActivity();
    }
}
```

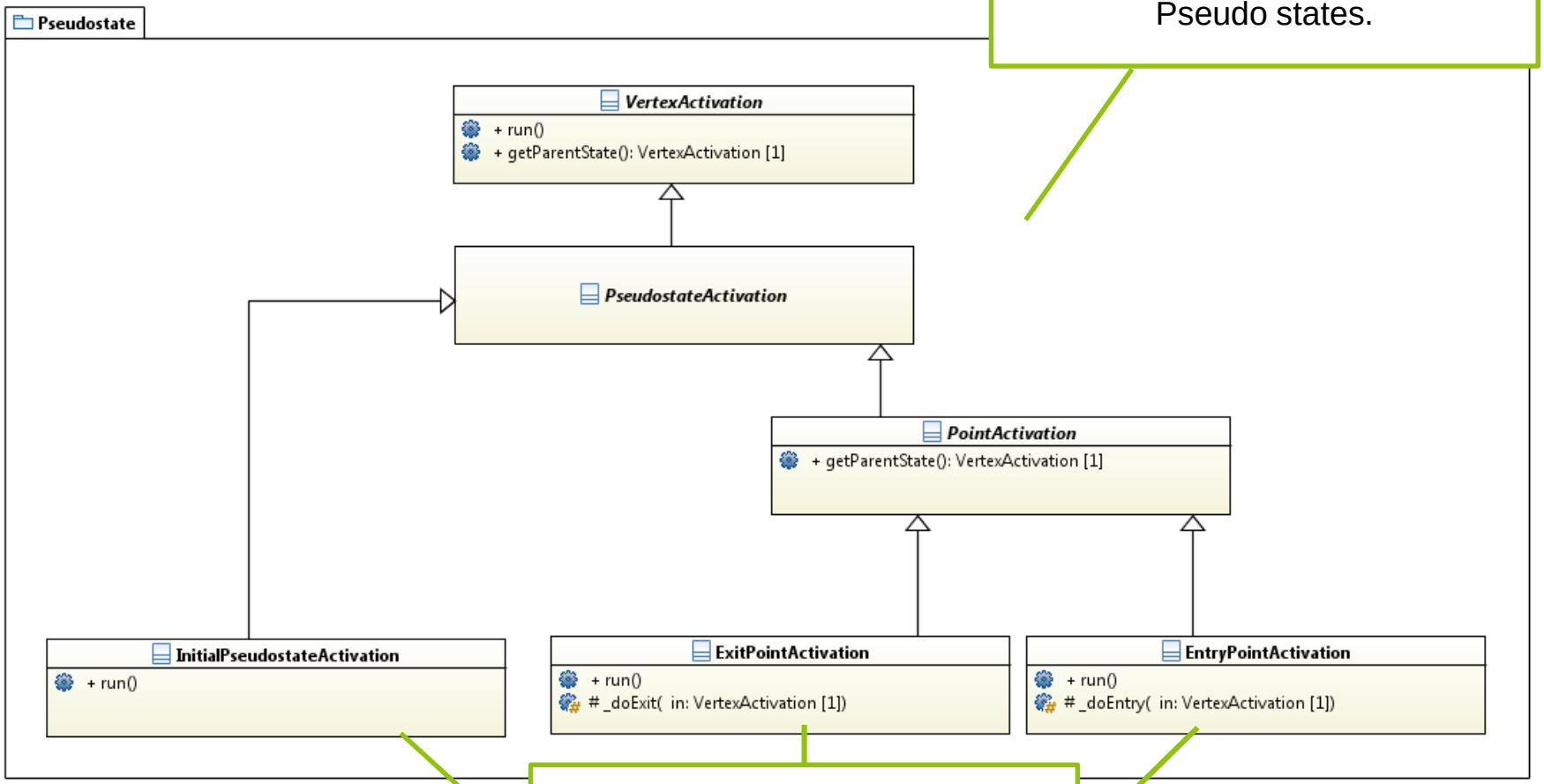
Capture semantics of FinalState - Terminates enclosing region

```
@Override
public void run() {
    RegionActivation regionActivation = (RegionActivation) this.getParent();
    regionActivation.exit();
}
```

SEMANTIC MODEL

PSEUDO STATES

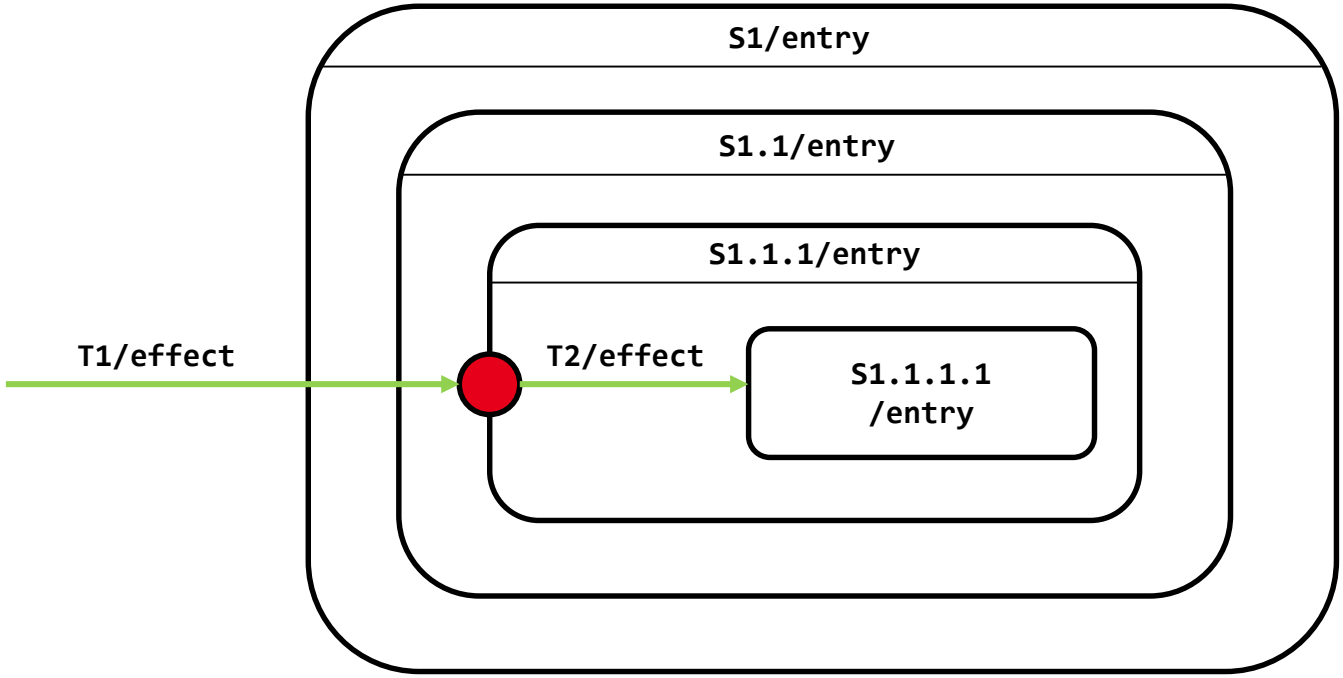
Newly introduced visitors for Pseudo states.



Semantic of each kind of pseudo state is encoded within a redefinition of the “run” operation

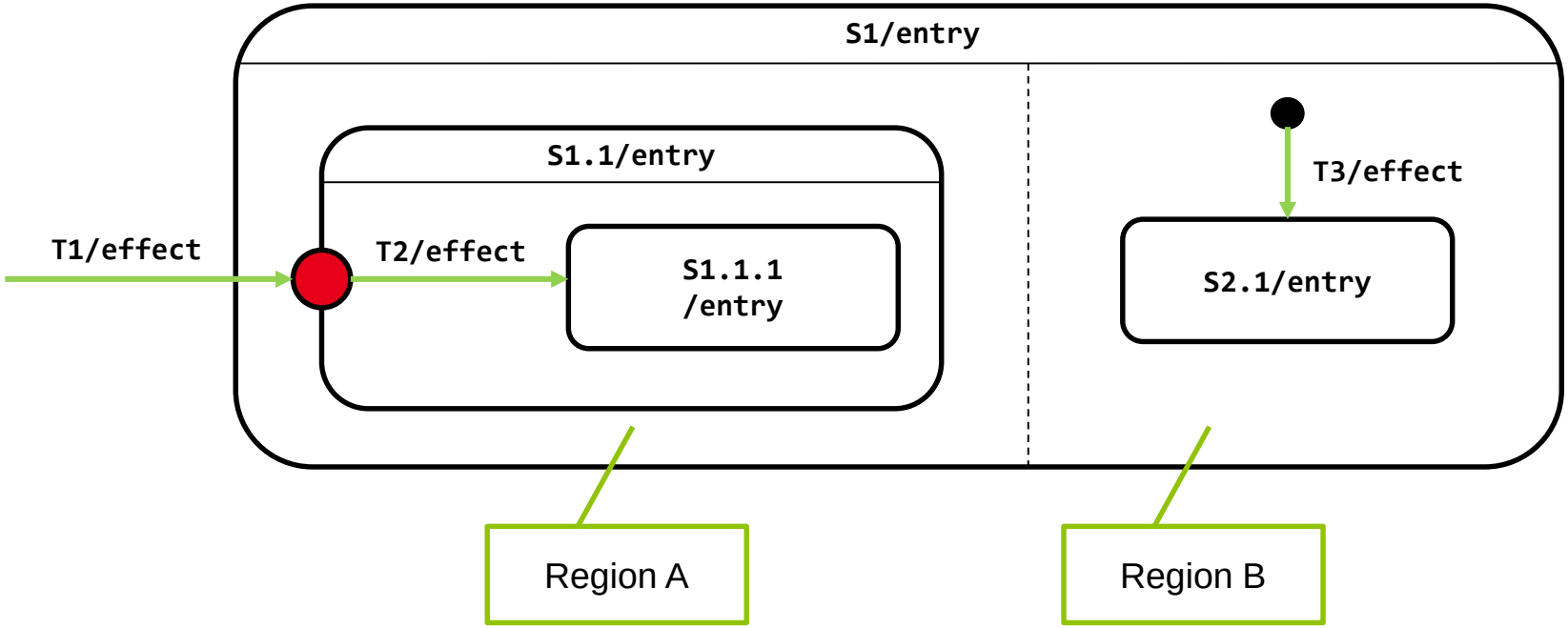
EntryPoint

- Execution sequence:
  - Effect of incoming transition
  - Entry of composite states on which the entry is placed. Its parent state, if any, is also entered
  - Effect of the outgoing transition
  - Entry of the state that is the target of the outgoing transition



T1(effect)::S1(entry)::S1.1(entry)::S1.1.1(entry)::T2(effect)::S1.1.1.1(entry)

EntryPoint and regions

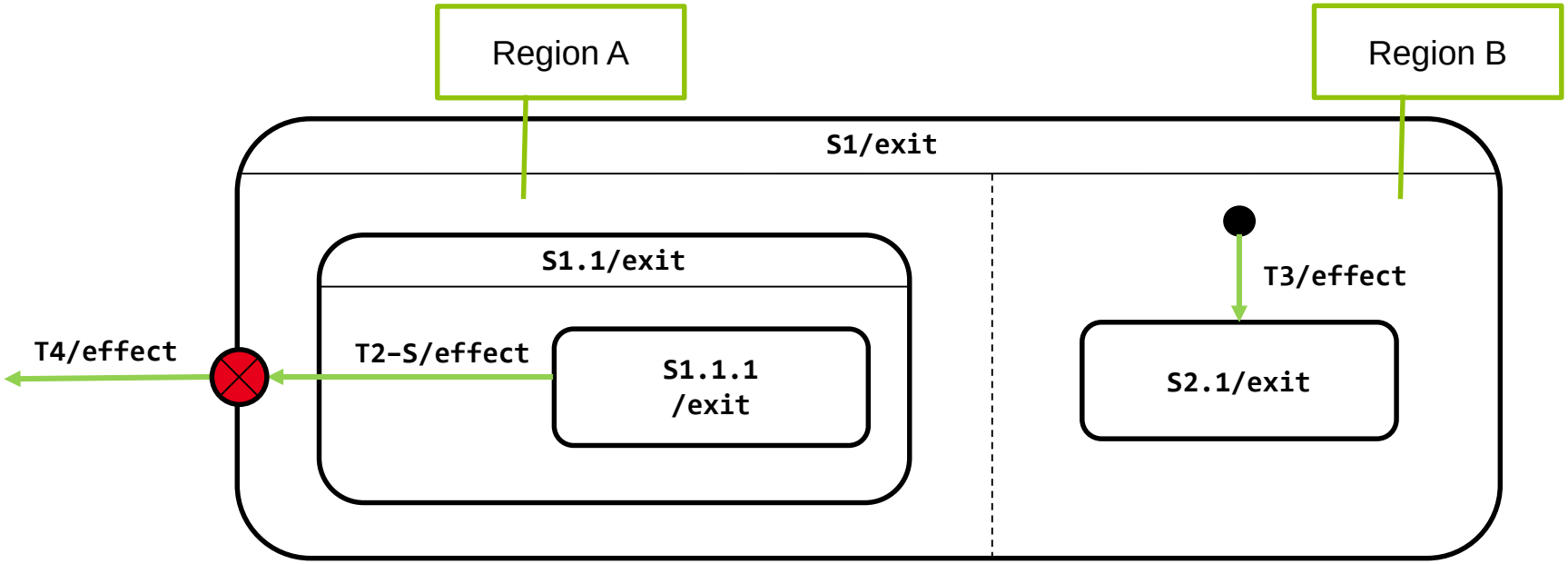


- Regarding UML 2.5 region must be entered
  - cf. entry point paragraph on p. 369
- To us the execution sequence should look like:

```
T1(effect)::S1(entry)::[S1.1(entry)::T2(effect)::S1.1.1(entry)|T3(effect)::S2.1(entry)]
```

Done in //

ExitPoint and regions



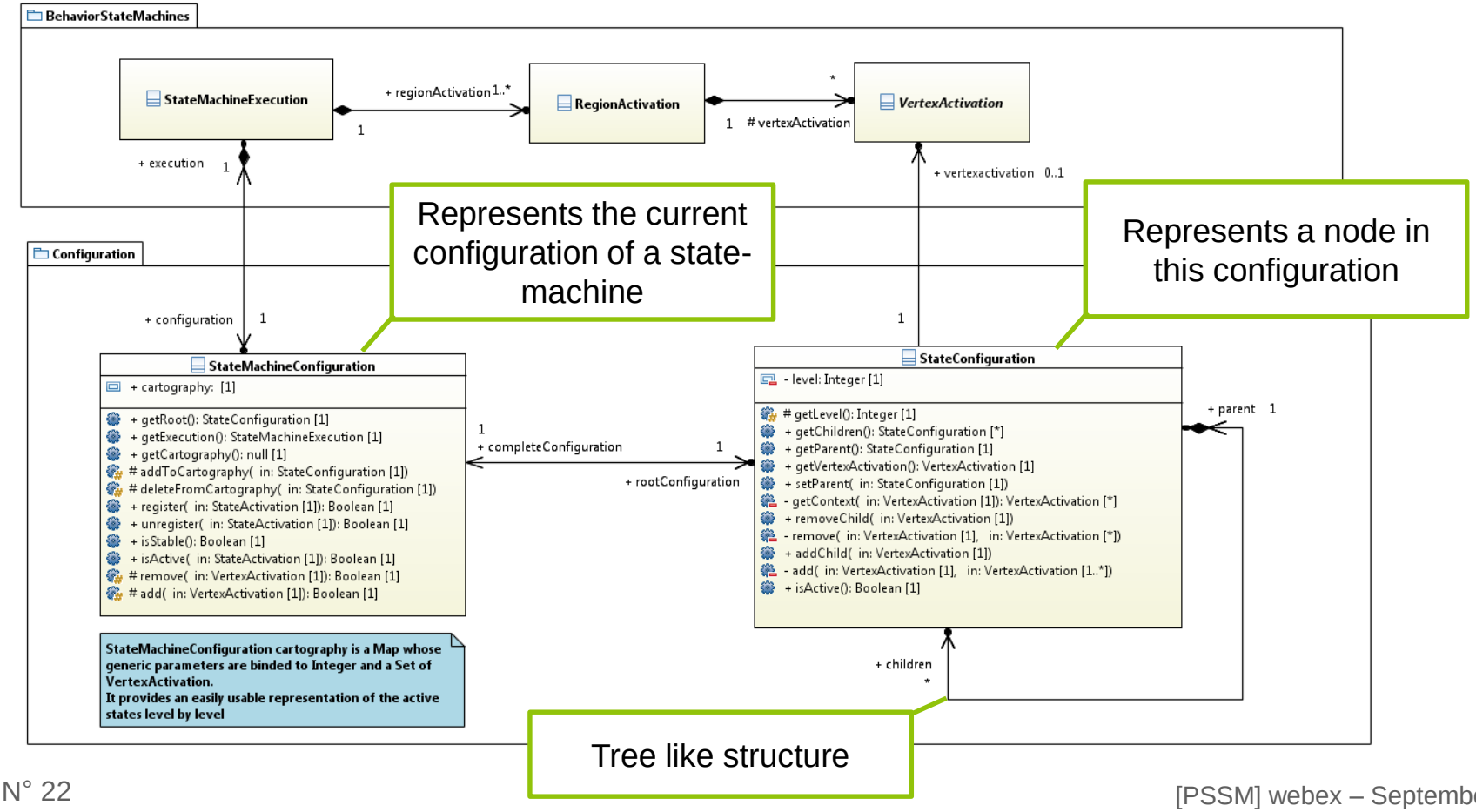
- Regarding UML 2.5, each region must be exited
  - cf. “Exiting state” section on page 369
- To us the execution sequence should look like
  - On reception of S:

```
S1.1.1(exit)::T2(effect)::S1.1(exit)::S2.1(exit)::S1(exit)::T4(effect)
```

STATE-MACHINES CONFIGURATIONS

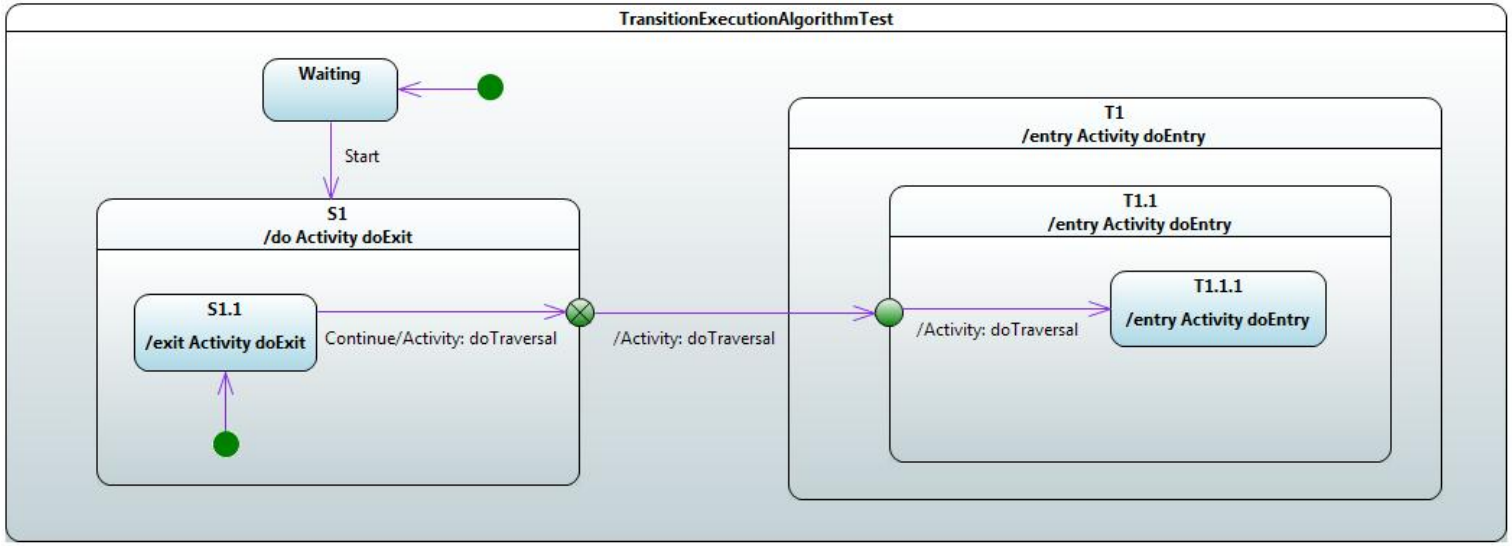
State-machine configuration

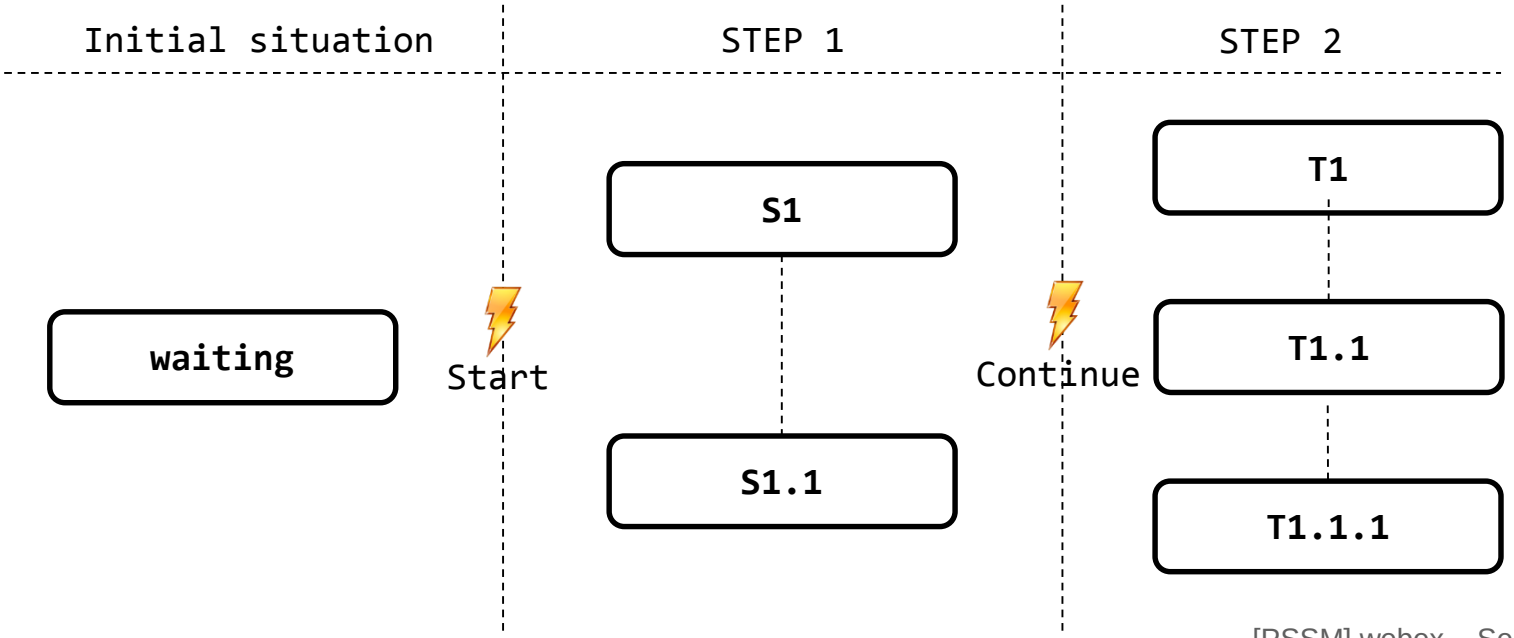
- Representation of the set of active states of an executed state-machine
- Built dynamically all along the state-machine execution
- Used to reason about available transition for firing
- Probably useful when dealing with History



SEMANTIC MODEL

STATE-MACHINE CONFIGURATION EXAMPLE

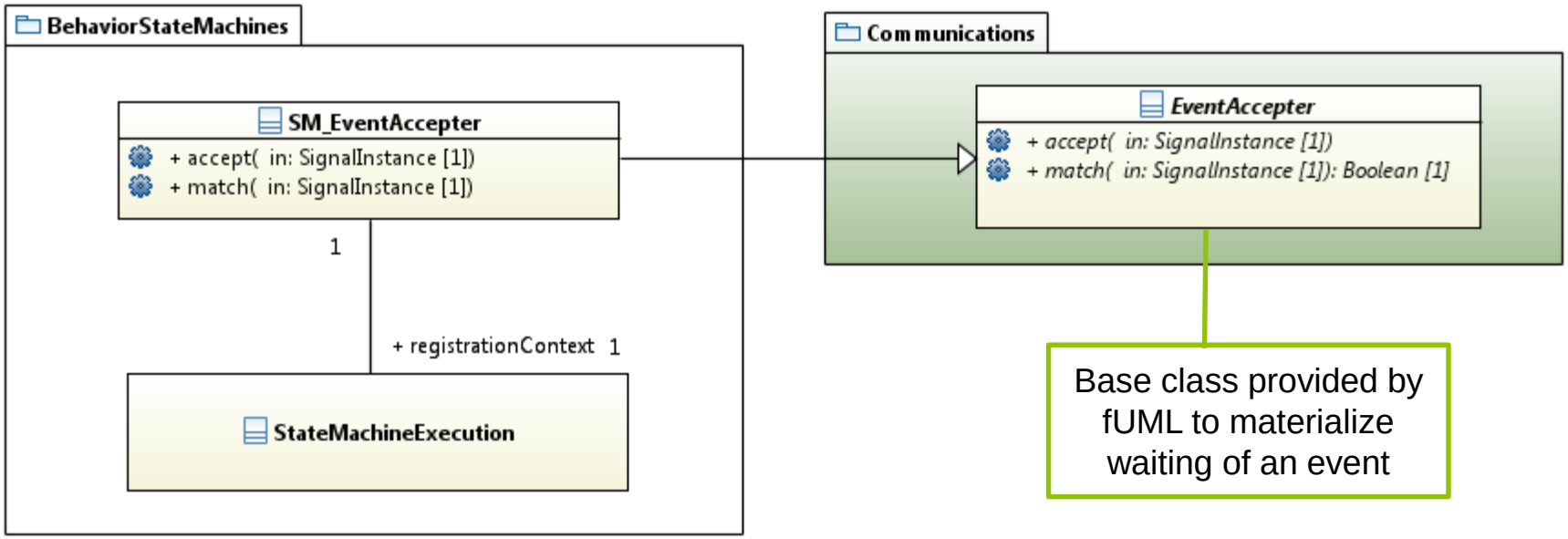




EVENT DISPATCHING AND RUN TO COMPLETION STEP

SEMANTIC MODEL

EVENT HANDLING



Accepter in the context of state-machines

- Used differently than the way it was done for Activities
  - In activities: an event accepter is generated when an “AcceptEventAction” is encountered. You can have many of them in the “waitingEventAccepter” list.
  - In the context of state-machine only one is needed per state-machine execution.
- “match”: evaluate the configuration to search for “fireable” transitions
- “accept”: fire a selected transition – propagate RTC step

Accept implementation

Transition selection based on the configuration attached to the state-machine execution

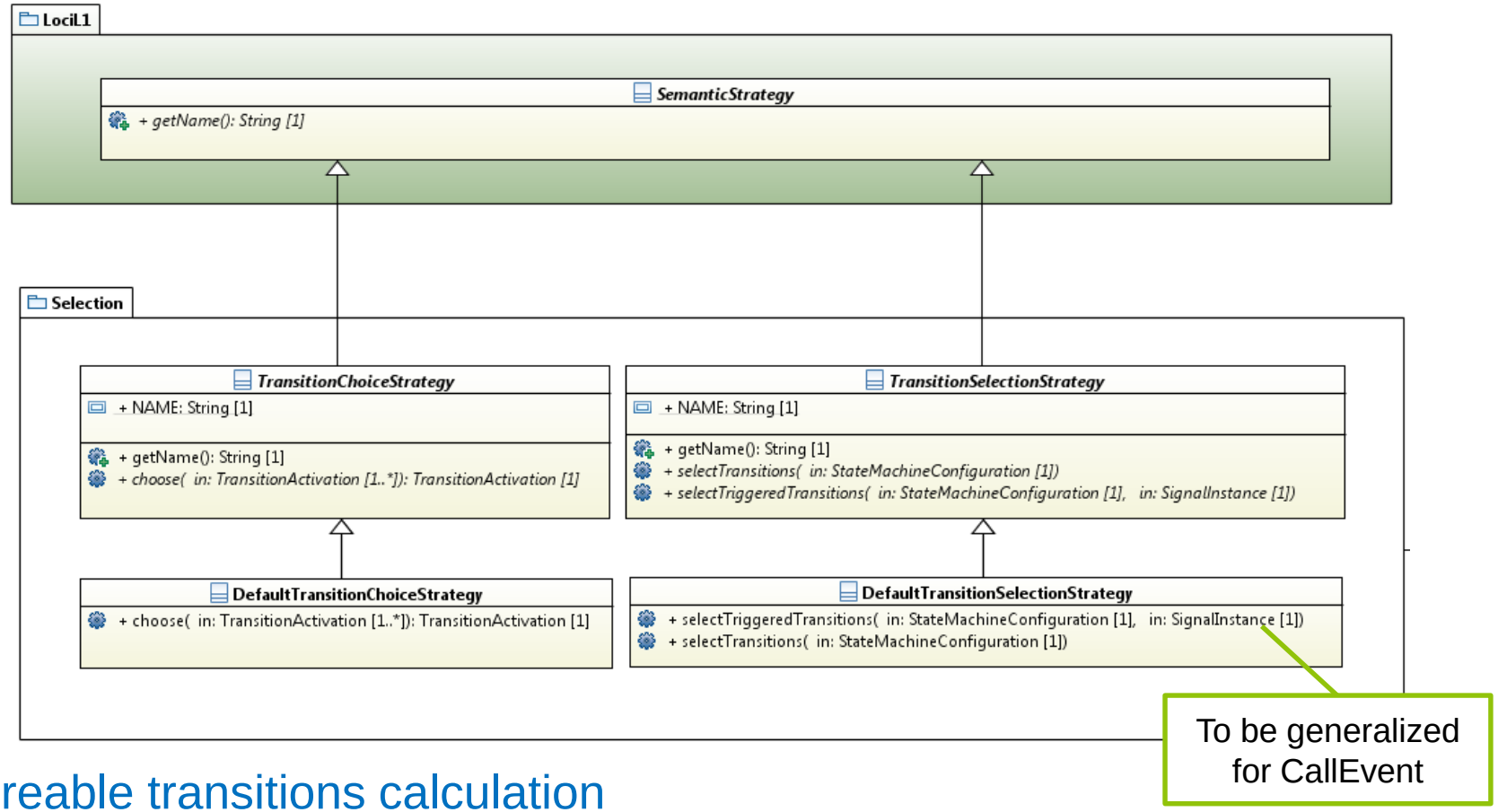
```
@Override
public void accept(SignalInstance signalInstance) {
    /*1. Select the transition that will fire according to priority rules*/
    TransitionSelectionStrategy selectionStrategy = (TransitionSelectionStrategy) this.registrationContext.locus.fact
    List<TransitionActivation> fireableTransition = selectionStrategy.selectTriggeredTransitions(((StateMachineExecut
    TransitionChoiceStrategy choiceStrategy = (TransitionChoiceStrategy)this.registrationContext.locus.factory.getStr
    if(!fireableTransition.isEmpty()){
        /*1.1. Fire the choosen transition */
        TransitionActivation transitionActivation = choiceStrategy.choose(fireableTransition);
        transitionActivation.fire();
        /*1.2. Continue to fire transitions (not triggered) while it is possible*/
        fireableTransition = selectionStrategy.selectTransitions(((StateMachineExecution)this.registrationContext).ge
        while(!fireableTransition.isEmpty()){
            choiceStrategy = (TransitionChoiceStrategy)this.registrationContext.locus.factory.getStrategy(TransitionC
            transitionActivation = choiceStrategy.choose(fireableTransition);
            transitionActivation.fire();
            fireableTransition = selectionStrategy.selectTransitions(((StateMachineExecution)this.registrationContext
        }
    }
    /*2. Register an event accepter for the executed state-machine*/
    Object_ context = this.registrationContext.context;
    if(context!=null && context.objectActivation!=null){
        context.register(new SM_EventAccepter((StateMachineExecution)this.registrationContext));
    }
}
```

Fires the selected transition

Analyzes the possibility to continue the execution step.

SEMANTIC MODEL

FIREABLE TRANSITIONS CALCULATIONS



Fireable transitions calculation

- Defined as a semantic strategy: “TransitionSelectionStrategy”
  - Analyze the configuration to calculate the set of fireable transitions
  - Respect by construction priority rules given by the state-machine hierarchy
- In case of transition conflict within the same region
  - The “DefaultTransitionChoiceStrategy” is provided

selectTriggeredTransitions implementation

```
@Override
public List<TransitionActivation> selectTriggeredTra                    figuration,
        SignalInstance signal) {
    List<TransitionActivation> fireableTransition = new ArrayList<TransitionActivation>();
    Map<Integer, Set<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(transitionActivation.isTriggered() &&
                        transitionActivation.hasTrigger(signal)){
                    fireableTransition.add(transitionActivation);
                }
            }
        }
        if(!fireableTransition.isEmpty()){
            nextLevel = false;
        }else{
            i--;
        }
    }
    return fireableTransition;
}
```

Iterates over the cartography starting from the innermost level of the hierarchy

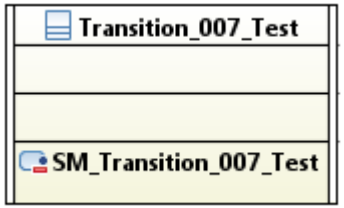
Register any transition outgoing the current state which as a trigger for the given signal

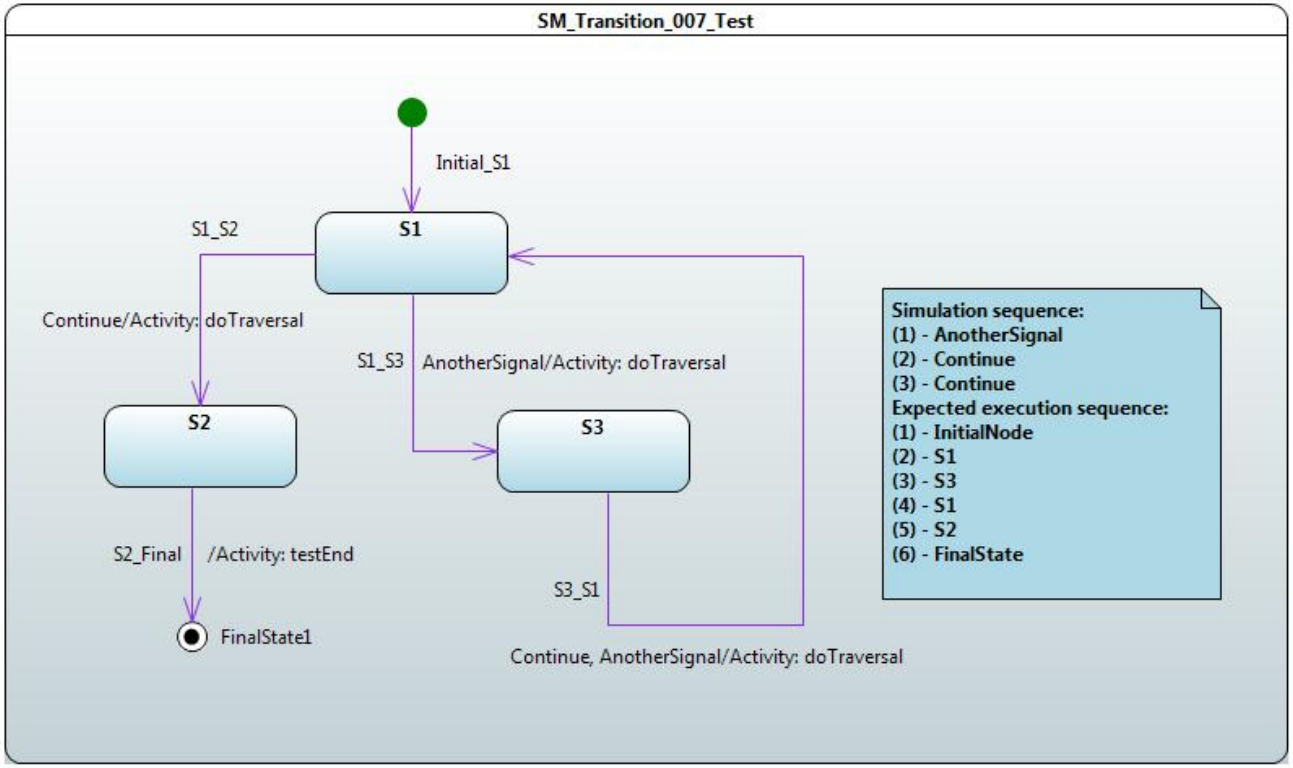
If fireable transitions are found at this level then stop the research

WANING: implementation uses a Map

What does characterize a step ?

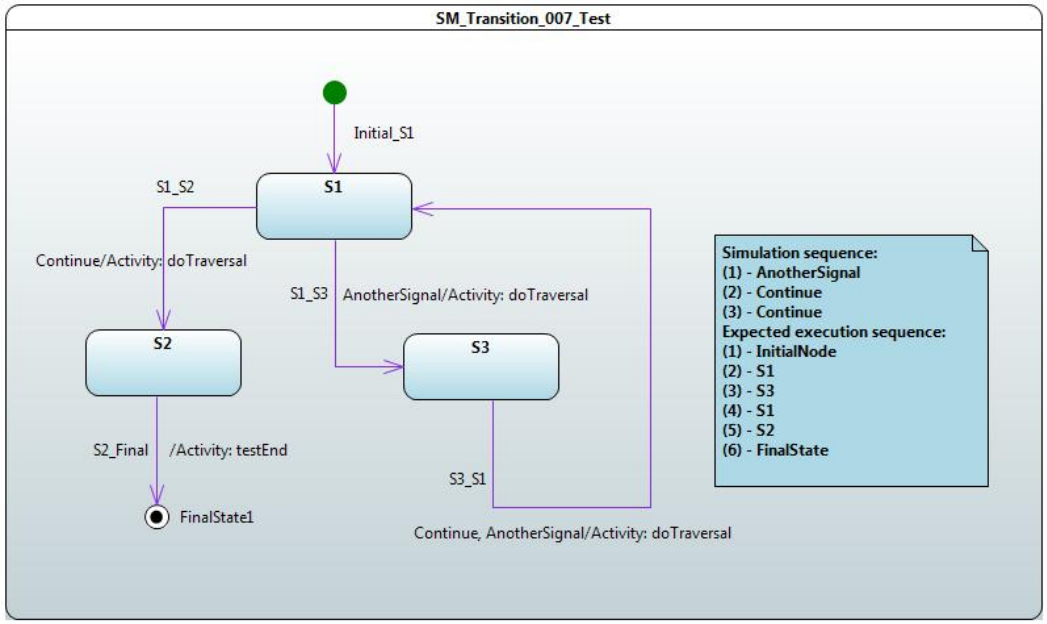
- A step is basically a sequence of “run()-fire()” operation calls
- The initial step of a state-machine is done in the “execute” operation
- Other are done in reaction to an event through the “accept” operation of an “SM_EventAccepter”





SEMANTIC MODEL

COMPLETE DYNAMIC OVER AN EXAMPLE



- [STEP1] => Initial_S1::S1(entry)
- Signal “AnotherSignal” received (TransitionSelectionStrategy is called)
- [STEP2] => S1_S3::S3(entry)
- Signal “Continue” received (TransitionSelectionStrategy is called)
- [STEP3] => S3_S1::S1(entry)
- Signal “Continue” received (TransitionSelectionStrategy is called)
- [STEP4] => S1_S2::S2(entry)::S2_Final => Termination of the state-machine execution

THE ELEMENTS THAT NEED TO BE SUPPORTED BY DECEMBER MEETING

WHAT SHOULD BE COVERED BY DECEMBER

UML State machines

Mandatory Requirements

- State
- Guarded Transition
- FinalState
- Triggered Transition (SignalEvent)
- State-machine as classifier behavior
- Completion Transition
- Triggered Transition (CallEvent)
- Orthogonal regions
- Local Transition
- doActivity of a State
- History, Fork, Join, Choice pseudostates
- Standalone State-machine

Needs to be introduced in fUML first

Except for Fork and Join

- Submachines (cf. section 6.6.1)
- Protocol State-machines (cf. section 6.6.2)
- State machines redefinition (cf. section 6.6.3)
- State-machine as method