Translation of UML Statecharts to AspectJ

Project Work

Submitted by:
Jesus Elias Sierra Paramo
elias.sierra@gmail.com
IMT
Matriculation Number: 31539

Supervised by:

Prof. Dr. Ralf Möller
STS - TUHH

M.Sc. Miguel GARCIA
STS - TUHH

Hamburg, Germany
2006-05-26

Declaration

I declare that:
this work has been prepared by myself, all literally or content-related quotations from other sources are clearly pointed out, and no other sources or aids than the ones that are declared are used.

Hamburg, 18.05.2005

Jesus Elias Sierra Paramo

Table of Contents

1. Introduction

Nowadays, Object-Oriented modeling has gained more importance in the first steps of the Software Development Process; having a conceptual design and with the help of existing tools can facilitate the code implementation and reduce time in this stage.
The most common modeling language used today is by far the UML (Unified Modeling Language). This provides standards for defining different model diagrams to help and improve the conceptual understanding of the application to be developed. The most commonly modeling diagrams used for representing a conceptual view of the application and its behavior are Class diagrams and Statecharts diagrams.
The Object Constraint Language (OCL) provides facilities to add constraints to the values that the objects can take during runtime, among other features. These constraints normally cannot be defined using only the UML class diagrams. OCL are used in this work to ensure a better performance during runtime.
Aspect Oriented Programming takes also an important role in the Design Phase. Complements the Object-Oriented programming, allowing modifying dynamically the static model by the so-called *weaving*. AOP defines (part of) the behavior of an application.
Executable UML is a relatively new tool in the development process of the Model Driven Architecture. It helps to generate an executable application using an abstract program model with a platform-specific model compiler.
This project is focused on the Statecharts involving the previous mentioned background. More details on these topics will be introduced later on.

1.1 Objectives

The intention of this project work at the beginning was to generate a specification to translate UML statechart diagrams to AspectJ starting from .uml and .ocl specifications defined in Octopus. Nevertheless, during the development of this work, another solution was found for specifying behavior in statecharts different than the intended AspectJ specification. This will be pointed out later on.
The intended approach is achieved by doing several iterations following the Incremental Software Development Process, during each of them introducing and exploring more advanced concepts of statechart diagrams.
Before starting the Iterations was necessary to get a background about Executable UML, Statecharts, AST's, Model Driven Architecture concepts, AOP (Aspect Oriented Programming), OCL (Object Constraint Languages) and the Octopus tool for code generation.
The intention of the iterations was to explore different techniques and find problems in the implementation of Statecharts and, when possible, suggest how these problems can be solved out by developing a statechart metamodel and generating instances with Octopus.
The selected technology to be used on this project work is Eclipse as IDE and AspectJ as AOP language for exploring behavior of statecharts.

1.2 Structure of the work

This project work is divided into several chapters. Chapters 2 to 6 provide a short and necessary background needed to understand the objectives and intentions of the related work; this background is mainly used on Chapters 7 and 8. Nevertheless in this work is assumed a previous knowledge from the reader in the topics covered from Chapter 2 to 6.

For more extensive information about this topics, please consult the sources and references at the end of this work.

Next, a small description about each chapter is given:

Chapter 1
Provides a brief introduction on the main topics used to develop this project work as well as the objectives.

Chapter 2
Gives a small introduction to Statecharts and a description of their main components and notations, as well as one example.

Chapter 3
Introduces the basic concepts of Aspect Oriented Programming.

Chapter 4
Provides a brief description of Executable UML (xUML) and Model Driven Architecture (MDA) and the relation between them.

Chapter 5
Gives a brief description of the Object Constraint Language and its principal constraints.

Chapter 6
Introduces the Octopus tool used with the Eclipse IDE and provides a brief description of the Code generation process.

Chapter 7
In this chapter is where the main work comes into action. Here the proposed metamodel for Statechart is shown, it generates various iterations of the Software Development Process and shows some examples generated by them.

Chapter 8
Here the last example from the previous chapter with Octopus is implemented, by using defined .uml and OCL expressions.

Chapter 9
Conclusion of the related work

2. Statecharts

2.1 Introduction

Statecharts are used to describe behavior of specified objects in a system. Each object can have different states in a program execution. It consists of discrete states and transitions. Each state represents different context of the behavior.
State machine modeling is the basis for various real-time methods, and according to Ben Meadowcroft[1], there are some benefits of Statecharts that overcome the limitations of State Machines by providing a construct known as the and-state, allowing the state chart to have substates of a higher level state active at the same time.

2.2 Statechart components

The components of a statechart diagram are[2]:

Finite abstract machine

A finite abstract machine (M) is an abstract model consisting of:

- A finite State set (*S*)
- A starting State (s_0)
- An input alphabet (S)
- A mapping function d=Sx*S* Æ *S*
- An input sequence acceptance function b= *S* Æ {0,1} such that *M*= (*S,* S, s_0, d, b)

A finite abstract machine is an abstract machine that defines a set of conditions of existence (called “states”), a set of behaviors or actions performed on each of those states, and a set of events that cause changes in states according to a finite and well-defined rule set.

Basic notation of Statecharts

Transitions

Indicate that the state machine responds to an event while in certain states.
Transitions affecting a superstate apply at all levels of nesting within that superstate.
Transitions are modeled taking approximately zero time to execute, as implied that by the statement that an object spends all of its time in states. If a transition can take a significant amount of time, then the object should be decomposed into more states so that eventually, the time taken to get from a predecessor state to a subsequent state is insignificant.
A null transition is a transition which is evaluated only once upon entrance to the source state. If it has no guard, or if the guard evaluates to TRUE, then the transition is taken immediately.

The general syntax for a transition is:
event-name ‘(‘ parameter-list’)’ ‘[‘ guard ‘]’ ‘/’ action-list ‘^’ event-list

Guards

A guard is a Boolean condition that returns a TRUE or FALSE value that controls whether or not a transition is taking following the receipt of a triggering event. A transition with a guard is only taken if the triggering event occurs and the guard evaluates to TRUE.

Pseudostates
Conditional pseudostates are a notational shorthand for multiple exiting transitions all triggered by the same event but each having different guards

Events
The UML defines 4 different kinds of events:
- Signal Event: an event due to some external asynchronous process
- Call Event: an event due to the execution of an operation within the object
- Change Event: an event due to the change in the value of an attribute
- Time Event: an event due to the lapse of an interval of time

Actions
Actions are small atomic behaviors executed at specified points in a state machine. They are assumed to take an insignificant amount of time to execute and are non-interruptible.
Actions are separated from the event-name and guard with slash ("/") and are always executed in a predefined order:
1. Exit actions of the source state(s)
2. Transition actions
3. Entry actions of the target state(s)

2.3 Statechart Diagram example

A basic statechart diagram is displayed in the next page:

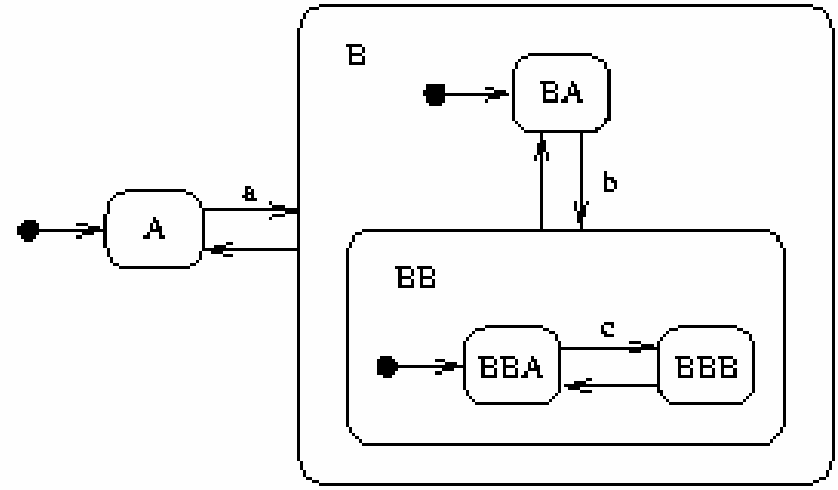

Figure 1: Basic statechart diagram

In the previous example, transitions are represented as arrows, states are represented as squares and the small black circles represent Pseudostates: initial and entry. We can observe, that the State B contains 2 Sub-states (BA and BB), while BB contains also 2 Sub-states (BBA and BBB). The initial state is the small black circle at the left that makes an automatic transition to state A. with the event A is possible to go to state B and then with the entry Pseudo state drives you to state BA. The same happens with event b which leads you to state BB and then the Sub-state BBA is reached.

3. Aspect Oriented Programming

3.1 Introduction

AOP is a new methodology that allows the separation of crosscutting concerns in applications by introducing the so called “aspects”.
According to James Holmes from Oracle[3], AOP provides a solution for abstracting cross-cutting code that spans object hierarchies without functional relevance to the code it spans. Instead of embedding cross-cutting code in classes, AOP allows you to abstract the cross-cutting code into a separate module (known as an aspect) and then apply the code dynamically where it is needed. You achieve dynamic application of the cross-cutting code by defining specific places (known as pointcuts) in your object model where cross-cutting code should be applied. At runtime or compile time, depending on your AOP framework, cross-cutting code is injected at the specified pointcuts. Essentially, AOP allows you to introduce new functionality into objects without the objects' needing to have any knowledge of that introduction.

3.1.1 Terminology

AOP is composed mainly by the following concepts:

Join points
Specifies well-defined points in the execution of the program where an aspect can apply. Includes method execution, instantiation of an object and the throwing of an exception. These join points can only be identified at runtime.

Advice
Is a way of affecting behavior at the specified join points; executes the code that is applied to a specified pointcut, or cross-cuts, your existing object model. Can be executed before, after or around the join point. Advice code is what modifies the behavior or properties of an existing object. Advice is also commonly referred to as introductions or mix-ins.

Pointcuts
These define the points in your model where advice will be applied. Pointcuts can collect the context by selecting join points.

Aspects
These encapsulate advice and pointcuts into functional units in much the same way that OOP uses classes to package fields and methods into cohesive units. For example, you might have a logging aspect that contains advice and pointcuts for applying logging code to all setter and getter methods on objects.

3.1.2 Existing Frameworks and Tools

In order to start implementing AOP; it is necessary to get a framework to work with. Until now, neither JAVA nor any other O-O programming language provides built-in support for AOP. There are differences within the existing Frameworks that need to be considered before choosing one. One difference is the way aspects are defined and

applied; aspects could be defined and applied with code whereas with other frameworks could be defined in a XML configuration file. Another difference is that some frameworks use bytecode manipulation to tie into an object model, and others use proxy-based systems.

Currently, there exist several AOP frameworks, some of them providing integration with Eclipse. For the development of this work, AspectJ for Eclipse was selected for AOP. Following are some of the existing frameworks:

Name	**Language support**
Spring	Java (Platform independent)
AspectJ	Java Eclipse plug in
AspectJ browser	Java
Encase	C# (.NET Platform)
Nanning [4]	Java
JBoss [5]	Java
Aspectwerkz 2 [6]	Java

Table 1 : AOP Frameworks

3.2 AspectJ

AspectJ for Eclipse[7] is a seamless aspect-oriented extension to the Java programming language; enables a clean modularization of crosscutting concerns, such as error checking and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support, and multi-object protocols.

In the development of this project, several implementations using aspects in Statecharts were made for controlling the statechart behavior. Nevertheless, the same behavior-controlling was reached by implementing this control in one method of the metamodel I developed.
This migration of behavior-controlling will be pointed out in the chapter 7 and explained with some examples.

4. Executable UML

4.1 xUML Introduction

According to Stephen J. Mellor and Marc J. Balcer[8], Executable UML is a major innovation in the field of software development. It is designed to produce a comprehensive and understandable model of a solution independent of the organization of the software implementation. It is a highly abstract thinking tool that aids in the formalization of knowledge, and is also a way of describing the concepts that make up abstract solutions to software development problems.
As a foundation for Model Driven Architecture, Executable UML provides the key technology for expressing application domains in a platform-independent manner.

4.2 Model Driven Architecture

4.2.1 Introduction

MDA is an architecture defined by the OMG[9] for the Software Development; provides a new way for writing specifications in a platform-independent way by raising the level of abstraction. A complete MDA specification consists of a definitive platform-independent base UML model, plus one or more platform-specific models and interface definition sets, each describing how the base model is implemented on a different middleware platform.

4.2.2 MDA Components

Following are the components of the MDA, taken from MDA Explained[10]:

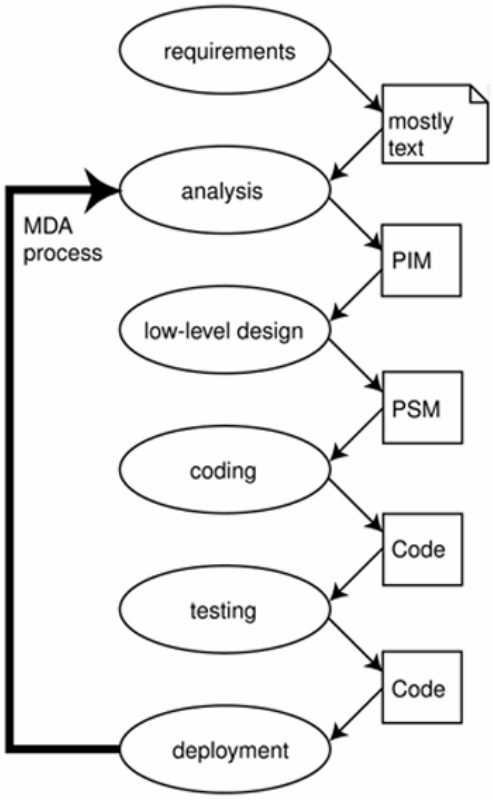

Figure 2: Model Driven Architecture components

Platform Independent Model

The first model that MDA defines is a model with a high level of abstraction that is independent of any implementation technology. This is called a Platform Independent Model (PIM).

Platform Specific Model

In the next step, the PIM is transformed into one or more Platform Specific Models (PSMs). A PSM is tailored to specify your system in terms of the implementation constructs that are available in one specific implementation technology.

A PIM is transformed into one or more PSMs. For each specific technology platform a separate PSM is generated. Most of the systems today span several technologies, therefore it is common to have many PSMs with one PIM.

Code

The final step in the development is the transformation of each PSM to code. Because a PSM fits its technology rather closely, this transformation is relatively straightforward.

The MDA defines the PIM, PSM, and code, and also defines how these relate to each other. A PIM should be created, then transformed into one or more PSMs, which then are transformed into code. The most complex step in the MDA development process is the one in which a PIM is transformed into one or more PSMs.

5. OCL

5.1 OCL Introduction

There is a need to describe additional constraints about the objects in the model that cannot be described in UML models. *OCL* is used to specify these additional constraints.
According to Jos Warmer and Anneke Kleppe[11], the Object Constraint Language (OCL) is a language that enables one to describe expressions and constraints on object-oriented models and other object modeling artifacts. An *expression* is an indication or specification of a value. A *constraint* is a restriction on one or more values of (part of) an object-oriented model or system.
In combination with UML, allows developing more effective, consistent, and coherent models that are critical to working with MDA.

5.2 OCL Components

There are four types of constraints (from [12]):

- An *invariant* is a constraint that states a condition that must always be met by all instances of the class, type, or interface. An invariant is described using an expression that evaluates to true if the invariant is met. Invariants must be true all the time.
- A *precondition* to an operation is a restriction that must be true at the moment that the operation is going to be executed. The obligations are specified by postconditions.
- A *postcondition* to an operation is a restriction that must be true at the moment that the operation has just ended its execution.
- A *guard* is a constraint that must be true before a state transition fires.

The *context definition* of an OCL expression specifies the model entity for which the OCL expression is defined. Usually this is a class, interface, datatype, or component. In terms of the UML standard, this is called a *Classifier*.

6. Octopus

6.1 Octopus Introduction

Octopus [13] is an Eclipse Tool that conforms to version 2.0 of the OCL standard; provides two main functionalities:

1. Statically check OCL expressions. It checks the syntax, as well as the expression types, and the correct use of model elements like association roles and attributes.
2. Transform the UML model, including the OCL expressions, into Java code.

6.2 Octopus code generation

Next, the process of code Generation in Octopus is briefly described.

The process of Octopus code generation encompasses three important things: the reading, processing and code generation of the Octopus models (.uml files) , the reading, processing and code generation of the OCL expressions (.ocl files), and the merge within them.

In order to analyze the code generation in Octopus, a second instance of Eclipse using the Octopus that generates generics is debugged. I made this second instance by using the corresponding Octopus and other plug-ins as show below:

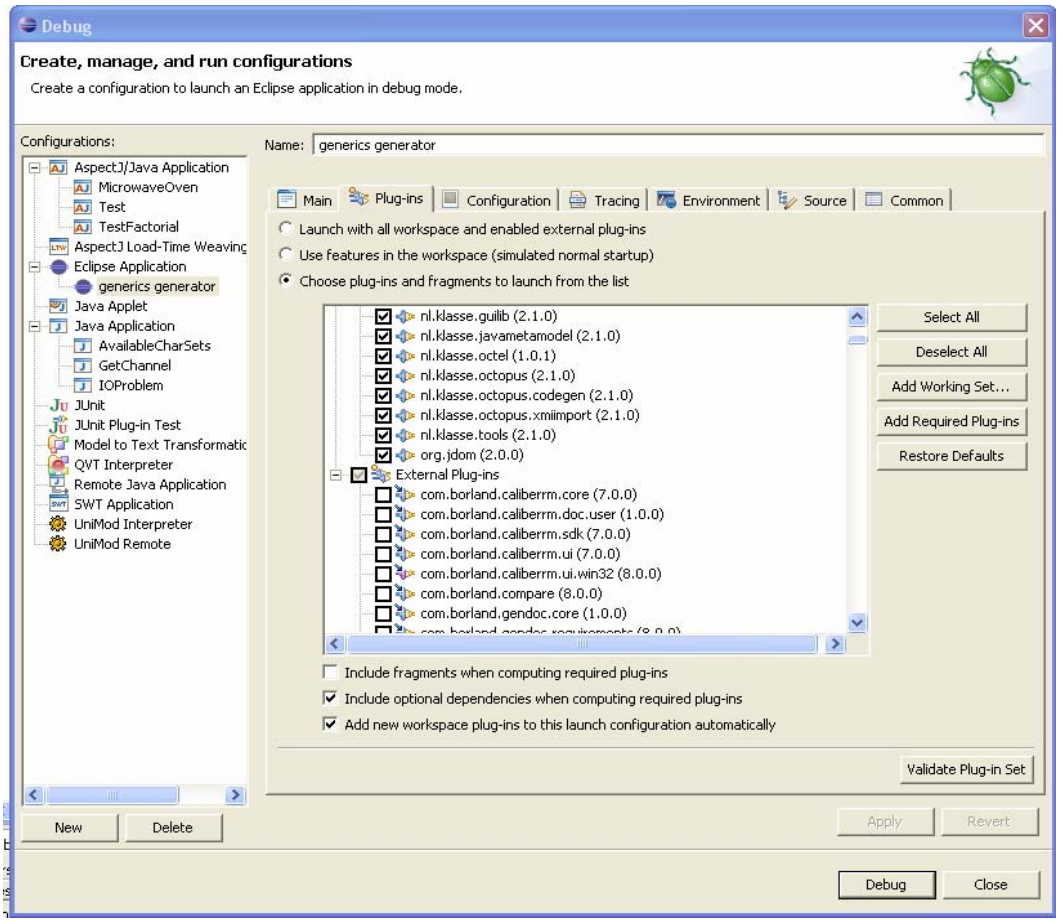

Figure 3: Selection of plug-ins for debugging in Octopus

After creating my second instance I created there a new Java project, added the Octopus nature and created one .uml and one .ocl file. Then I ran the Generate Java Code to debug in the original Eclipse instance.

The process of generating java code, is mainly done in the class *TransformationController* with the method *generate.* The instance of *TransformationController* is created in the class *GenerateCodeAction* and then it s method *generate* is called. This method takes as input an instance of *IPackage* and *IProgressMonitor.*

The first step is to split the interfaces using the method *generateInterfaceSplit*, which reads the .uml model and returns a *IPackage* instance, in this case a *PackageImpl* instance called "Example", which is the package I specified in my .uml file.

The second step generates a Java model with an instance of the class *OJPackage* using the method *generateMiddleTier()*. This method performs the following actions:

1. Transform the *IPackage* to *OJPackage* using the method *transform()* from an instance of the class *ModellController*.
2. Validates the *OJPackage* java model to be well formed and get the errors and warnings.
3. Generate the OCL expressions using the method generateExpressions(). This method creates an instance of the class *ExpressionController* and executes the method transform. The latter method performs several important task that are described next:
    a. Check if the transformation is possible using a visitor.
    b. An *OclUtilityCreator* instance is created and its method *makeOclUtilities* is called which create the Tuple types, standard library and the invariant helper classes. This must be done before any other OCL processing.
    c. The OCL operations definitions are added by creating an instance of the class *DefOperationGenerator* and using a visitor in the class *IPackage.*
    d. The OCL attributes definitions are added by creating an instance of the class *DefAttributeGenerator* and using a visitor. This is very similar to the *DefOperationGenerator*.
    e. Add OCL initial expressions by creating an instance of the class *AttrExpressionGenerator* and the correspoinding visitor
    f. An instance of *OperExpressionGenerator* is created with its corresponding visitor to add the OCL expressions attached to the operations
    g. The invariant operations are added by calling a visitor of the instance *InvariantGenerator*.
    h. Finally, the *OverwritesGenerator* instance add other features by calling a visitor.
4. The *generateMultChecks* method is executed, which in turn creates an instance of the class MultCheckGenerator and its visitor. This method performs the multiplicity check.
5. Generate extra operations: *toString()*, *getIdentifyingString()*, *getCopy()* and *allInstances()*. This methods are added to every generated class by calling the method *generateExtraOpers().*
6. Finally, the *generateVisitor()* method is called to create the Visitor interface to visit all elements.

The third step generates the storage tier calling the method *generateStore()*. This creates an XML file if was selected in the Octopus codegen properties.

The fourth step encompass the generations of User Interface if the option is selected in the Octopus codegen properties. This is done by calling the method *generateUI().*

The fifth step is to split the generated classes in the middle tier by calling the method *generateGenSplit()*. This method creates an instance of the class *GenerationSpliter* which in turn executes its method *transformModel(OJPackage).* This method generates the classes defined by the user.

Finally, the sixth step is to write the generated code in files by creating an instance of the class *FileGenerator* and executing its method *write()*

7. Statecharts Generation

7.1 Statechart metamodel definition

The first step for implementing statecharts I made was to develop a metamodel.
After several iterations implementing statecharts, to the following metamodel was achieved at the last of these iterations.

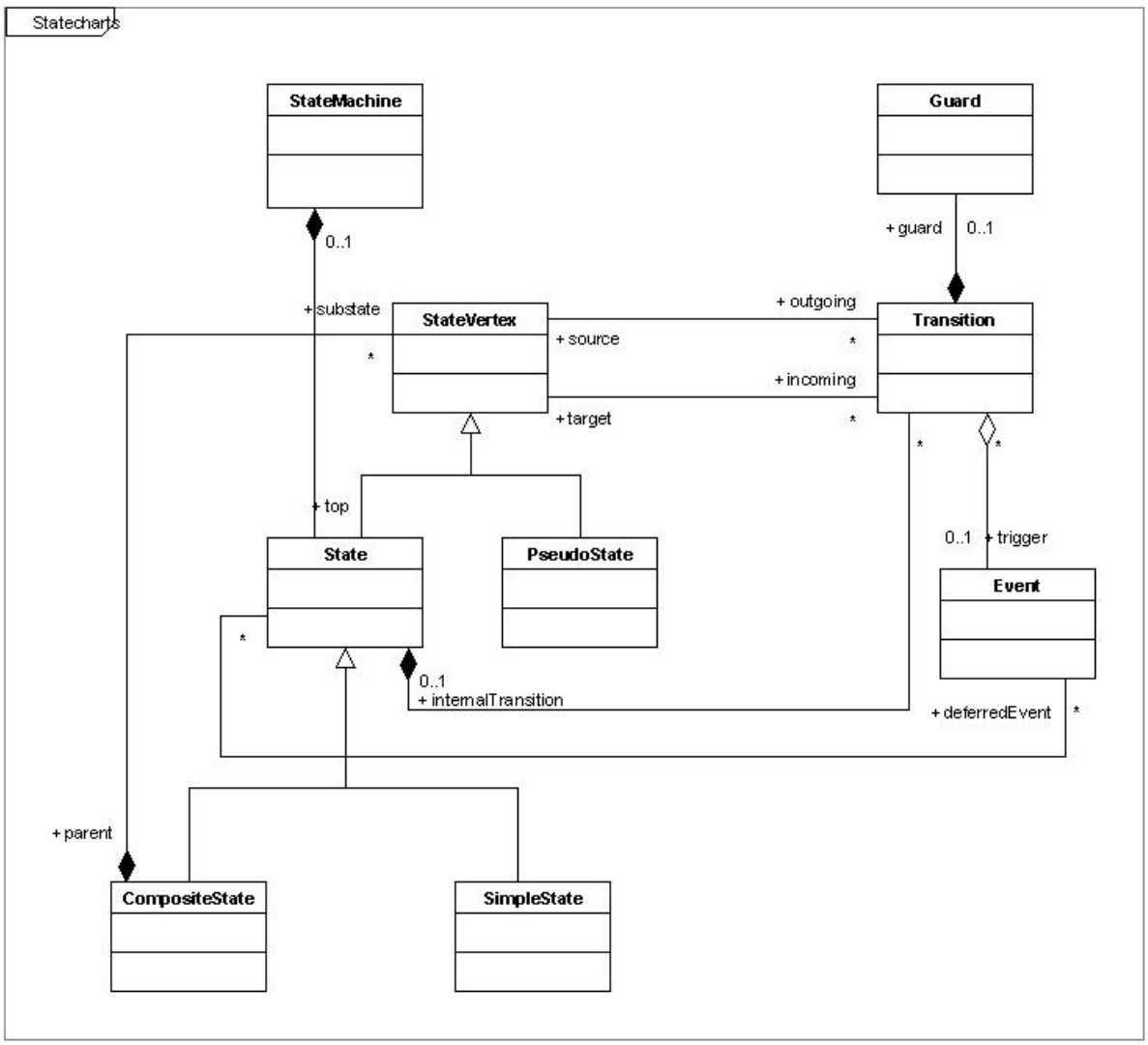

Figure 4: Statechart metamodel

Next I describe each of the classes of this metamodel and the role they play.

StateMachine.java.
Has a top state, to which everything belongs. Acts as container for the states and transitions.

StateVertex.java
Is an abstract class. Contains a name; a set of transitions, each of them is triggered by only one event. Also contains an outgoing and incoming transition, but they are not implemented in this example, also has a parent of the type *CompositeState*.

State.java
Inherits from StateVertex. Contains internal transitions and a set of events.

CompositeState.java

Inherits from State; contains a set of sub-states of the type StateVertex.

SimpleState.java

Inherits from State and represent a simple state; doesn't have sub-states.

Guard.java

Boolean expression to restrict a transition to be executed.

Event.java

Event that triggers a transition. Contains a set of States and a set of Transitions. One event can trigger different transitions depending on the current State. Also contains a method called *addTransition* that add the inherited transitions from its parent, this will be explained later.

Transition.java

Specifies transitions between states. Contains a source State, target State, trigger Event and a Guard.

7.2 Prototype specification

After the metamodel definition, I explain how my proposed prototype can be implemented.
Having the metamodel pre-defined classes, it is necessary to create a new class instancing the state machine and creating the corresponding states, events and transitions.
The first step is to define a Statemachine:

```
// statemachine declaration
public StateMachine statemachine1=null;
```

The second step is to define the existing states, including composite States, simple States and Pseudostates. A declaration of them is expressed as following respectively:

```
// States declaration
public static final StateVertex SIMPLESTATE_1 = new
SimpleState("SIMPLESTATE_1 ");
public static final StateVertex COMPOSITESTATE_1 = new CompositeState
("COMPOSITESTATE_1");
public static final StateVertex PSEUDOSTATE_1 = new PseudoState
("PSEUDOSTATE_1 ", "INITIAL");
```

Then, the events and transitions must be declared.

```
// Event declaration
public Event event1 = new Event("event1");

//transition declaration
public Transition t1 = new Transition(SOURCESTATE, TARGETSTATE, event1);
```

The guards also must be declared

```
// Guard declaration
Public Guard guard1 = null;
```

Next, in the constructor, the statemachine initialization is created. In the final implementation of the metamodel, the sub-states for the composite states are added automatically in the *CompositeState* constructor and the parents are respectively added in the *StateVertex* constructor. The transitions are also added to their corresponding trigger events in the *Transition* constructor. The same way, the corresponding guards of a transition are added in the Guard constructor.

```
// Initializes the statemachine
statemachine1= new StateMachine (new State (“statemachine1”));
```

In an earlier version of the metamodel implementation, it was necessary to manually aggregate the substates, the transitions triggered by events as well as the guards to transitions. These were specified as follows:

```
// Assign substates
COMPOSITESTATE1.addSubState(COMPOSITESTATE2);
COMPOSITESTATE1.addSubState(COMPOSITESTATE3);

// Add the transitions generated by Events
Event1.addTransition(t1);
Event1.addTransition(t2);

// Add the guards to the corresponding transitions
t1.addGuard(guard1);
```

Now it is necessary to define the *executeEvent()* and *doTransition()* methods, which are always the same. In the first one is where the behavior-controlling was migrated from aspects to this method. Whenever an Event occurs, it looks if the invoked event exists for the current state; if so, the transition takes place, otherwise raises a *ProtocolViolationException* and there is where the behavior-controlling exists. These methods would look like the following:

```
public void executeEvent(Event e) throws ProtocolViolationException{
      Iterator it=e.transitions.iterator();
      while (it.hasNext()){
            Transition t= (Transition)it.next();
            if (t.source==currentState){
                  doTransition(t);
                  return;
            }
      }
            throw new ProtocolViolationException();
}

public void doTransition(Transition transition) throws
ProtocolViolationException {
      if (executeGuard(transition.guard)||transition.guard==null){
            onExit(transition.source); //executes onExit action for source
                                       //State in transition
            currentState=transition.target;
            onEntry(transition.target); //executes onEntry action for
                                        //target State in transition
      }
}
```

Having this done, it is necessary to implement the *executeGuard(), onEntry()* and *onExit()* methods. Here is where all the actions definitions rely. The corresponding actions followed by the *onEntry()* and *onExit()* are precisely user-defined methods.
The way this is implemented for identifying the current state is done by if-else statements. Next, the *onEntry()* actions are shown would look like:

```
public void onEntry(StateVertex state) throws ProtocolViolationException{
      if (state==STATE1){
            System.out.println("onEntry Action from TrafficLight."+
            state.getName());
            state1Action(state);
      }
```

```
        else if (state==STATE2){
              System.out.println("onEntry Action from TrafficLight."+
              state.getName());
              state2Action(state);
        }
        else if (state==STATE3){
              System.out.println("onEntry Action from TrafficLight."+
              state.getName());
              state3Action(state);
        }

}
```

In the previous code, the methods *state1Action()*, *state2Actions()* and *state3Action()* are user-defined to specify the actions to execute (if any) when the entry to a new state is performed.
The same applies for the *onExit()* actions.

The *executeGuard()* method will looks something like the following:

```
public void executeGuard(Guard guard){
      boolean result=false;
      if (guard==guard1){
            // some conditional evaluation
      }
      else if (guard==guard2){
            // some conditional evaluation
      }
      return result;
}
```

After all of this is defined, it is necessary to create an entry point to the application; this is done by creating a main method in the implementation, creating an instance and calling the *executeEvent* method to execute the first and subsequent events.

7.3 Generating Statecharts based on prototype

In order to implement statecharts, it was necessary to make several iterations of the SDP, on each of them I instantiated the metamodel implementing an example.
The following examples go from the implementation of simpler statechart metamodel to a more complex one, including each time more components like sub-states, history states, guards and so on.

7.3.1 Bahnübergang example

After analyzing some example results of simple Statecharts omitted in this report, I moved towards a more complex example beginning the next iteration. This time I implemented a Bahnübergang statechart example.
This example and statechart diagrams are taken from[14] and consists of four different statecharts. Here are introduced and implemented new elements of the statecharts: composite states, initial Pseudo-states, when clause and concurrent statemachines.
The components of this example are:

Traffic Light

Has four states, as depicted in the diagram below; and four events. The top states are *activated* and *deactivated*; and the states *on* and *off* are sub-states of *activated*. The initial state of the traffic light is *deactivated*, when the *activate* event occurs, then moves toward

the state *activate*, which is a composite state. Once has entered to the *activate* state, then moves automatically to the *off* state, as depicted in the black circle representing a *Pseudostate* of initial state. When the event *switchOn* occurs, then a transition is triggered to the state *on* and the *switchLightOn* action is executed. The opposite happens when the event *switchOff* is triggered.
The statechart representing the traffic light is shown below:

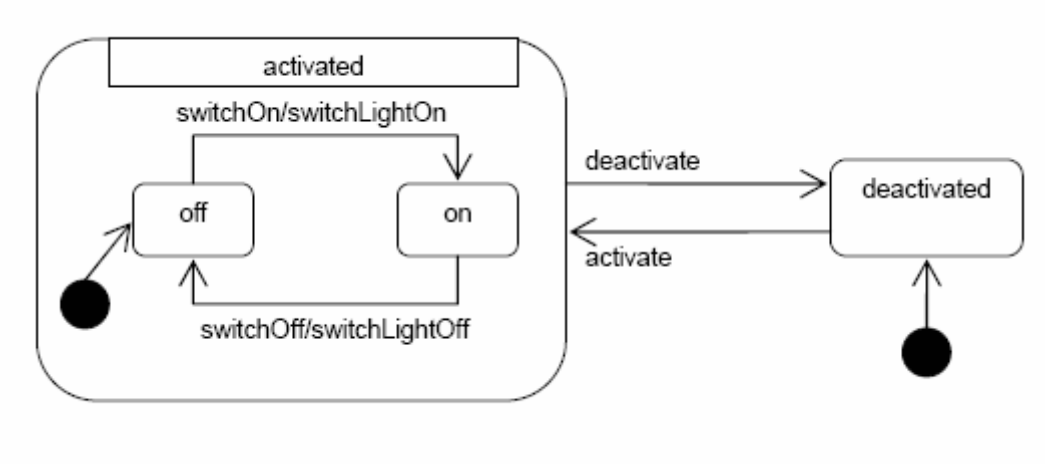

Figure 5: Traffic Light statechart diagram

Gate

As well as the traffic light diagram, has two top states *activated* and *deactivated*. In this case, the state *activated* has 6 sub-states: *opening, closing, opened, closed, faulty* and one Pseudostate that is the initial state when entering into the *activate* state. When the event *close* triggers the transition from state *opened* to *closing*, the *onEntry* actions are performed and the *startEngineDown* action takes place; if after a maximum period of time which is represented by *MAX_CLOSING_DURATION*, the signal *gateClosed* does not come, then a transition to the *faulty* state is performed, executing the *onExit* actions from the *closing* state before doing the transition; otherwise a transition to the state *closed* is performed. When the current state is *faulty*, then until the event *repaired* is triggered, a transition will take place going to the *deactivate* state. Something similar happens when the event *open* occurs as is depicted in the diagram shown in the next page.

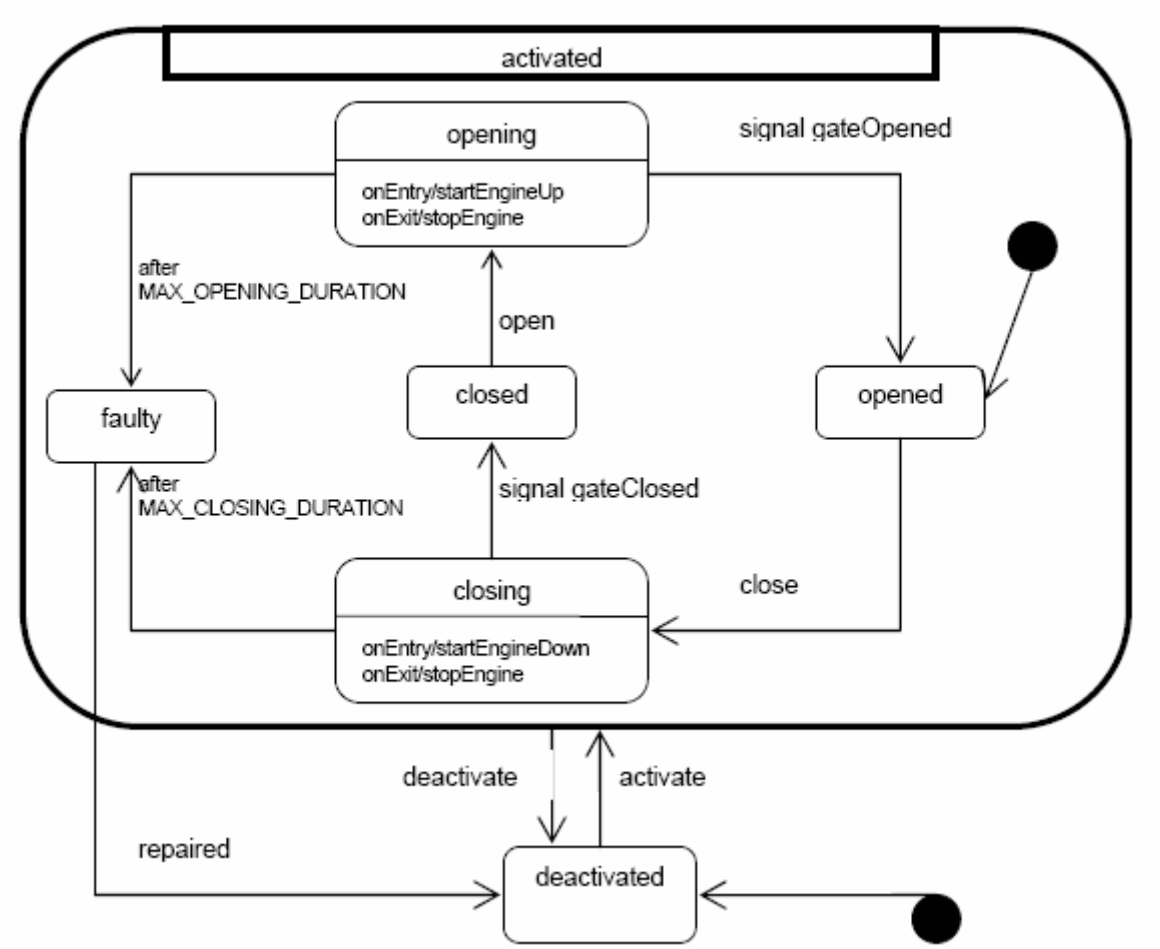

Figure 6: Gate Statechart diagram

Trail Observer

This diagram has also the two top components *activated* and *deactivated* as the previous ones. The initial state after entering in the *activate* state is the *idle* state. When the *trainApproaching* signal Event is triggered, a transition to the *trainPassing* state is performed; once this state is reached, stays there waiting for an event signal. If before receiving a signal the *MAX_PASSING_DURATION* time is reached, then a transition to the *faulty* state takes place. If the event signal received is *trainPassed*, also a transition to the *faulty* state is done. Otherwise, if the signal *trainApproaching* is received, a transition to the *idle* state is performed.

Whenever a transition reaches the *faulty* state, a transition take place to the *deactivated* state only after the *repaired* event occurs.

The diagram representing the Trail Observer statechart is shown in the next page:

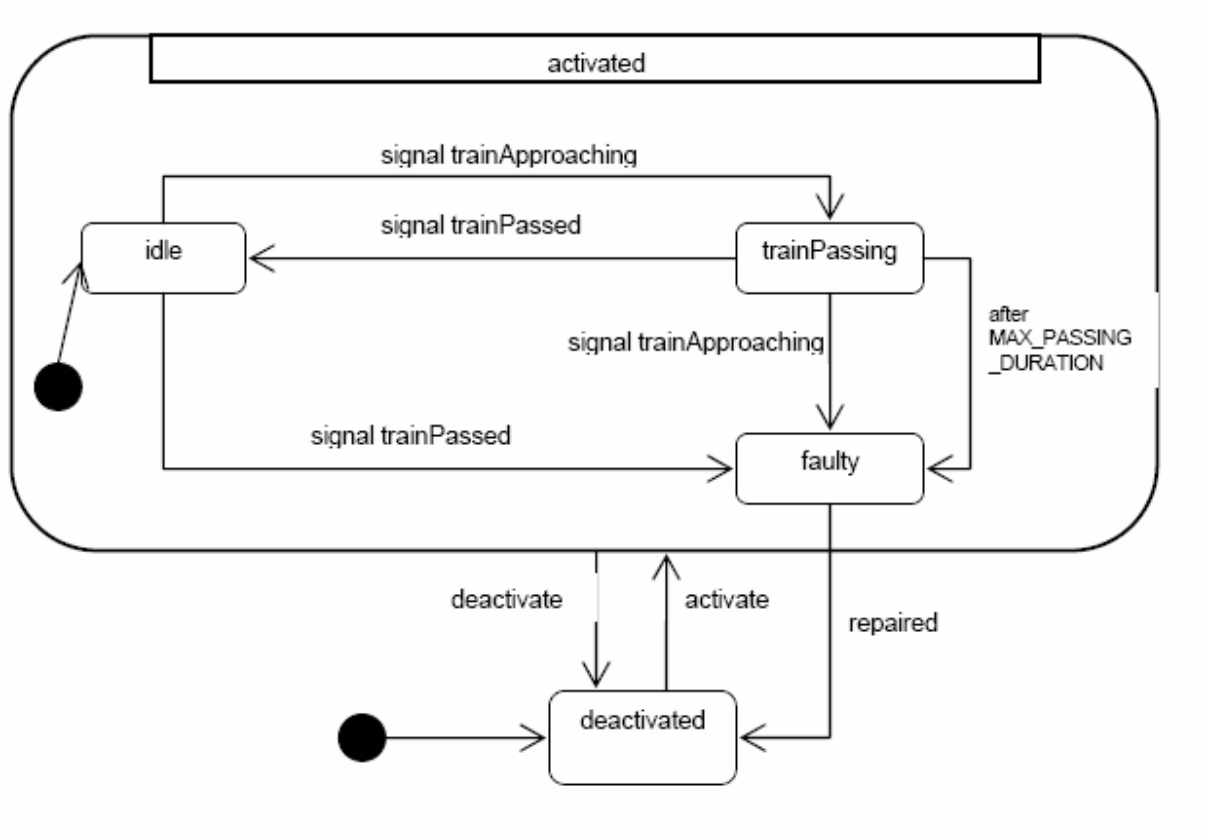

Figure 8: Trail Observer statechart diagram

Bahnübergang

This is the main controller statechart that integrates everything. It has two traffic light statemachines, two gates and two trail observers. Has also two top states, activated and deactivated. The initial state is deactivated. When the activate event occurs, a transition is made to the state activated, and then to the idle state. Then when the state *trainPassing* state is reached in either the trail observer 1 or the trail observer 2, a transition to the state *preparingToClose* is performed and the actions of this transition take place; switching on the lights of the traffic light 1 and 1 respectively. Once in the *preparingToClose* state, after a the time reaches the *CAR_STOP_DURATION* time, another transition is done by going to the state *closedForCars* and executing the actions closing the gate 1 and 2. From there, when both trail observers are in the state *idle*, the actions of open both gates are performed and stay there until both gates arrive to the state *opened*. When this occurs, a transition to the state *idle* is done and both traffic lights are switched off.

Whenever one of the gates or trail observer reaches the *faulty* state, a transition to the *deactivated* state is done no matter the current state, the gates are closed and the traffic lights are turned off.

The depiction of this diagram is shown in the next page:

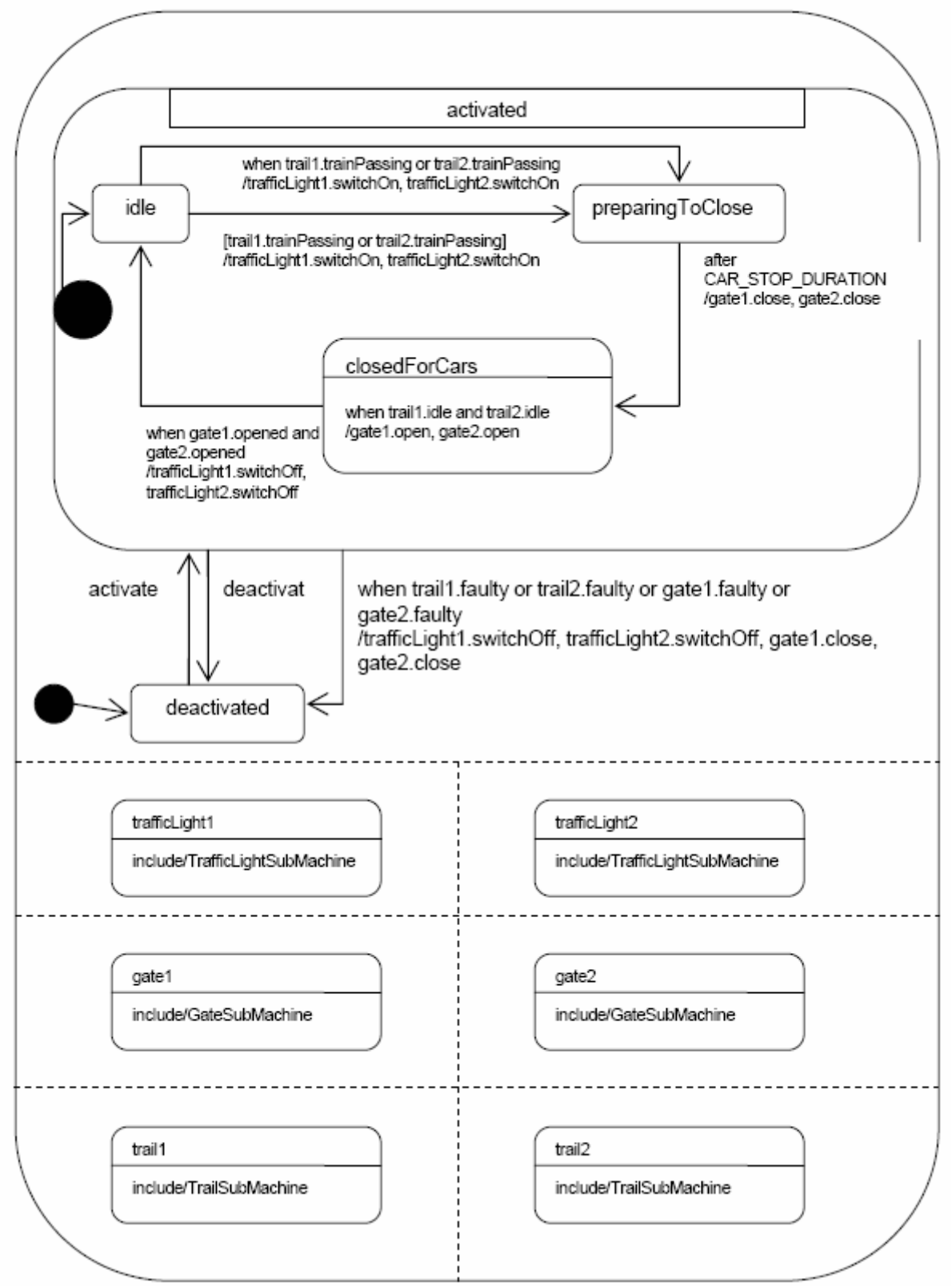

Figure 9: BahnÜbergang statechart diagram

Implementation

To implement this example, instances of the pre-defined classes of the metamodel where created. Additionally these classes were also implemented:

ProtocolViolationException.java

Raises a Protocol Violation Exception when an event generates an invalid transition. In other words, when an event call is executed, if the transition generated by this event does not exist in the current state, then the *ProtocolViolationException* is raised, this helps to control behavior on the implementation.

Gate.java

StateMachine instance that contains the states representing the gate statechart, its corresponding events, transitions, *onEntry()*, *onExit()* actions and the *executeEvent()* method.

TrafficLight.java

Statemachine instance representing the traffic light state chart, as the same as the Gate class, contains its corresponding states, events, transitions and methods.

TrailObserver.java

The same as the previous two classes, this one representing the Trail Observer statechart.

Bahn.java

This class represents the whole application, has the same elements as the previous three classes, but these also creates two instances to each of them to perform the concurrent statecharts.

Implementation details

Before showing an example of the execution of this implementation, is important to mention some aspects needed to introduce in these classes in order to achieve the expected result.

In the declaration of the classes of the four statecharts, all the states, events and transitions are created and some of them initialized. In the constructor of these classes the instance to the statemachine is created and the transitions are assigned to their corresponding events. Each statechart implementation has its own *onEntry()*, *onExit()* and *executeEvent()* methods, this last one is the same for all the classes and the other ones are adapted specific for the states corresponding on each of the classes.

Next, how looks like the declaration and constructor of the Traffic Light statechart implementation is shown in the next page:

```
public class TrafficLight {
    // Define Statemachine
    StateMachine trafficLight=null;
    // Define States
    public static final StateVertex DEACTIVATED=new SimpleState("DEACTIVATED");
    public static final StateVertex ACTIVATED= new CompositeState("ACTIVATED");//no parent
    public static final StateVertex ON = new CompositeState("ON",ACTIVATED);
    public static final StateVertex OFF= new CompositeState("OFF",DEACTIVATED);
    public static final StateVertex INITIALACTIVATED= new PseudoState("INITIALACTIVATED", "INITIAL");
    public StateVertex currentState= DEACTIVATED;
    public StateVertex historyState=null;
    // Define Events
    public Event switchOn=new Event("switchOn");
    public Event switchOff=new Event("switchOff");
    public Event activate=new Event("activate");
    public Event deactivate=new Event("deactivate");
    // Define Transitions
    public Transition t1= new Transition(OFF,ON,switchOn);
    public Transition t2= new Transition(ON,OFF,switchOff);
    public Transition t3= new Transition(ACTIVATED,DEACTIVATED,deactivate);
    public Transition t4= new Transition(DEACTIVATED,ACTIVATED,activate);
    public Transition t5= new Transition(ACTIVATED,INITIALACTIVATED,activate);
    public Transition t6= new Transition(INITIALACTIVATED,OFF,activate);

    public TrafficLight(String name){
        trafficLight= new StateMachine(new State(name));
        // Assign Substates
        ON.addParent(ACTIVATED);
        OFF.addParent(ACTIVATED);
        ACTIVATED.addSubState(ON);
        ACTIVATED.addSubState(OFF);
        // Add the transitions generated by Events
        switchOn.addTransition(t1);
        switchOff.addTransition(t2);
        activate.addTransition(t4);
        activate.addTransition(t5);
        activate.addTransition(t6);
        deactivate.addTransition(t3);
    }
```

Figure 10: Constructor declaration of traffic light statechart

In the class *Event.java*, the method *addTransition()* was modified to allow to automatically create the transitions from all the substates to their parent's existing transitions, this is a kind of transition inheritance. With this, for example, in the Gate statechart implementation, there was no necessary to define transitions from all the substates of the *activated* state to the *deactivated* state. This was done automatically in the mentioned method. Next I show the corresponding "transition inheritance" of the method.

```
// This function add the trasition to the event set and to the StateVertex set.
// Also generate automatically the outgoing transitions in Composite states by inheriting
// the trasitions from its parent
public void addTransition(Transition transition){

    if (transition.source instanceof CompositeState){
    //  CompositeState temp=(CompositeState)transition.source;
        Iterator it=transition.source.getSubStates().iterator();
        while (it.hasNext()){
            StateVertex st= (StateVertex)it.next();
            Transition t=new Transition(st,transition.target,transition.trigger);
            st.transitions.add(t);
            transitions.add(t);
        }
    }
    transitions.add(transition);
    transition.source.transitions.add(transition);
}
```

Figure 11: Declaration of the method *addTransition()*

For representing the initial states in the composite states, I created one Pseudostate on each statechart implementation called *INITIALACTIVATED.* When the *onEntry()* action of the *ACTIVATED* state is executed, then automatically 2 transitions are performed; one from the *ACTIVATED* state to the *INITIALACTIVATED*, and later from *INITIALACTIVATED* to the initial state defined in the transition.

With this as previous introduction, the implementation of the Bahn statechart is shown.
First all the states, transitions, events, and instances to the other statecharts are created and initialized. By default, everything is deactivated, including the Bahn statechart. In the main method, only the execution of the event *activate* is necessary, since from that point, the system starts to interact with the other statemachines following the protocol defined at the beginning of this example.
Is important to mention that in order to simulate failures of the gate, was implemented a random function that from time to time generate a transition to the *faulty* state, generating an alteration of the application enforcing the rules previously specified.
The *Thread.sleep()* method is invoked in several methods of the *onEntry()* actions of the Bahn class to make a delay and understand the process is following.
Several validations were made to avoid the raising of a *protocolViolationException* and to guarantee the protocol but if an invalid transition is forced, then it will in turn generate the *Exception.*
The activate event of the Bahn statechart, also activates the gates, traffic lights and trail observers. When entering to a state in the Bahn statechart, a checking is performed to see whether any gate or trail observer is faulty; when this case happens, the Bahn statechart executes a transition to the state *DEACTIVATED*; once in this state, verifies which element is in faulty state and repairs them. After that generates a transition automatically to the *ACTIVATED* state and thus, activating also all the deactivated components and starting over again the whole protocol. This is how the execution of the Bahn implementation keeps executing infinitely.

In the next page, part of the implementation of the *onEntry()* action of the Bahn class is shown, where the validations and delays are set.

```
public void onEntry(StateVertex state) throws ProtocolViolationException, InterruptedException{
    if (state==IDLE){
        System.out.println("onEntry Action from "+state.getName());
        while(train1.currentState!=train1.TRAINPASSING && train2.currentState!=train2.TRAINPASSING){
            Thread.sleep(1000);
            double randomValue = 3+ (Math.random() * 10);
            System.out.println(randomValue);
            if(checkFaulty())
                doTransition(t2);
            if (randomValue>7)
                train1.executeEvent(train1.trainApproaching);
            else
                train2.executeEvent(train2.trainApproaching);
        }
        if(checkFaulty())
            doTransition(t2);
        doTransition(t5);
    }
    else if (state==PREPARINGTOCLOSE){
        System.out.println("onEntry Action from "+state.getName());
        Thread.sleep(1000);
        if (gate1.currentState==gate1.OPENED)
            gate1.executeEvent(gate1.close);
        if (gate2.currentState==gate2.OPENED)
            gate2.executeEvent(gate2.close);
        if(checkFaulty()){
            doTransition(t2);
        }
        doTransition(t6);;
    }
    else if (state==ACTIVATED){
        System.out.println("onEntry Action from "+state.getName());
        if (train1.currentState==train1.FAULTY)
            train1.executeEvent(train1.repaired);
        if (train2.currentState==train2.FAULTY)
```

Figure 12: Part of the *onEntry()* method implementation

After executing the event Activate in the main method, as was pointed out before, all the other statechart instances are activated and entering into their corresponding initial state. Then the Bahn waits until the state of the other instances changes and then takes the corresponding action.

Next, part of the output of the execution of this implementation is shown:

```
onEntry Action from Gate.ACTIVATED
onExit Action from Gate.ACTIVATED
onEntry Action from Gate.INITIALACTIVATED
onExit Action from Gate.INITIALACTIVATED
onEntry Action from Gate.OPENED
onExit Action from ACTIVATED
onEntry Action from INITIALACTIVATED
onEntry Action from IDLE
9.173592576491744
onExit Action from TrailObserver.IDLE
onEntry Action from TrailObserver.TRAINPASSING
32.19452946048904
onExit Action from TrailObserver.TRAINPASSING
onEntry Action from TrailObserver.FAULTY
7.81743406548471
onExit Action from ACTIVATED
onEntry Action from DEACTIVATED
onExit Action from TrailObserver.FAULTY
onEntry Action from TrailObserver.DEACTIVATED
onExit Action from Gate.OPENED
onEntry Action from Gate.CLOSING
Gate.startEngineDown()
9.302555620957182
onExit Action from Gate.CLOSING
Gate.stopEngine()
onEntry Action from Gate.CLOSED
onExit Action from Gate.OPENED
onEntry Action from Gate.CLOSING
Gate.startEngineDown()
7.678678624373694
onExit Action from Gate.CLOSING
Gate.stopEngine()
onEntry Action from Gate.CLOSED
onExit Action from DEACTIVATED
onEntry Action from ACTIVATED
onExit Action from TrailObserver.DEACTIVATED
onEntry Action from TrailObserver.ACTIVATED
onExit Action from TrailObserver.ACTIVATED
onEntry Action from TrailObserver.INITIALACTIVATED
onEntry Action from TrailObserver.IDLE
onExit Action from ACTIVATED
onEntry Action from INITIALACTIVATED
```

Figure 13: Output of the Bahnübergang statechart implementation

In this part we can see that one of the *TrailObserver* instances, got into the *faulty* state and then was deactivated, this deactivated transition was triggered from the Gate implementation. Then at almost the bottom of the output, we can see that again is changed from *deactivated* state to *activated* state, and again, performed from the Gates implementation.

For more detail of the implementation, refer to the appendix at the end of this report.

7.3.2 Elevator example

In this implementation, new components from statechart are introduced: guards, choice-Pseudostate and history state.
For the development of this example, the following is assumed:
The elevator has 10 buttons representing a floor each one, starting from 0 until 9.
The floor 0 is the main and starting floor.
After reaching a floor, it stays there until a press-Button event is executed.
The alarm-button is not considered.

Next, I present the description of the components if the Elevator State chart.

Button

A Button inside an elevator represents a Floor. It has only two states: pressed and not-pressed. It has also an initial Pseudostate that goes to the not-pressed state.

The statechart diagram representing the Button used in this example is the following:

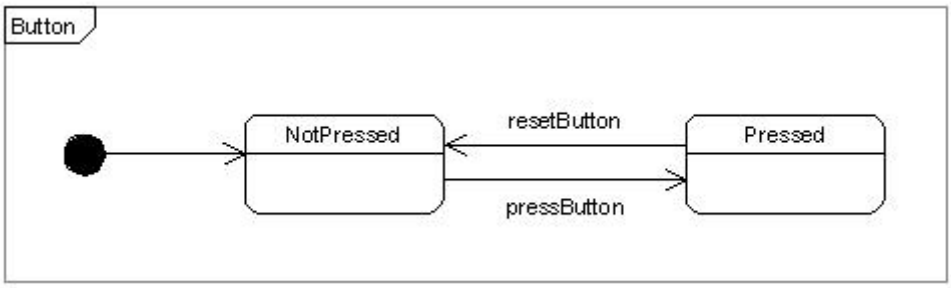

Figure 14: Button statechart diagram

Door

A door is another component of the Elevator and only has one. It has two main states called *activated* and *deactivated*; *activated* is a Composite state that includes the following sub-states: *faulty, opened, closed, opening, closing*. When the door is in the *opening* state, and a timeout limit of opening a door is reached, then a transition is executed to the *faulty* state. The similar happens while in the *closing* state. This is simulated by a random function in the implementation.
Only after the repaired event occurs, the door goes from the *faulty* state to the *deactivated* state.
The initial state of this statechart is *deactivated* and the representation of this statechart diagram used in this example is the following:

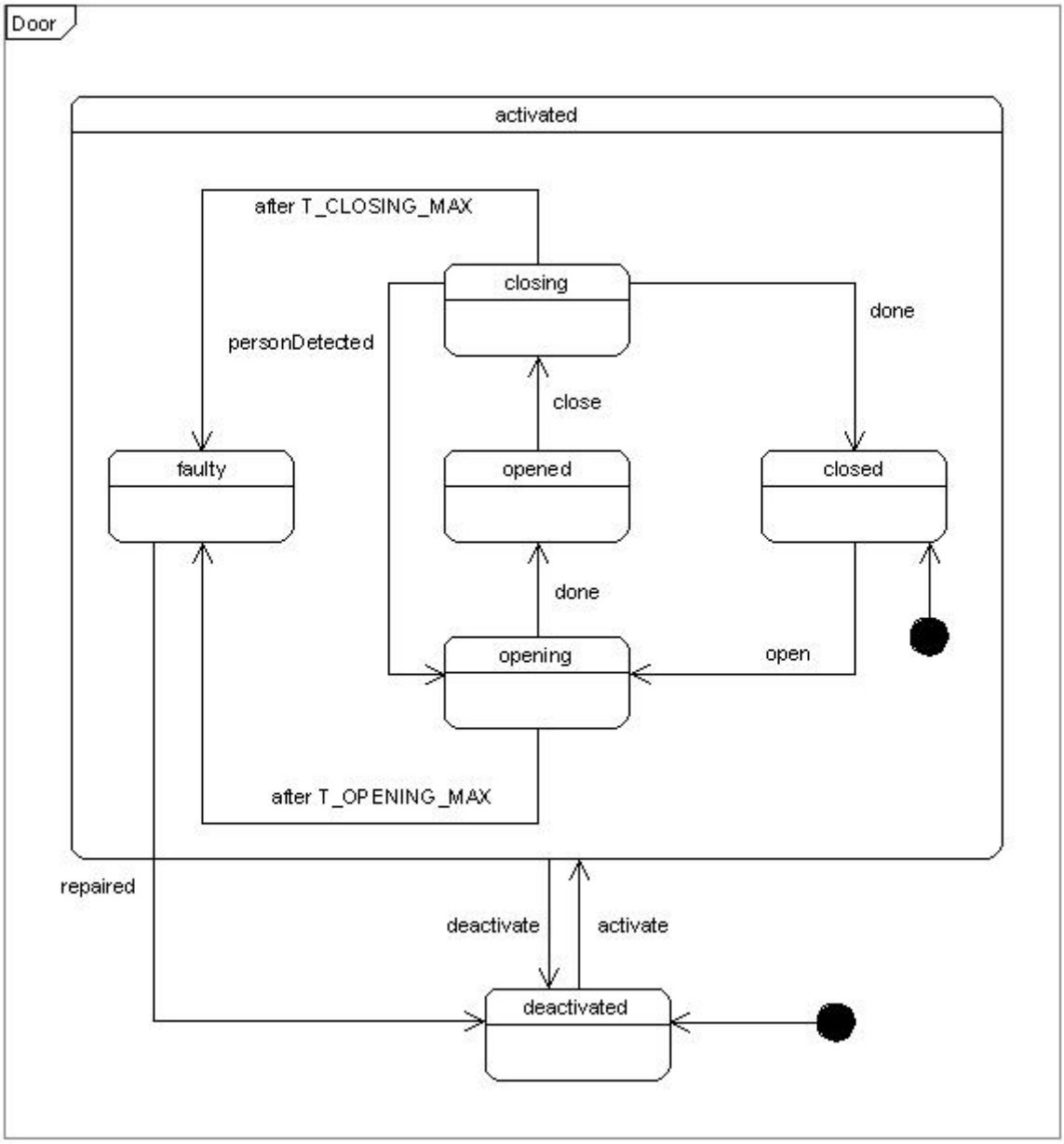

Figure 15: Door statechart diagram

Elevator

This represents the main statechart. It has a series of Buttons and one Door. Each of these components represented in orthogonal regions. Here is introduced the choice-Pseudostate, represented by a diamond in the diagram. In this implementation, a history-state is used to store the last moving-direction of the elevator. Depending on the history-State and the next Floor to move to, the decision is made to generate a transition to the *MovingUp* state or the *MovingDown* state.

Three guards are implemented to control the transitions. One is to verify that the next floor to move is not the current floor, and the other two are to verify that the door is closed before start moving either up or down.

The representation of this statechart used in this example is shown in the next page:

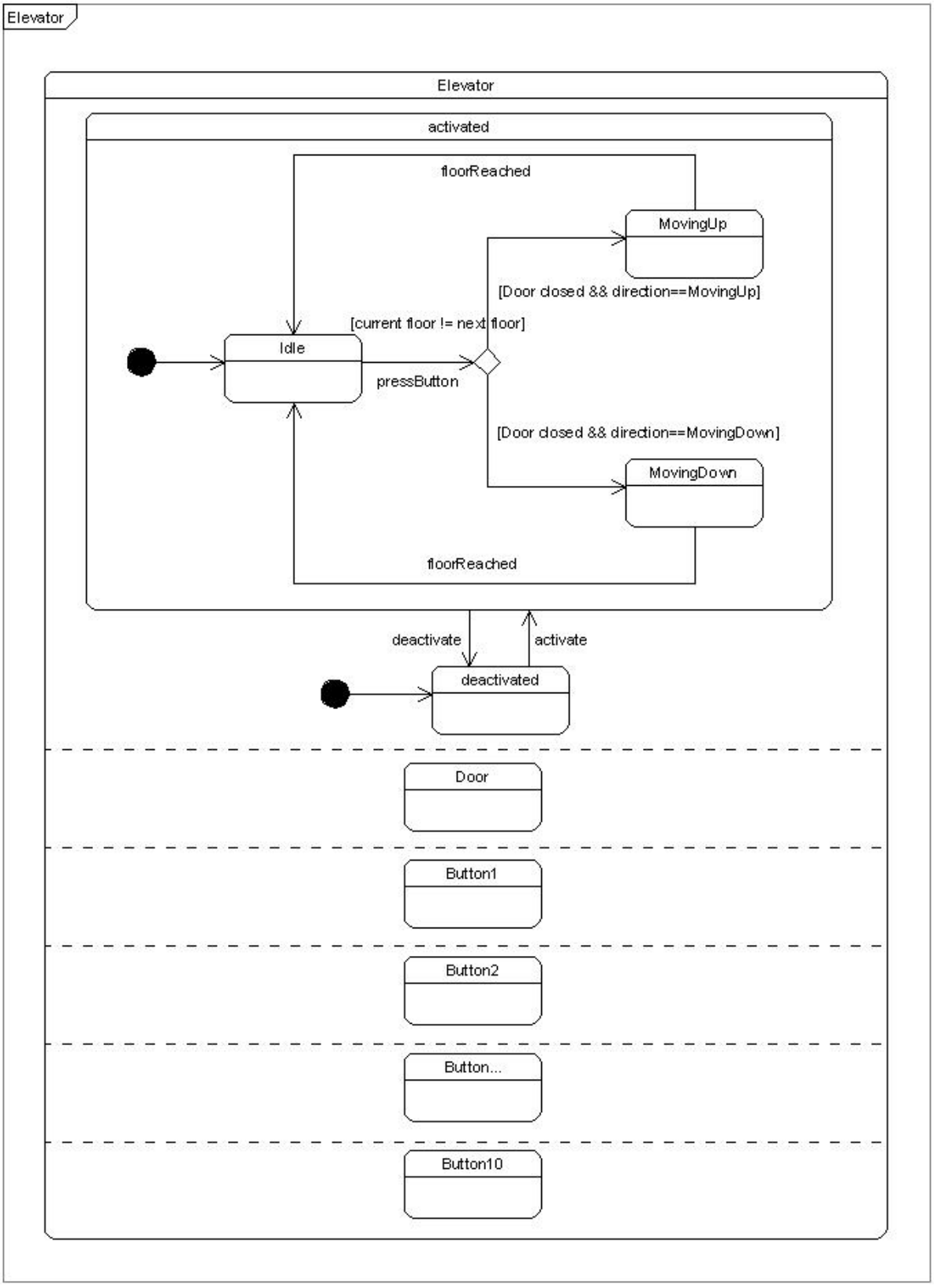

Figure 16: Elevator statechart diagram

Note that the State called *Button...* is used to abbreviate the states from *Button3* to *Button9*.

Implementation

To implement the elevator example, instances of the pre-defined classes of the metamodel where created. Additionally these classes were implemented:

ProtocolViolationException.java

Raises a Protocol Violation Exception when an event generates an invalid transition. In other words, when an event call is executed, if the transition generated by this event doesn't exist in the current state, then the *ProtocolViolationException* is raised, this helps to control behavior on the implementation.

Door.java

StateMachine instance that contains the states representing the Door statechart, its corresponding events, transitions, *onEntry()*, *onExit()* actions and the *executeEvent()* and *executeGuard()* methods.

Button.java

Represent a different floor in the elevator, can be pressed or not pressed.

Elevator.java

Main statemachine instance containing 10 Buttons, each of them in a different orthogonal region; and one door.

Here is implemented all the functionality and behavior of the interacting statecharts.

Implementation details

To create the 10 buttons of this elevator, an array of type Button was created, and then initialized automatically in the constructor. This was done in order to avoid to generate a lot of code for each Button (suppose that the elevator has 100 Buttons).

Two methods are created to retrieve the next floor to move depending on the moving direction, these methods iterate over the array starting from the current state, and return the next floor to go. The implementations of such methods are the following:

```
public int getNextFloorUp(){
    for (int i =currentFloor;i<=9;i++){
        if (buttons[i].currentState==Button.PRESSED)
            return i;

    }return currentFloor;
}
public int getNextFloorDown(){
    for (int i =currentFloor;i>=0;i--){
        if (buttons[i].currentState==Button.PRESSED)
            return i;

    }return currentFloor;
}
```

Figure 17: *getNextFloorUp()* and *getNextFloorDown* methods implementation

Like the example before, all the States are declared and initialized, as well as the new choice-Pseudostate.

A history-state is defined to store the last moving direction of the elevator. This is necessary to establish a well behavior of the elevator; suppose that the elevator is moving upwards going to the floor 5th, then, it still has 2 more floors to visit, the floors 7th and 9th. If the floor 2nd is pressed, first it has to go up and stop by the floor 7th and 9th respectively, and after that it should go down to the 2nd floor.

Two Integer variables are defined to store the current floor and the next floor to move.

When there are no more floors pressed, then the elevator stays in the last floor visited and wait until one button is pressed.
All this behavior specification is defined in the following methods: the *onEntry()* method of the *Idle* state (initial state after is activated), which performs the following:
If the last moving direction of the elevator (represented by the history-state) is going up and the next floor to move is bigger than the current floor, or the moving direction is going down and there are no more floors downwards to visit but there is one floor upwards to visit, then the next floor to visit is retrieved, the moving direction is updated and the transition to the choice Pseudostate is performed. Something similar happens when moving down. All of this is represented in the following code:

```
public void onEntry(StateVertex state) throws ProtocolViolationException, InterruptedException{
    if (state==IDLE){
        System.out.println("onEntry Action from Elevator."+state.getName());
        Thread.sleep(1000);
        if((historyState==MOVINGUP&&getNextFloorUp()>currentFloor)||(historyState==MOVINGDOWN&&
                getNextFloorUp()>currentFloor&&getNextFloorDown()==currentFloor)){
            nextFloor=getNextFloorUp();
            historyState=MOVINGUP;
            doTransition(t9);
            return;
        }
        else if((historyState==MOVINGDOWN&&getNextFloorDown()<currentFloor)||(historyState==MOVINGUP&&
                getNextFloorDown()<currentFloor&&getNextFloorUp()==currentFloor)){
            nextFloor=getNextFloorDown();
            historyState=MOVINGDOWN;
            doTransition(t9);
            return;
        }
        else
            System.out.println("No Button Pressed");
    }
    else if (state==ACTIVATED){
```

Figure 18: behavior for the elevator defined in the state *IDLE* of the *onEntry()* method

After a transition to the choice-Pseudostate is performed, is necessary to open and close the door and update the current moving direction and retrieve the next floor to move, this is defined in the *onExit()* method of the Idle state. This is necessary due to avoid falling in a false Guard validation in the initial state when the current floor and the next floor are both 0. This specification is defined as follows:

```
public void onExit(StateVertex state) throws ProtocolViolationException, InterruptedException{
    if (state==IDLE){
        System.out.println("onExit Action from Elevator."+state.getName());
        if((historyState==MOVINGUP&&getNextFloorUp()>currentFloor)||(historyState==MOVINGDOWN&&
                getNextFloorUp()>currentFloor&&getNextFloorDown()==currentFloor)){
            nextFloor=getNextFloorUp();
            historyState=MOVINGUP;
            if (door.currentState==door.CLOSED){
                door.executeEvent(door.open);
                System.out.println("Opening door...");
            }
            Thread.sleep(500);
            if (door.currentState==door.OPENED){
                door.executeEvent(door.close);
                System.out.println("Closing door...");
            }
        }
        else if((historyState==MOVINGDOWN&&getNextFloorDown()<currentFloor)||(historyState==MOVINGUP&&
                getNextFloorDown()<currentFloor&&getNextFloorUp()==currentFloor)){
            nextFloor=getNextFloorDown();
            historyState=MOVINGDOWN;
            if (door.currentState==door.CLOSED){
                door.executeEvent(door.open);
                System.out.println("Opening door...");
            }
            Thread.sleep(500);
            if (door.currentState==door.OPENED){
                door.executeEvent(door.close);
                System.out.println("Closing door...");
            }
```

Figure 19: Definition of opening and closing the elevator's door in the *onExit()* method

After retrieving the next floor to move correctly in the right direction defined in the previous code, is necessary to implement the behavior for the choice-Pseudostate (represented by the *MOVING* state).
When a transition arrives to this Pseudostate, a decision is made depending on the direction to move based on the history-state. After one decision is made, evaluates the corresponding Guard and after successful validation, moves either towards *MovingUp* state or *MovingDown* state.
This decision making is defined in the *onEntry()* actions of the choice-Pseudostate and looks like the following.

```
else if (state==MOVING){
    System.out.println("onEntry Action from Elevator."+state.getName());
    Thread.sleep(1000);
    if (historyState==MOVINGUP){
        System.out.println("Current Floor = "+currentFloor);
        System.out.println("Next Floor Up= "+nextFloor);
        doTransition(t10);
        return;
    }
    else if (historyState==MOVINGDOWN){
        System.out.println("Current Floor = "+currentFloor);
        System.out.println("Next Floor Down= "+nextFloor);
        doTransition(t11);
        return;
    }
```

Figure 20: Using the history-state in the choice-state

Once the target floor is reached, it is important to turn off the corresponding button in the array. This is implemented in the *onExit()* actions from the *MOVINGUP* and *MOVINGDOWN* states, as shown in the next page:

```
else if (state==ACTIVATED){
    System.out.println("onExit Action from Elevator."+state.getName());
}
else if (state==MOVINGUP){
    System.out.println("onExit Action from Elevator."+state.getName());
    if (buttons[currentFloor].currentState==buttons[currentFloor].PRESSED){
        buttons[currentFloor].executeEvent(buttons[currentFloor].resetButton);
        System.out.println("Button "+currentFloor+" reseted");
    }
}
else if (state==MOVINGDOWN){
    System.out.println("onExit Action from Elevator."+state.getName());
    if (buttons[currentFloor].currentState==buttons[currentFloor].PRESSED){
        buttons[currentFloor].executeEvent(buttons[currentFloor].resetButton);
        System.out.println("Button "+currentFloor+" reseted");
    }
```

Figure 21: Turning off the button of the reached floor

As difference from the Bahnübergang implementation, here is introduced the guards in the code to validate the execution of transactions.
In order to achieve the functionality of the guards, the method *executeGuard()* is created which validates the execution of a transition in some state.
After several attempts to find out the best way to implement this validation, I decided to implement them in a similar manner like the *onEntry()* and *onExit()* methods. Thus, the code validation has to be introduced only once within this method and it will be executed before whenever a transition takes place.
The code enforcing the guards validation in this example is the following:

```
public boolean executeGuard(Guard guard){
    boolean result=false;
    if (guard==movingUpGuard){
        if (door.currentState==door.CLOSED&&nextFloor>currentFloor)
            result=true;
        else
            result=false;
    }
    else if (guard==movingDownGuard){
        if (door.currentState==door.CLOSED&&nextFloor<currentFloor)
            result=true;
        else
            result=false;
    }
    else if (guard==buttonPressedGuard){
        if (currentFloor!=getNextFloorUp()||currentFloor!=getNextFloorDown())
            result=true;
        else
            result=false;
    }
    return result;
}
```

Figure 22: Implementation of the guards in the elevators statechart example

In this code, the first and second Guards, establishes that the door must be closed and that the next floor to move is a valid one depending on the direction. The third Guard defines that when a button is pressed, it will execute a transition only when the next floor to move is not the same as the current floor.

Once the behavior definition for the statechart is defined as explained before, a main method was created to have an entry-point to the statechart.
In this run-execution, I created an instance to the Elevator class; next, I activated the door. After that I activated the Elevator. Then, I simulated that two persons request the service of the elevator at different floors; in this case at the floor 5th. and 8th. This is done by explicitly indicating the floor as shown in the code.
Then the *pressButton* event is triggered in order to attend to these requests and start moving. After this floors are visited, then its simulated that another request is done for the 2nd floor by either somebody requesting the service in that floor, or somebody inside the Elevator pressed the 2nd button. Then the method *pressButton* is executed again and the elevator moves to the corresponding floor and stays there.
The code for the main method in this test is the following:

```
public static void main (String[] args)throws ProtocolViolationException, InterruptedException{
    Elevator e=new Elevator("Elevator");
    e.door.executeEvent(e.door.activate);
    e.executeEvent(e.activate);
    e.buttons[5].executeEvent(e.buttons[5].pressButton);
    e.buttons[8].executeEvent(e.buttons[8].pressButton);
    e.executeEvent(e.pressButton);
    e.buttons[2].executeEvent(e.buttons[2].pressButton);
    e.executeEvent(e.pressButton);
    e.executeEvent(e.pressButton);
}
```

Figure 23: Main method implementation

The output of the execution of this implementation is the following:

```
onExit Action from Gate.DEACTIVATED
onEntry Action from Door.ACTIVATED
onExit Action from Gate.ACTIVATED
onEntry Action from Door.INITIALACTIVATED
onExit Action from Gate.INITIALACTIVATED
onEntry Action from Door.CLOSED
onExit Action from Elevator.DEACTIVATED
onEntry Action from Elevator.ACTIVATED
onExit Action from Elevator.ACTIVATED
onEntry Action from Elevator.IDLE
No Button Pressed
onExit Action from Button.NOTPRESSED
onEntry Action from Button.PRESSED
onExit Action from Button.NOTPRESSED
onEntry Action from Button.PRESSED
onExit Action from Elevator.IDLE
onExit Action from Gate.CLOSED
onEntry Action from Door.OPENING
6.286695467853814
onExit Action from Gate.OPENING
onEntry Action from Door.OPENED
Opening door...
onExit Action from Gate.OPENED
onEntry Action from Door.CLOSING
6.805590158751868
onExit Action from Gate.CLOSING
onEntry Action from Door.CLOSED
Closing door...
onEntry Action from Elevator.MOVING
Current Floor = 0
Next Floor Up= 5
onExit Action from Elevator.MOVING
onEntry Action from Elevator.MOVINGUP
onExit Action from Elevator.MOVINGUP
onExit Action from Button.PRESSED
onEntry Action from Button.NOTPRESSED
Button 5 reseted
onEntry Action from Elevator.IDLE
onExit Action from Elevator.IDLE
onExit Action from Gate.CLOSED
onEntry Action from Door.OPENING
7.812877424799556
onExit Action from Gate.OPENING
onEntry Action from Door.OPENED
Opening door...
onExit Action from Gate.OPENED
onEntry Action from Door.CLOSING
5.018280789432293
onExit Action from Gate.CLOSING
onEntry Action from Door.CLOSED
Closing door...
onEntry Action from Elevator.MOVING
Current Floor = 5
Next Floor Up= 8
onExit Action from Elevator.MOVING
onEntry Action from Elevator.MOVINGUP
onExit Action from Elevator.MOVINGUP
onExit Action from Button.PRESSED
onEntry Action from Button.NOTPRESSED
Button 8 reseted
onEntry Action from Elevator.IDLE
No Button Pressed
onExit Action from Button.NOTPRESSED
onEntry Action from Button.PRESSED
onExit Action from Elevator.IDLE
onExit Action from Gate.CLOSED
onEntry Action from Door.OPENING
8.583961203487345
onExit Action from Gate.OPENING
onEntry Action from Door.OPENED
Opening door...
onExit Action from Gate.OPENED
onEntry Action from Door.CLOSING
8.699566213108062
onExit Action from Gate.CLOSING
onEntry Action from Door.CLOSED
Closing door...
onEntry Action from Elevator.MOVING
Current Floor = 8
Next Floor Down= 2
onExit Action from Elevator.MOVING
onEntry Action from Elevator.MOVINGDOWN
onExit Action from Elevator.MOVINGDOWN
onExit Action from Button.PRESSED
onEntry Action from Button.NOTPRESSED
Button 2 reseted
onEntry Action from Elevator.IDLE
No Button Pressed
Transition not executed, Guard=false
```

Figure 24: Output for the Elevator statechart implementation

As we could see in this output, the desired behavior is reached. First, the elevator is moved from the 0^{th} to the 5^{th} and 8^{th} floor and stays there, then is moved effectively to the second floor and stays there waiting for further requests.

Note: For a complete view of the code of this implementation, please refer to the appendix at the end of this report.

8. Octopus Integration

In order to follow the MDA paradigm, I developed in the last iteration a PIM (Platform Independent Model) for representing the metamodel. In this case I used Octopus to generate big part of the code of the Statecharts in Java by using .uml and .ocl specifications. Nevertheless it was not possible to generate the complete functionality of the model by using only these specifications because of the existing code-generation limitations on the MDA process. So it was needed to manually adapt some of them.

8.1 Generation of metamodel with Octopus

To generate the metamodel using Octopus, I created a .uml specification that generates the classes I developed manually in the last iterations. This .uml specification follows a specified syntax defined by Octopus; which is quite understandable by the readers.
Part of the .uml model I implemented is the following:

```
<package> test

<class> StateVertex
      <attributes>
            + name: String;
            + transitions: Set(Transition);
      <operations>
            + addParent(s: StateVertex);
            + addSubState(s: StateVertex);
            + getSubStates(): Set(StateVertex);
<endclass>

<class> Transition
<endclass>

<class> Event
      <attributes>
            + name: String;
<endclass>

<class> PseudoState <specializes> StateVertex
      <attributes>
      + kind: String;
<endclass>

<class> CompositeState <specializes> State
<endclass>

<associations>
      + StateMachine.statemachine[0..1] <-> State.top[1];
      + Transition.<noName>[0..*] -> + StateVertex.source[1];
      + Transition.<noName>[0..*] -> + StateVertex.target[1];
      + Transition.transition[1] <-> + Guard.guard[0..1];
      + Transition.transitions[0..*] <-> + Event.trigger[0..1];
      + State.<noName>[0..*] -> + Transition.internalTransition[1];
      + State.states[0..*] <-> + Event.defferedEvent[0..*];
      + StateVertex.substates[0..*] <-> CompositeState.parent[1];
<endpackage>
```

Here, the definition of a class is specified inside the tags **<class> <endclass>**, with their respective attributes and methods (if any). The associations are specified after the tag **<associations>**, with their respective cardinality and navigation.
For a complete view of the .uml specification, refer to the appendix at the end of this report.

After defining the model, I defined some OCL expressions in order to constraint a bit more the .uml model.
OCL is powerful for complementing UML models, and by adding OCL expressions in my model, more information can be added to it. Because of the project deadline time limitation, in this iteration at the last stage of my project work I implemented just a few OCL expressions for specifying the initial values that some objects needs to take. But more things can be carried out with these expressions like constraining guards or any other object's vale.
In this case, I didn't use OCL expressions in the metamodel generation, but in the *Elevator* example implemented and explained later on in this report.

Once the .uml and .ocl specifications are defined in Octopus, the code by using the Octopus code-generation feature is generated. Some code adaptations needed to be done for getting the desired functionality. One of the methods to be adapted was the method *addToTransitions(Transition t)* from the generated class *Event*.
The adaptation looks like the following:

```
/** Implements addition of a single element to feature '+ transitions : Set(Transition)'

 GENBY NavigationCreator::generateAssociationToMany() , includeField , usesFacade=false
 *
 * @param element
 */
public void addToTransitions(Transition transition) {
    if ( transition == null ) {
        return;
    }
    if ( this.f_transitions.contains(transition) ) {
        return;
    }
    if (transition.getSource() instanceof CompositeState){
        Set<StateVertex> set= transition.getSource().getSubStates();
            Iterator it=set.iterator();
            while (it.hasNext()){
                StateVertex st= (StateVertex)it.next();
                Transition t=new Transition(st,transition.getTarget(),transition.getTrigger());
                st.getTransitions().add(t);
                getTransitions().add(t);
            }
        }
        f_transitions.add(transition);
        transition.getSource().getTransitions().add(transition);
    }
```

Figure 25: Adaptation of the method *addToTransitions()* generated by Octopus

In this method implementation, we added the functionality to add the outgoing transitions of a composite state to all of its sub-states.
Another method to be adapted was the constructor of the generated classes Transition and Guard; in the first one was necessary to add automatically the transition to the triggered event, and in the second case, the Guard added itself to the corresponding transition received as parameter in the constructor. Both cases look as follows respectively:

```
public Transition(StateVertex source, StateVertex target, Event trigger, Guard guard){
    this.f_source= source;//.getStateVertex(source);
    this.f_target=target;//.getStateVertex(target);
    this.f_trigger=trigger;
    this.f_guard=guard;
    this.f_trigger.addToTransitions(this);
}
public Transition(StateVertex source, StateVertex target, Event trigger){
    this.f_source= source;//.getStateVertex(source);
    this.f_target=target;//.getStateVertex(target);
    this.f_trigger=trigger;
    this.f_guard= null;
    this.f_trigger.addToTransitions(this);
}
```

Figure 26: Adaptation of the *Transition* constructors

```
public Guard(Transition t){
    this.setTransition(t);
    t.setGuard(this);
}
```

Figure 27: Adaptation of the Guard constructor

For a complete view of the adaptation of the generated code with Octopus, refer to the appendix at the end of this report.

8.2 Generation of the Elevator example with Octopus

After having the code of the metamodel generated with Octopus. I implemented in a similar way a .uml and .ocl specifications to generate the skeleton of the Elevator example.
One of the classes defined in the .uml specification looks like follows:

```
<class> Button
    <attributes>
        + statemachine: StateMachine;
        + PRESSED: StateVertex;
        + NOTPRESSED: StateVertex;
        + INITIAL: StateVertex;
        + currentState: StateVertex;
        + pressButton: Event;
        + resetButton: Event;
        + initial: Event;
        + t1: Transition;
        + t2: Transition;
        + t3: Transition;
        <operations>
        + executeEvent(e: Event);
        + executeGuard(g: Guard): Boolean;
        + doTransition(t: Transition);
        + onEntry(state: StateVertex);
        + onExit(state: StateVertex);
<endclass>
```

Figure 28: Definition of the Button class in the .uml specification

Some of the OCL expressions I generated are the following. Refer to the appendix to a complete view of these expressions.

```
context Elevator:: nextFloor: Integer
      init: 0

context Door:: currentState: StateVertex
      init: DEACTIVATED

context Door:: MAX_CLOSING_DURATION: Integer
      init: 10
```

The first expression refers to the Elevator class defined by the **context** *Elevator*, then specifies the attribute *nextFloor* of type *Integer* will have the initial value of "0". The similar happens with the next expressions.
To a complete view of the .uml and .ocl specification for this example, refer to the appendix at the end of this report.

Like in the metamodel code generation, some methods needed to be adapted. In this case, the behavior of the Elevator's Statecharts relies on the *onEntry()* and *onExit()* methods and needed to be implemented by hand. As well as the initialization of the States, Transitions, Events and so on.
One of such adaptations looks as follows:

```
/** Implements the user defined operation '+ onExit( state: StateVertex )'
 *
 * @param state
 */
public void onExit(StateVertex state) throws ProtocolViolationException, InterruptedException{

    if (state==f_iDLE){
        System.out.println("onExit Action from Elevator."+state.getName());
        if((f_historyState==f_mOVINGUP&&getNextFloorUp()>f_currentFloor)||(f_historyState==f_mOVINGDOWN&&
                getNextFloorUp()>f_currentFloor&&getNextFloorDown()==f_currentFloor)){
            f_nextFloor=getNextFloorUp();
            f_historyState=f_mOVINGUP;
            if (door.getCurrentState()==door.getCLOSED()){
                door.executeEvent(door.getOpen());
                System.out.println("Opening door...");
            }
            Thread.sleep(500);
            if (door.getCurrentState()==door.getOPENED()){
                door.executeEvent(door.getClose());
                System.out.println("Closing door...");
            }
        }
        else if((f_historyState==f_mOVINGDOWN&&getNextFloorDown()<f_currentFloor)||(f_historyState==f_mOVINGUP&&
                getNextFloorDown()<f_currentFloor&&getNextFloorUp()==f_currentFloor)){
            f_nextFloor=getNextFloorDown();
            f_historyState=f_mOVINGDOWN;
            if (door.getCurrentState()==door.getCLOSED()){
                door.executeEvent(door.getOpen());
                System.out.println("Opening door...");
            }
            Thread.sleep(500);
            if (door.getCurrentState()==door.getOPENED()){
                door.executeEvent(door.getClose());
                System.out.println("Closing door...");
            }
        }

    }
```

Figure 29: Adaptation of the *onExit()* actions of the state *f_iDLE* generated by Octopus

To get a complete view of the adapted source code, refer to the appendix at the end of this report.

After having adapted the generated code from the .uml and .ocl specifications, the following main method was implemented as an entry point in a similar way as the one defined in the first Elevator implementation:

```
public static void main (String[] args)throws ProtocolViolationException, InterruptedException{
    Elevator e=new Elevator("Elevator");
    e.door.executeEvent(e.door.getActivate());
    e.executeEvent(e.getActivate());
    e.buttons[5].executeEvent(e.buttons[5].getPressButton());
    e.buttons[8].executeEvent(e.buttons[8].getPressButton());
    e.executeEvent(e.getPressButton());
    e.buttons[2].executeEvent(e.buttons[2].getPressButton());
    e.executeEvent(e.getPressButton());
    e.executeEvent(e.getPressButton());
}
```

Figure 30: Implementation of the main method for the Elevator example generated by Octopus

With this, I got the same functionality of the Elevator example as the previous implementation done without Octopus, but simplifying the code implementation by generating a big part of the metamodel and the skeleton of the desired Statechart.

9. Conclusion

During the development of this project work, I discovered and dealt with different existing difficulties when implementing Statecharts. That drove me to analyze different situations the Statecharts can fall into and to develop a prototype that helps for an easier implementation of the basic statechart notation.

Starting from getting background knowledge of Statecharts and exploring different technologies to develop a prototype, I ended up by just choosing a few of them, the ones I considered necessary to not make my prototype too complex to understand and implement.

Even though at the beginning of the project I had the intention to generate a prototype for implementing Statecharts using Aspects in a compilation based style, I arrived to a different implementation based on an interpretative one. Thus, having the potentially advantage of allowing adapting the statechart at runtime and giving more flexibility.

Because of the time limit of the duration of this project work, only the main components of the Statechart diagrams are able to be implemented with this prototype, leaving place for future extension and adaptability of this prototype with further components like Fork and Join Pseudostates.

At the last stage of the project, the integration with Octopus was done and I generated my metamodel prototype from .uml specifications and reducing considerably the implementation effort of it. Further adaptability and improvements can be done by using also OCL expressions like the ones used in the examples implementations.

I hope this work helps the reader to get a better understanding about the implementation of Statecharts and the possible different solutions that can be reached by using the prototype I developed following the MDA in the incremental software development way.

10. References

[1] Statecharts, *by Ben Meadowcroft.* http://www.benmeadowcroft.com/reports/statechart/
[2] UML Statecharts, *by Bruce Powel Douglass* http://www.embedded.com/1999/9901/9901feat1.htm
[3] Taking Aspect One Step Further, *By James Holmes.* http://www.oracle.com/technology/oramag/oracle/04-sep/o54aop.html
[4] Nanning Aspects. http://nanning.codehaus.org/
[5] JBoss Aspect Oriented Programming. http://labs.jboss.com/portal/jbossaop/index.html
[6] AspectWerkz, Plain Java AOP. http://aspectwerkz.codehaus.org/
[7] AspectJ for Eclipse. http://www.eclipse.org/aspectj/
[8] Executable UML: A foundation for Model Driven Architecture. By *Stephen J. Mellor, Marc J. Balcer*. Addison Wesley. May 14, 2002.
[9] OMG Model Driven Architecture.http://www.omg.org/mda/
[10] MDA Explained: The Model Driven Architecture™: Practice and Promise, By *Anneke Kleppe, Jos Warmer, Wim Bast*. Addison Wesley. April 21, 2003.
[11] Object Constraint Language, by *Jos Warmer* and *Anneke Kleppe*. http://www.klasse.nl/ocl
[12] Introduction to OCL, by *Jos Warmer* and *Anneke Kleppe*. http://www.klasse.nl/ocl/ocl-introduction.html
[13] Octopus: OCL Tool for precise UML Specifications, by *Jos Warmer* and *Anneke Kleppe*. http://www.klasse.nl/octopus/
[14] UML Statecharts, by *Max Göbel*. http://tfs.cs.tu-berlin.de/vila/www_ws03/folien/statecharts_ausarbeitung_final.pdf

Additional sources

- AspectJ in Action, Practical Aspect Oriented Programming. *Ramnivas Laddad*. Manning
- The Object Constraing Language. Jos Warmer, Anneke Kleppe. Addison Wesley. August 29, 2003.
- UML2 Diagrams. http://www.visual-paradigm.com/VPGallery/diagrams/State.html
- UML2 Statechart Diagrams, by *SPARX Systems*. http://www.sparxsystems.com/resources/uml2_tutorial/uml2_statediagram.html
- UML Metamodel. http://www.cse.msu.edu/~cse870/Materials/UML11_Metamodel_Diagrams.pdf
- Mapping UML Statecharts to Java code. By *Azim Niaz, Jiro Tanaka.* http://www.iplab.cs.tsukuba.ac.jp/paper/international/niaz_se2004.pdf
- Using Aspects to Abstract and Modularize Statecharts, by *Mark Mahoney, Atef Bader, Tzilla Elrad, Omar Aldawud*. http://www.cs.iit.edu/~oaldawud/AOM/mahoney.pdf

11. Appendix

Metamodel source code

StateMachine.java

```
public class StateMachine {
	public State top;

	public StateMachine(State state){
		this.top=state;
	}
}
```

StateVertex.java

```
public class StateVertex {
private String name;
private CompositeState parent;
public Set <Transition> transitions= new HashSet();

public StateVertex(String state){
	this.name=state;
	this.parent=null;
}
public StateVertex(String state, CompositeState parent){
	this.name=state;
	this.parent=parent;
}
public void addSubState(StateVertex state){}
public Set<StateVertex> getSubStates(){return null;}
public String getName(){
	return name;
}
public void setName(String name){
	this.name=name;
}
}
```

State.java

```
public class State extends StateVertex{
public StateMachine statemachine;
public Transition internalTransition=null;
public Set <Event> deferredEvents= new HashSet();

public State(String state){
	super(state);
}
}
```

PseudoState.java

```
public class PseudoState extends StateVertex{
public String kind;

public PseudoState(String name, String kind){
	super(name);
	this.kind=kind;
}
}
```

SimpleState.java

```
public class SimpleState extends State{
public SimpleState(String name){
        super(name);
}
}
```

CompositeState.java

```
public class CompositeState extends State {
public Set <StateVertex> substates= new HashSet();

public CompositeState(String name){
        super(name);
}
public CompositeState(String name, StateVertex parent){
        super(name);
        if (parent!=null)
                parent.addSubState(this);
}
public void addSubState (StateVertex state){
        substates.add(state);
}
public Set<StateVertex> getSubStates(){
        return substates;
}
}
```

Transition.java

```
public class Transition {
        public StateVertex source=null;
        public StateVertex target=null;
        public Event trigger=null;
        public Guard guard=null;

        public Transition(StateVertex source, StateVertex target, Event trigger, Guard guard){
                this.source= source;
                this.target=target;
                this.trigger=trigger;
                this.guard=guard;
                trigger.addTransition(this);
        }
        public Transition(StateVertex source, StateVertex target, Event trigger){
                this.source= source;
                this.target=target;
                this.trigger=trigger;
                this.guard= null;
                trigger.addTransition(this);
        }
        public void addGuard(Guard g){
                this.guard=g;
        }
}
```

Event.java.

```
public class Event {
private String name;
public Set <Transition> transitions=new HashSet();
public Set <State> states=new HashSet();

public Event(String name){
        this.name=name;
}

// This function add the trasition to the event set and to the StateVertex set.
// Also generate automatically the outgoing transitions in Composite states by inheriting
// the trasitions from its parent
public void addTransition(Transition transition){
        if (transition.source instanceof CompositeState){
                Iterator it=transition.source.getSubStates().iterator();
                while (it.hasNext()){
```

```
                StateVertex st= (StateVertex)it.next();
                Transition t=new Transition(st,transition.target,transition.trigger);
                st.transitions.add(t);
                transitions.add(t);
            }
        }
        transitions.add(transition);
        transition.source.transitions.add(transition);
}
public void addToState(State state){
        states.add(state);
}
}
```

Guard.java

```
public class Guard {
Transition transition=null;

public Guard(Transition t){
        this.transition=t;
        t.addGuard(this);
}
}
```

ProtocolViolationException.java

```
public class ProtocolViolationException extends Exception{
public ProtocolViolationException(){
        super();
        System.out.println(""+"ProtocolViolationException");
}
}
```

BahnÜbergang example code :

TrafficLight.java

```
public class TrafficLight {
// Define Statemachine
StateMachine trafficLight=null;
// Define States
public static final StateVertex DEACTIVATED=new SimpleState("DEACTIVATED");
public static final StateVertex ACTIVATED= new CompositeState("ACTIVATED");//no parent
public static final StateVertex ON = new CompositeState("ON",ACTIVATED);
public static final StateVertex OFF= new CompositeState("OFF",DEACTIVATED);
public static final StateVertex INITIALACTIVATED= new PseudoState("INITIALACTIVATED", "INITIAL");
public StateVertex currentState= DEACTIVATED;
public StateVertex historyState=null;
// Define Events
public Event switchOn=new Event("switchOn");
public Event switchOff=new Event("switchOff");
public Event activate=new Event("activate");
public Event deactivate=new Event("deactivate");
// Define Transitions
public Transition t1= new Transition(OFF,ON,switchOn);
public Transition t2= new Transition(ON,OFF,switchOff);
public Transition t3= new Transition(ACTIVATED,DEACTIVATED,deactivate);
public Transition t4= new Transition(DEACTIVATED,ACTIVATED,activate);
public Transition t5= new Transition(ACTIVATED,INITIALACTIVATED,activate);
public Transition t6= new Transition(INITIALACTIVATED,OFF,activate);

public TrafficLight(String name){
        trafficLight= new StateMachine(new State(name));
        // Assign Substates
        ON.addParent(ACTIVATED);
        OFF.addParent(ACTIVATED);
        ACTIVATED.addSubState(ON);
        ACTIVATED.addSubState(OFF);
        // Add the transitions generated by Events
```

```
        switchOn.addTransition(t1);
        switchOff.addTransition(t2);
        activate.addTransition(t4);
        activate.addTransition(t5);
        activate.addTransition(t6);
        deactivate.addTransition(t3);
}

public void executeEvent(Event e) throws ProtocolViolationException{
        Iterator it=e.transitions.iterator();
        while (it.hasNext()){
                Transition t= (Transition)it.next();
        if (t.source==currentState){
                        doTransition(t);
                        return;
                }
        }
        throw new ProtocolViolationException();
}

public void doTransition(Transition transition) throws ProtocolViolationException{

        onExit(transition.source); //executes onExit action for source State in transition
        currentState=transition.target;
        onEntry(transition.target); //executes onEntry action for target State in transition

}

public void onEntry(StateVertex state) throws ProtocolViolationException{
        if (state==ON){
                System.out.println("onEntry Action from TrafficLight."+state.getName());
                switchLightOn(state);
        }
        else if (state==OFF){
                System.out.println("onEntry Action from TrafficLight."+state.getName());
                switchLightOff(state);
        }
        else if (state==ACTIVATED){
                System.out.println("onEntry Action from TrafficLight."+state.getName());
                onEntryActivated(state);
        }
        else if (state==DEACTIVATED){
                System.out.println("onEntry Action from TrafficLight."+state.getName());
        }
        else if (state==INITIALACTIVATED){
                System.out.println("onEntry Action from TrafficLight."+state.getName());
        }
}

        private void onEntryActivated(StateVertex state) throws ProtocolViolationException{
                historyState= state;
                doTransition(t5);
                doTransition(t6);

        }

        private void switchLightOn(StateVertex state){
                System.out.println("switchLightOn() executed in State TrafficLight."+state.getName());
        }
        private void switchLightOff(StateVertex state){
                System.out.println("switchLightOff() executed in State TrafficLight."+state.getName());
        }

        public void onExit(StateVertex state){

                if (state==ON){
                        System.out.println("onExit Action from TrafficLight."+state.getName());
                }
                else if (state==OFF){
                        System.out.println("onExit Action from TrafficLight."+state.getName());
                }
                else if (state==ACTIVATED){
                        System.out.println("onExit Action from TrafficLight."+state.getName());
                }
                else if (state==DEACTIVATED){
                        System.out.println("onExit Action from TrafficLight."+state.getName());
                }
```

```
        }
}
```

Gate.java

```
public class Gate {
public static final int MAX_CLOSING_DURATION=10;
public static final int MAX_OPENING_DURATION=10;

// State declaration
StateMachine gate= null;
public static final StateVertex DEACTIVATED= new SimpleState("DEACTIVATED");
public static final StateVertex ACTIVATED= new CompositeState("ACTIVATED");
public static final StateVertex INITIALACTIVATED= new PseudoState("INITIALACTIVATED", "INITIAL");
public static final StateVertex OPENED= new CompositeState("OPENED", ACTIVATED);
public static final StateVertex CLOSED= new CompositeState("CLOSED", ACTIVATED);
public static final StateVertex OPENING= new CompositeState("OPENING", ACTIVATED);
public static final StateVertex CLOSING= new CompositeState("CLOSING", ACTIVATED);
public static final StateVertex FAULTY= new CompositeState("FAULTY", ACTIVATED);
public StateVertex currentState= DEACTIVATED;
public StateVertex historyState=null;

// Event declaration
public Event gateOpened= new Event("gateOpened");
public Event gateClosed= new Event("gateClosed");
public Event open= new Event("open");
public Event close= new Event("close");
public Event repaired= new Event("repaired");
public Event activate= new Event("activate");
public Event deactivate= new Event("deactivate");
public Event timeout= new Event("timeout");

// Transition declaration

public Transition t1= new Transition(DEACTIVATED,ACTIVATED,activate,null);
public Transition t2= new Transition(ACTIVATED,DEACTIVATED,deactivate,null);
public Transition t3= new Transition(ACTIVATED,INITIALACTIVATED,activate,null);
public Transition t4= new Transition(INITIALACTIVATED,CLOSED,activate,null);
public Transition t5= new Transition(OPENED,CLOSING,close,null);
public Transition t6= new Transition(CLOSING,CLOSED,gateClosed,null);
public Transition t7= new Transition(CLOSING,FAULTY,timeout,null);        // Transition generated
public Transition t8= new Transition(CLOSED,OPENING,open,null);
public Transition t9= new Transition(OPENING,OPENED,gateOpened,null);
public Transition t10= new Transition(OPENING,FAULTY,timeout,null);
public Transition t11= new Transition(FAULTY,DEACTIVATED,repaired,null);

// Constructor

public Gate(String name){
        gate= new StateMachine(new State(name));

        // Assign Substates
        ACTIVATED.addSubState(OPENED);
        ACTIVATED.addSubState(CLOSED);
        ACTIVATED.addSubState(OPENING);
        ACTIVATED.addSubState(CLOSING);
        ACTIVATED.addSubState(FAULTY);

        // Add the transitions generated by Events
        activate.addTransition(t1);
        deactivate.addTransition(t2);
        activate.addTransition(t3);
        activate.addTransition(t4);
        close.addTransition(t5);
        gateClosed.addTransition(t6);
        timeout.addTransition(t7);
        open.addTransition(t8);
        gateOpened.addTransition(t9);
        timeout.addTransition(t10);
        repaired.addTransition(t11);


}
public void executeEvent(Event e) throws ProtocolViolationException{
        Iterator it=e.transitions.iterator();
```

```
        while (it.hasNext()){
                Transition t= (Transition)it.next();
        if (t.source==currentState){
                        doTransition(t);
                        return;
                }
        }
        throw new ProtocolViolationException();
}

public void doTransition(Transition transition) throws ProtocolViolationException{

        onExit(transition.source); //executes onExit action for source State in transition
        currentState=transition.target;
        onEntry(transition.target); //executes onEntry action for target State in transition

}

public void onEntry(StateVertex state) throws ProtocolViolationException{
        if (state==OPENED){
                System.out.println("onEntry Action from Gate."+state.getName());
        }
        else if (state==CLOSED){
                System.out.println("onEntry Action from Gate."+state.getName());
        }
        else if (state==ACTIVATED){
                System.out.println("onEntry Action from Gate."+state.getName());
                doTransition(t3);
                doTransition(t4);
        }
        else if (state==OPENING){
                System.out.println("onEntry Action from Gate."+state.getName());
                startEngineUp();
                double randomValue = 3+ (Math.random() * 10);
                System.out.println(randomValue);
                if (randomValue<MAX_OPENING_DURATION)
                        doTransition(t9);
                else
                        doTransition(t10);

        }
        else if (state==CLOSING){
                System.out.println("onEntry Action from Gate."+state.getName());
                startEngineDown();
                double randomValue = 3+ (Math.random() * 10);
                System.out.println(randomValue);
                if (randomValue<MAX_CLOSING_DURATION)
                        doTransition(t6);
                else
                        doTransition(t7);

        }
        else if (state==FAULTY){
                System.out.println("onEntry Action from Gate."+state.getName());
        //      doTransition(t11);
        }
        else if (state==DEACTIVATED){
                System.out.println("onEntry Action from Gate."+state.getName());
        //      doTransition(t1);
        }
        else if (state==INITIALACTIVATED){
                System.out.println("onEntry Action from Gate."+state.getName());
        }

}
public void startEngineDown(){
        System.out.println("Gate.startEngineDown()");
}
public void startEngineUp(){
        System.out.println("Gate.startEngineUp()");
}
public void stopEngine(){
        System.out.println("Gate.stopEngine()");
}

public void onExit(StateVertex state){
```

```
        if (state==OPENED){
                System.out.println("onExit Action from Gate."+state.getName());
        }
        else if (state==CLOSED){
                System.out.println("onExit Action from Gate."+state.getName());
        }
        else if (state==FAULTY){
                System.out.println("onExit Action from Gate."+state.getName());
        }
        else if (state==OPENING){
                System.out.println("onExit Action from Gate."+state.getName());
                stopEngine();
        }
        else if (state==CLOSING){
                System.out.println("onExit Action from Gate."+state.getName());
                stopEngine();
        }
        else if (state==ACTIVATED){
                System.out.println("onExit Action from Gate."+state.getName());
        }
        else if (state==DEACTIVATED){
                System.out.println("onExit Action from Gate."+state.getName());
        }
        else if (state==INITIALACTIVATED){
                System.out.println("onExit Action from Gate."+state.getName());
        }
}
}
```

TrailObserver.java

```
public class TrailObserver {
public static final int MAX_PASSING_DURATION=30;
public StateMachine trailObserver=null;
//       State declaration
public static final StateVertex DEACTIVATED= new SimpleState("DEACTIVATED");
public static final StateVertex ACTIVATED= new CompositeState("ACTIVATED");
public static final StateVertex INITIALACTIVATED= new PseudoState("INITIALACTIVATED", "INITIAL");
public static final StateVertex IDLE= new CompositeState("IDLE", ACTIVATED);
public static final StateVertex TRAINPASSING= new CompositeState("TRAINPASSING", ACTIVATED);
public static final StateVertex FAULTY= new CompositeState("FAULTY", ACTIVATED);
public StateVertex currentState= DEACTIVATED;
public StateVertex historyState=null;

// Event declaration

public Event trainPassed=new Event("trainPassed");
public Event trainApproaching=new Event("trainApproaching");
public Event activate=new Event("activate");
public Event deactivate=new Event("deactivate");
public Event repaired=new Event("repaired");
public Event timeout=new Event("timeout");

// Transition Declaration

public Transition t1= new Transition(DEACTIVATED, ACTIVATED, activate,null);
public Transition t2= new Transition(ACTIVATED,DEACTIVATED,deactivate,null);
public Transition t3= new Transition(ACTIVATED,INITIALACTIVATED,activate,null);
public Transition t4= new Transition(INITIALACTIVATED,IDLE,activate,null);
public Transition t5= new Transition(IDLE,TRAINPASSING,trainApproaching,null);
public Transition t6= new Transition(TRAINPASSING,IDLE,trainPassed,null);
public Transition t7= new Transition(TRAINPASSING,FAULTY,trainApproaching,null);
public Transition t8= new Transition(TRAINPASSING,FAULTY,timeout,null);
public Transition t9= new Transition(IDLE,FAULTY,trainPassed,null);
public Transition t10= new Transition(FAULTY,DEACTIVATED,repaired,null);

public TrailObserver(String name){
        trailObserver=new StateMachine(new State(name));

        // Assign substates
        ACTIVATED.addSubState(IDLE);
        ACTIVATED.addSubState(TRAINPASSING);
        ACTIVATED.addSubState(FAULTY);

        // Add transitions generated by Events
```

```
        activate.addTransition(t1);
        deactivate.addTransition(t2);
        activate.addTransition(t3);
        activate.addTransition(t4);
        trainApproaching.addTransition(t5);
        trainPassed.addTransition(t6);
        trainApproaching.addTransition(t7);
        timeout.addTransition(t8);
        repaired.addTransition(t10);
}

public void executeEvent(Event e) throws ProtocolViolationException{
        Iterator it=e.transitions.iterator();
        while (it.hasNext()){
                Transition t= (Transition)it.next();
        if (t.source==currentState){
                        doTransition(t);
                        return;
                }
        }
        throw new ProtocolViolationException();
}

public void doTransition(Transition transition) throws ProtocolViolationException{

        onExit(transition.source);
        currentState=transition.target;
        onEntry(transition.target);
}

public void onEntry(StateVertex state) throws ProtocolViolationException{
        if (state==IDLE){
                System.out.println("onEntry Action from TrailObserver."+state.getName());
                boolean signalTrainPassed=false;
                boolean signalTrainApproaching=false;
                if (signalTrainPassed)
                        doTransition(t9);
                else if (signalTrainApproaching)
                        doTransition(t5);

        }
        else if (state==TRAINPASSING){
                System.out.println("onEntry Action from TrailObserver."+state.getName());
                double randomValue = 23+ (Math.random() * 10);
                boolean signalTrainApproaching=false;
                boolean signalTrainPassed=true;
                System.out.println(randomValue);
                if(signalTrainApproaching){
                        doTransition(t7);
                        return;
                }
                if (randomValue<MAX_PASSING_DURATION){
                        doTransition(t6);
                        return;
                }
                else if (signalTrainPassed)
                        doTransition(t8);
        }
        else if (state==ACTIVATED){
                System.out.println("onEntry Action from TrailObserver."+state.getName());
                doTransition(t3);
                doTransition(t4);
        }
        else if (state==FAULTY){
                System.out.println("onEntry Action from TrailObserver."+state.getName());
        }
        else if (state==DEACTIVATED){
                System.out.println("onEntry Action from TrailObserver."+state.getName());
        }
        else if (state==INITIALACTIVATED){
                System.out.println("onEntry Action from TrailObserver."+state.getName());
        }

}
public void onExit(StateVertex state) throws ProtocolViolationException{
```

```
        if (state==IDLE){
                System.out.println("onExit Action from TrailObserver."+state.getName());
        }
        else if (state==TRAINPASSING){
                System.out.println("onExit Action from TrailObserver."+state.getName());
        }
        else if (state==FAULTY){
                System.out.println("onExit Action from TrailObserver."+state.getName());
        }
        else if (state==ACTIVATED){
                System.out.println("onExit Action from TrailObserver."+state.getName());
        }
        else if (state==DEACTIVATED){
                System.out.println("onExit Action from TrailObserver."+state.getName());

        }
}
}
```

Bahn.java

```
public class Bahn {
StateMachine Bahn=null;
public static final int CAR_STOP_DURATION=30;
public TrafficLight trafficLight1=new TrafficLight("trafficLight1");
public TrafficLight trafficLight2=new TrafficLight("trafficLight2");
public Gate gate1=new Gate("Gate1");
public Gate gate2=new Gate("Gate2");
public TrailObserver trail1= new TrailObserver("TrailObserver1");
public TrailObserver trail2= new TrailObserver("TrailObserver2");


public static final StateVertex DEACTIVATED= new SimpleState("DEACTIVATED");
public static final StateVertex ACTIVATED= new CompositeState("ACTIVATED");
public static final StateVertex INITIALACTIVATED= new PseudoState("INITIALACTIVATED", "INITIAL");
public static final StateVertex IDLE= new CompositeState("IDLE", ACTIVATED);
public static final StateVertex CLOSEDFORCARS= new CompositeState("CLOSEDFORCARS", ACTIVATED);
public static final StateVertex PREPARINGTOCLOSE= new CompositeState("PREPARINGTOCLOSE", ACTIVATED);
public StateVertex currentState= DEACTIVATED;
public StateVertex historyState=null;

public Event activate=new Event("activate");
public Event deactivate=new Event("deactivate");
public Event change=new Event("change");    // Generated to follow the metamodel
public Event faulty=new Event("faulty");

public Transition t1= new Transition(DEACTIVATED, ACTIVATED, activate,null);
public Transition t2= new Transition(ACTIVATED,DEACTIVATED,deactivate,null);
public Transition t3= new Transition(ACTIVATED,INITIALACTIVATED,activate,null);
public Transition t4= new Transition(INITIALACTIVATED,IDLE,activate,null);
public Transition t5= new Transition(IDLE,PREPARINGTOCLOSE,change,null);
public Transition t6= new Transition(PREPARINGTOCLOSE,CLOSEDFORCARS,change,null);
public Transition t7= new Transition(CLOSEDFORCARS,IDLE,change,null);
public Transition t8= new Transition(ACTIVATED,DEACTIVATED,faulty,null);
public Transition t9= new Transition(CLOSEDFORCARS,DEACTIVATED,faulty,null);


public Bahn(String name) throws ProtocolViolationException{
        Bahn= new StateMachine(new State(name));

        ACTIVATED.addSubState(IDLE);
        ACTIVATED.addSubState(CLOSEDFORCARS);
        ACTIVATED.addSubState(PREPARINGTOCLOSE);

        activate.addTransition(t1);
        deactivate.addTransition(t2);
        activate.addTransition(t3);
        activate.addTransition(t4);
        change.addTransition(t5);
        change.addTransition(t6);
        change.addTransition(t7);
        faulty.addTransition(t8);
        faulty.addTransition(t9);
}
public void executeEvent(Event e) throws ProtocolViolationException,  InterruptedException{
        Iterator it=e.transitions.iterator();
```

```
        while (it.hasNext()){
                Transition t= (Transition)it.next();
        if (t.source==currentState){
                        doTransition(t);
                        return;
                }
        }
        throw new ProtocolViolationException();
}

public void doTransition(Transition transition) throws ProtocolViolationException,  InterruptedException{

        onExit(transition.source);
        currentState=transition.target;
        onEntry(transition.target);
}

public void onEntry(StateVertex state) throws ProtocolViolationException, InterruptedException{
        if (state==IDLE){
                System.out.println("onEntry Action from "+state.getName());
                while(trail1.currentState!=trail1.TRAINPASSING && trail2.currentState!=trail2.TRAINPASSING){
                        Thread.sleep(1000);
                        double randomValue = 3+ (Math.random() * 10);
                        System.out.println(randomValue);
                        if(checkFaulty())
                                doTransition(t2);
                        if (randomValue>7)
                                trail1.executeEvent(trail1.trainApproaching);
                        else
                                trail2.executeEvent(trail2.trainApproaching);
                }
                if(checkFaulty())
                        doTransition(t2);
                doTransition(t5);
        }
        else if (state==PREPARINGTOCLOSE){
                System.out.println("onEntry Action from "+state.getName());
                Thread.sleep(1000);
                if (gate1.currentState==gate1.OPENED)
                        gate1.executeEvent(gate1.close);
                if (gate2.currentState==gate2.OPENED)
                        gate2.executeEvent(gate2.close);
                if(checkFaulty()){
                        doTransition(t2);
                }
            doTransition(t6);;
        }
        else if (state==ACTIVATED){
                System.out.println("onEntry Action from "+state.getName());
                if (trail1.currentState==trail1.FAULTY)
                        trail1.executeEvent(trail1.repaired);
                if (trail2.currentState==trail2.FAULTY)
                        trail2.executeEvent(trail2.repaired);
                if (gate1.currentState==gate1.FAULTY)
                        gate1.executeEvent(gate1.repaired);
                if (trafficLight1.currentState==trafficLight1.DEACTIVATED)
                        trafficLight1.executeEvent(trafficLight1.activate);
                if (trafficLight2.currentState==trafficLight2.DEACTIVATED)
                        trafficLight2.executeEvent(trafficLight2.activate);
                if (trail1.currentState==trail1.DEACTIVATED)
                        trail1.executeEvent(trail1.activate);
                if (trail2.currentState==trail2.DEACTIVATED)
                        trail2.executeEvent(trail2.activate);
                if (gate1.currentState==gate1.DEACTIVATED)
                        gate1.executeEvent(gate1.activate);
                if (gate2.currentState==gate2.DEACTIVATED)
                        gate2.executeEvent(gate2.activate);
                if(checkFaulty()){
                        doTransition(t2);
                }
                doTransition(t3);
                doTransition(t4);
        }
        else if (state==CLOSEDFORCARS){
                System.out.println("onEntry Action from "+state.getName());

                if (gate1.currentState==gate1.CLOSED)
```

```
                // ensure that both gates are closed
                gate1.executeEvent(gate1.open);
        if ( gate2.currentState==gate2.CLOSED)
                gate2.executeEvent(gate2.open);
        if(checkFaulty()){
                doTransition(t2);
        }
        while(gate1.currentState!=gate1.OPENED && gate2.currentState!=gate2.OPENED){

                // if one door is faulty, then move to deactivated
                if(gate1.currentState==gate1.FAULTY || gate2.currentState==gate2.FAULTY){
                        doTransition(t9);
                }
                Thread.sleep(1000);
        }
        if (trafficLight1.currentState==trafficLight1.ON)
                trafficLight1.executeEvent(trafficLight1.switchOff);
        if (trafficLight2.currentState==trafficLight2.ON)
                trafficLight2.executeEvent(trafficLight2.switchOff);
        doTransition(t7);
        //return;
    }
    else if (state==DEACTIVATED){
        System.out.println("onEntry Action from "+state.getName());
        if (trail1.currentState==trail1.FAULTY)
                trail1.executeEvent(trail1.repaired);
        if (trail2.currentState==trail2.FAULTY)
                trail2.executeEvent(trail2.repaired);
        if (gate1.currentState==gate1.FAULTY)
                gate1.executeEvent(gate1.repaired);
        if (gate2.currentState==gate2.FAULTY)
                gate2.executeEvent(gate2.repaired);
        if (gate1.currentState==gate1.OPENED)
                gate1.executeEvent(gate1.close);
        if (gate2.currentState==gate2.OPENED)
                gate2.executeEvent(gate2.close);
        if(trafficLight1.currentState==trafficLight1.ON)
                trafficLight1.executeEvent(trafficLight1.switchOff);
        if(trafficLight2.currentState==trafficLight2.ON)
                trafficLight2.executeEvent(trafficLight2.switchOff);
        if(checkFaulty()){
                doTransition(t2);

        }
        doTransition(t1);
    }
    else if (state==INITIALACTIVATED){
        System.out.println("onEntry Action from "+state.getName());
    }

}
public boolean checkFaulty(){
    if (trail1.currentState==trail1.FAULTY || trail2.currentState==trail2.FAULTY ||
                gate1.currentState==gate1.FAULTY || gate2.currentState==gate2.FAULTY )
        return true;
    return false;
}

public void onExit(StateVertex state){

    if (state==IDLE){
        System.out.println("onExit Action from "+state.getName());
    }
    else if (state==PREPARINGTOCLOSE){
        System.out.println("onExit Action from "+state.getName());
    }
    else if (state==CLOSEDFORCARS){
        System.out.println("onExit Action from "+state.getName());
    }
    else if (state==ACTIVATED){
        System.out.println("onExit Action from "+state.getName());
    }
    else if (state==DEACTIVATED){
        System.out.println("onExit Action from "+state.getName());
    }
}
```

```
public static void main(String [] args)throws ProtocolViolationException,  InterruptedException {
	try {
		System.out.println("Begin test1() <<<<<<<<");
		Bahn o= new Bahn("Bahn Übergang");
		o.executeEvent(o.activate);
		o.executeEvent(o.deactivate);
		o.executeEvent(o.activate);
		o.executeEvent(o.change);
		//o.executeEvent(o.deactivate);
		System.out.println("End test1() >>>>>>>>");
	} catch (ProtocolViolationException e) {
		e.printStackTrace();
	}
}
}
```

Elevator example code :

Door.java

```
public class Door {
public static final int MAX_CLOSING_DURATION=10;
public static final int MAX_OPENING_DURATION=10;

// State declaration
StateMachine door= null;
public static final StateVertex DEACTIVATED= new SimpleState("DEACTIVATED");
public static final StateVertex ACTIVATED= new CompositeState("ACTIVATED");
public static final StateVertex INITIALACTIVATED= new PseudoState("INITIALACTIVATED", "INITIAL");
public static final StateVertex OPENED= new CompositeState("OPENED",ACTIVATED);
public static final StateVertex CLOSED= new CompositeState("CLOSED",ACTIVATED);
public static final StateVertex OPENING= new CompositeState("OPENING",ACTIVATED);
public static final StateVertex CLOSING= new CompositeState("CLOSING",ACTIVATED);
public static final StateVertex FAULTY= new CompositeState("FAULTY",ACTIVATED);
public StateVertex currentState= DEACTIVATED;
public StateVertex historyState=null;

//      Event declaration
public Event gateOpened= new Event("gateOpened");
public Event gateClosed= new Event("gateClosed");
public Event open= new Event("open");
public Event close= new Event("close");
public Event repaired= new Event("repaired");
public Event activate= new Event("activate");
public Event deactivate= new Event("deactivate");
public Event timeout= new Event("timeout");
public Event personDetected= new Event ("personDetected");

// Transition declaration

public Transition t1= new Transition(DEACTIVATED,ACTIVATED,activate,null);
public Transition t2= new Transition(ACTIVATED,DEACTIVATED,deactivate,null);
public Transition t3= new Transition(ACTIVATED,INITIALACTIVATED,activate,null);
public Transition t4= new Transition(INITIALACTIVATED,CLOSED,activate,null);
public Transition t5= new Transition(OPENED,CLOSING,close,null);
public Transition t6= new Transition(CLOSING,CLOSED,gateClosed,null);
public Transition t7= new Transition(CLOSING,FAULTY,timeout,null);        // Transition generated
public Transition t8= new Transition(CLOSED,OPENING,open,null);
public Transition t9= new Transition(OPENING,OPENED,gateOpened,null);
public Transition t10= new Transition(OPENING,FAULTY,timeout,null);
public Transition t11= new Transition(FAULTY,DEACTIVATED,repaired,null);
public Transition t12= new Transition(CLOSING,OPENING,personDetected,null);

// Contructor
public Door(String name){
	door=new StateMachine(new State(name));
}

public void executeEvent(Event e) throws ProtocolViolationException{
	Iterator it=e.transitions.iterator();
	while (it.hasNext()){
		Transition t= (Transition)it.next();
	if (t.source==currentState){
```

```
                doTransition(t);
                return;
            }
        }
        throw new ProtocolViolationException();
}

public boolean executeGuard(Transition t){
        return true;
}

public void doTransition(Transition transition) throws ProtocolViolationException{
            onExit(transition.source); //executes onExit action for source State in transition
            currentState=transition.target;
            onEntry(transition.target); //executes onEntry action for target State in transition
}

public void onEntry(StateVertex state) throws ProtocolViolationException{
        if (state==OPENED){
            System.out.println("onEntry Action from Door."+state.getName());
        }
        else if (state==CLOSED){
            System.out.println("onEntry Action from Door."+state.getName());
        }
        else if (state==ACTIVATED){
            System.out.println("onEntry Action from Door."+state.getName());
            doTransition(t3);
            doTransition(t4);
        }
        else if (state==OPENING){
            System.out.println("onEntry Action from Door."+state.getName());
            double randomValue = 3+ (Math.random() * 10);
            System.out.println(randomValue);
            if (randomValue<MAX_OPENING_DURATION){
                doTransition(t9);
                return;
            }
            else{
                doTransition(t10);
                return;
            }
        }
        else if (state==CLOSING){
            System.out.println("onEntry Action from Door."+state.getName());
            double randomValue = 3+ (Math.random() * 10);
            System.out.println(randomValue);
            if (randomValue>=MAX_CLOSING_DURATION){
                doTransition(t7);
                return;
            }
            else if (randomValue<MAX_CLOSING_DURATION && randomValue >5){
                doTransition(t6);
                return;
            }
            else                            // Person detected
                doTransition(t12);
            return;

        }
        else if (state==FAULTY){
            System.out.println("onEntry Action from Door."+state.getName());
        }
        else if (state==DEACTIVATED){
            System.out.println("onEntry Action from Door."+state.getName());
        }
        else if (state==INITIALACTIVATED){
            System.out.println("onEntry Action from Door."+state.getName());
        }

}

public void onExit(StateVertex state){

        if (state==OPENED){
            System.out.println("onExit Action from Gate."+state.getName());
        }
        else if (state==CLOSED){
```

```
		System.out.println("onExit Action from Gate."+state.getName());
	}
	else if (state==FAULTY){
		System.out.println("onExit Action from Gate."+state.getName());
	}
	else if (state==OPENING){
		System.out.println("onExit Action from Gate."+state.getName());
	}
	else if (state==CLOSING){
		System.out.println("onExit Action from Gate."+state.getName());
	}
	else if (state==ACTIVATED){
		System.out.println("onExit Action from Gate."+state.getName());
	}
	else if (state==DEACTIVATED){
		System.out.println("onExit Action from Gate."+state.getName());
	}
	else if (state==INITIALACTIVATED){
		System.out.println("onExit Action from Gate."+state.getName());
	}
}
}
```

Button.java

```
public class Button {

public StateMachine button= null;
public static final StateVertex PRESSED= new SimpleState("PRESSED");
public static final StateVertex NOTPRESSED= new SimpleState("NOTPRESSED");
public static final StateVertex INITIAL= new PseudoState("INITIAL", "INITIAL");
public StateVertex currentState= NOTPRESSED;
public StateVertex historyState=null;
// Event Declaration
public Event pressButton=new Event("pressButton");
public Event resetButton=new Event("resetButton");
public Event initial=new Event("initial");
// Transition Declaration
public Transition t1 = new Transition(NOTPRESSED,PRESSED,pressButton,null);
public Transition t2 = new Transition(PRESSED,NOTPRESSED,resetButton,null);
public Transition t3 = new Transition(INITIAL,NOTPRESSED,initial,null);

public Button(String name){
	button= new StateMachine(new State(name));
}
public void executeEvent(Event e) throws ProtocolViolationException{
	Iterator it=e.transitions.iterator();
	while (it.hasNext()){
		Transition t= (Transition)it.next();
	if (t.source==currentState){
			doTransition(t);
			return;
		}
	}
	throw new ProtocolViolationException();
}
public void doTransition(Transition transition) throws ProtocolViolationException{

	onExit(transition.source);
	currentState=transition.target;
	onEntry(transition.target);
}

public void onEntry(StateVertex state) throws ProtocolViolationException{
	if (state==PRESSED){
		System.out.println("onEntry Action from Button."+state.getName());
	}
	else if (state==NOTPRESSED){
		System.out.println("onEntry Action from Button."+state.getName());
	}
}
public void onExit(StateVertex state) throws ProtocolViolationException{
	if (state==PRESSED){
		System.out.println("onExit Action from Button."+state.getName());
	}
```

```
        else if (state==NOTPRESSED){
                System.out.println("onExit Action from Button."+state.getName());
        }
}
}
```

Elevator.java

```
public class Elevator {
public Door door=new Door("Door");
public Button[] buttons= new Button[10];
// State declaration
public StateMachine elevator= null;
public static final StateVertex DEACTIVATED= new SimpleState("DEACTIVATED");
public static final StateVertex ACTIVATED= new CompositeState("ACTIVATED");
public static final StateVertex MOVINGUP= new CompositeState("MOVINGUP", ACTIVATED);
public static final StateVertex MOVINGDOWN= new CompositeState("MOVINGDOWN", ACTIVATED);
public static final StateVertex INITIALACTIVATED= new PseudoState("INITIALACTIVATED", "INITIAL");
public static final StateVertex MOVING= new PseudoState("MOVING", "CHOICE");
public static final StateVertex IDLE= new CompositeState("IDLE",ACTIVATED);
public StateVertex currentState= DEACTIVATED;
public StateVertex historyState=MOVINGUP; //Starts  on floor 0
public int currentFloor=0;
public int nextFloor=0;

// Events declaration
public Event activate=new Event("activate");
public Event deactivate=new Event("deactivate");
public Event faulty=new Event("faulty");
public Event floorReached=new Event("floorReached");
public Event pressButton= new Event("pressButton");

// Guards declaration
public Guard movingUpGuard= null;
public Guard movingDownGuard= null;
public Guard buttonPressedGuard= null;

// Transitions declaration
public Transition t1= new Transition(DEACTIVATED, ACTIVATED, activate);
public Transition t2= new Transition(ACTIVATED,DEACTIVATED,deactivate);
public Transition t3= new Transition(ACTIVATED,INITIALACTIVATED,activate);
public Transition t4= new Transition(INITIALACTIVATED,IDLE,activate);
public Transition t7= new Transition(MOVINGUP,IDLE,floorReached);
public Transition t8= new Transition(MOVINGDOWN,IDLE,floorReached);
public Transition t9=new Transition(IDLE,MOVING,pressButton,buttonPressedGuard);
public Transition t10=new Transition(MOVING,MOVINGUP,pressButton,movingUpGuard);
public Transition t11=new Transition(MOVING,MOVINGDOWN,pressButton,movingDownGuard);


public Elevator(String name){
        elevator= new StateMachine(new State(name));
        // initialize Guards on transitions
        buttonPressedGuard=new Guard(t9);
        movingUpGuard= new Guard (t10);
        movingDownGuard= new Guard (t11);
        // Assign buttons for each floor
        for (int i=0;i<10;i++){
                buttons[i]=new Button("Button"+i);
        }
}

public void executeEvent(Event e) throws ProtocolViolationException, InterruptedException{
        Iterator it=e.transitions.iterator();
        while (it.hasNext()){
                Transition t= (Transition)it.next();
                if (t.source==currentState){
                        doTransition(t);
                        return;
                }
        }
        throw new ProtocolViolationException();
}

public boolean executeGuard(Guard guard){
```

```
        boolean result=false;
        if (guard==movingUpGuard){
                if (door.currentState==door.CLOSED&&nextFloor>currentFloor)
                        result=true;
                else
                        result=false;
        }
        else if (guard==movingDownGuard){
                if (door.currentState==door.CLOSED&&nextFloor<currentFloor)
                        result=true;
                else
                        result=false;
        }
        else if (guard==buttonPressedGuard){
                if (currentFloor!=getNextFloorUp()||currentFloor!=getNextFloorDown())
                        result=true;
                else
                        result=false;
        }
        return result;
}

public void doTransition(Transition transition) throws ProtocolViolationException, InterruptedException{
        if (executeGuard(transition.guard)||transition.guard==null){
                onExit(transition.source);
                currentState=transition.target;
                onEntry(transition.target);
        }
        else
                System.out.println("Transition not executed, Guard=false");
}
public int getNextFloorUp(){
        for (int i =currentFloor;i<=9;i++){
                if (buttons[i].currentState==Button.PRESSED)
                        return i;
        }return currentFloor;
}
public int getNextFloorDown(){
        for (int i =currentFloor;i>=0;i--){
                if (buttons[i].currentState==Button.PRESSED)
                        return i;
        }return currentFloor;
}

public void onEntry(StateVertex state) throws ProtocolViolationException, InterruptedException{
        if (state==IDLE){
                System.out.println("onEntry Action from Elevator."+state.getName());
                Thread.sleep(1000);
                        if((historyState==MOVINGUP&&getNextFloorUp()>currentFloor)||(historyState==MOVINGDOWN&&
                                        getNextFloorUp()>currentFloor&&getNextFloorDown()==currentFloor)){
                                nextFloor=getNextFloorUp();
                                historyState=MOVINGUP;
                                doTransition(t9);
                                return;
                        }
                        else
if((historyState==MOVINGDOWN&&getNextFloorDown()<currentFloor)||(historyState==MOVINGUP&&
                                        getNextFloorDown()<currentFloor&&getNextFloorUp()==currentFloor)){
                                nextFloor=getNextFloorDown();
                                historyState=MOVINGDOWN;
                                doTransition(t9);
                                return;
                        }
                        else
                                System.out.println("No Button Pressed");
        }
        else if (state==ACTIVATED){
                System.out.println("onEntry Action from Elevator."+state.getName());
                doTransition(t3);
                doTransition(t4);
        }
        else if (state==MOVING){
                System.out.println("onEntry Action from Elevator."+state.getName());
                Thread.sleep(1000);
                if (historyState==MOVINGUP){
                        System.out.println("Current Floor = "+currentFloor);
                        System.out.println("Next Floor Up= "+nextFloor);
```

```
                doTransition(t10);
                return;
            }
            else if (historyState==MOVINGDOWN){
                System.out.println("Current Floor = "+currentFloor);
                System.out.println("Next Floor Down= "+nextFloor);
                doTransition(t11);
                return;
            }
        }
        else if (state==MOVINGUP){
            System.out.println("onEntry Action from Elevator."+state.getName());
            Thread.sleep(1000);
            currentFloor=nextFloor;
            executeEvent(floorReached);
        }
        else if (state==MOVINGDOWN){
            System.out.println("onEntry Action from Elevator."+state.getName());
            Thread.sleep(1000);
            currentFloor=nextFloor;
            executeEvent(floorReached);

        }
        else if (state==DEACTIVATED){
            System.out.println("onEntry Action from Elevator."+state.getName());
        }
}
public void onExit(StateVertex state) throws ProtocolViolationException, InterruptedException{
        if (state==IDLE){
            System.out.println("onExit Action from Elevator."+state.getName());
            if((historyState==MOVINGUP&&getNextFloorUp()>currentFloor)||(historyState==MOVINGDOWN&&
                    getNextFloorUp()>currentFloor&&getNextFloorDown()==currentFloor)){
                nextFloor=getNextFloorUp();
                historyState=MOVINGUP;
                if (door.currentState==door.CLOSED){
                    door.executeEvent(door.open);
                    System.out.println("Opening door...");
                }
                Thread.sleep(500);
                if (door.currentState==door.OPENED){
                    door.executeEvent(door.close);
                    System.out.println("Closing door...");
                }
            }
            else if((historyState==MOVINGDOWN&&getNextFloorDown()<currentFloor)||(historyState==MOVINGUP&&
                    getNextFloorDown()<currentFloor&&getNextFloorUp()==currentFloor)){
                nextFloor=getNextFloorDown();
                historyState=MOVINGDOWN;
                if (door.currentState==door.CLOSED){
                    door.executeEvent(door.open);
                    System.out.println("Opening door...");
                }
                Thread.sleep(500);
                if (door.currentState==door.OPENED){
                    door.executeEvent(door.close);
                    System.out.println("Closing door...");
                }
            }

        }
        else if (state==ACTIVATED){
            System.out.println("onExit Action from Elevator."+state.getName());
        }
        else if (state==MOVINGUP){
            System.out.println("onExit Action from Elevator."+state.getName());
            if (buttons[currentFloor].currentState==buttons[currentFloor].PRESSED){
                buttons[currentFloor].executeEvent(buttons[currentFloor].resetButton);
                System.out.println("Button "+currentFloor+" reseted");
            }
        }
        else if (state==MOVINGDOWN){
            System.out.println("onExit Action from Elevator."+state.getName());
            if (buttons[currentFloor].currentState==buttons[currentFloor].PRESSED){
                buttons[currentFloor].executeEvent(buttons[currentFloor].resetButton);
                System.out.println("Button "+currentFloor+" reseted");
            }
        }
```

```
        else if (state==MOVING){
                System.out.println("onExit Action from Elevator."+state.getName());
        }
        else if (state==DEACTIVATED){
                System.out.println("onExit Action from Elevator."+state.getName());
        }
}
public static void main (String[] args)throws ProtocolViolationException, InterruptedException{
        Elevator e=new Elevator("Elevator");
        e.door.executeEvent(e.door.activate);
        e.executeEvent(e.activate);
        e.buttons[5].executeEvent(e.buttons[5].pressButton);
        e.buttons[8].executeEvent(e.buttons[8].pressButton);
        e.executeEvent(e.pressButton);
        e.buttons[2].executeEvent(e.buttons[2].pressButton);
        e.executeEvent(e.pressButton);
        e.executeEvent(e.pressButton);
}
}
```

Files used to generate metamodel and Elevator example with Octopus:

model.uml

```
<package> test

<class> ProtocolViolationException
<endclass>

<class> StateMachine
<endclass>

<class> StateVertex
<attributes>
        + name: String;
        + transitions: Set(Transition);
<operations>
        + addParent(s: StateVertex);
        + addSubState(s: StateVertex);
        + getSubStates(): Set(StateVertex);
<endclass>

<class> Transition
<endclass>

<class> Event
<attributes>
        + name: String;
<endclass>

<class> Guard
<endclass>

<class> State <specializes> StateVertex
<endclass>

<class> PseudoState <specializes> StateVertex
<attributes>
+ kind: String;
<endclass>

<class> SimpleState <specializes> State
<endclass>

<class> CompositeState <specializes> State
<endclass>

<associations>
+ StateMachine.statemachine[0..1] <-> State.top[1];
+ Transition.<noName>[0..*] -> + StateVertex.source[1];
+ Transition.<noName>[0..*] -> + StateVertex.target[1];
+ Transition.transition[1] <-> + Guard.guard[0..1];
+ Transition.transitions[0..*] <-> + Event.trigger[0..1];
```

```
+ State.<noName>[0..*] -> + Transition.internalTransition[1];
+ State.states[0..*] <-> + Event.defferedEvent[0..*];
+ StateVertex.substates[0..*] <-> CompositeState.parent[1];
<endpackage>
```

Elevator.uml

```
<package> test

<class> Door
<attributes>
        + MAX_CLOSING_DURATION: Integer;
        + MAX_OPENING_DURATION: Integer;
        + statemachine: StateMachine;
        + DEACTIVATED: StateVertex;
        + ACTIVATED: StateVertex;
        + INITIALACTIVATED: StateVertex;
        + OPENED: StateVertex;
        + CLOSED: StateVertex;
        + OPENING: StateVertex;
        + CLOSING: StateVertex;
        + FAULTY: StateVertex;
        + currentState: StateVertex;
        + gateOpened: Event;
        + gateClosed: Event;
        + open: Event;
        + close: Event;
        + repaired: Event;
        + activate: Event;
        + deactivate: Event;
        + timeout: Event;
        + personDetected: Event;
        + t1: Transition;
        + t2: Transition;
        + t3: Transition;
        + t4: Transition;
        + t5: Transition;
        + t6: Transition;
        + t7: Transition;
        + t8: Transition;
        + t9: Transition;
        + t10: Transition;
        + t11: Transition;
        + t12: Transition;
<operations>
        + executeEvent(e: Event);
        + executeGuard(g: Guard): Boolean;
        + doTransition(t: Transition);
        + onEntry(state: StateVertex);
        + onExit(state: StateVertex);
<endclass>

<class> Button
<attributes>
        + statemachine: StateMachine;
        + PRESSED: StateVertex;
        + NOTPRESSED: StateVertex;
        + INITIAL: StateVertex;
        + currentState: StateVertex;
        + pressButton: Event;
        + resetButton: Event;
        + initial: Event;
        + t1: Transition;
        + t2: Transition;
        + t3: Transition;
        <operations>
        + executeEvent(e: Event);
        + executeGuard(g: Guard): Boolean;
        + doTransition(t: Transition);
        + onEntry(state: StateVertex);
        + onExit(state: StateVertex);
<endclass>

<class> Elevator
<attributes>
```

```
        + statemachine: StateMachine;
        + door: Door;
        + DEACTIVATED: StateVertex;
        + ACTIVATED: StateVertex;
        + INITIALACTIVATED: StateVertex;
        + MOVINGUP: StateVertex;
        + MOVINGDOWN: StateVertex;
        + MOVING: StateVertex;
        + IDLE: StateVertex;
        + historyState: StateVertex;
        + currentState: StateVertex;
        + currentFloor: Integer;
        + nextFloor: Integer;
        + activate: Event;
        + deactivate: Event;
        + faulty: Event;
        + floorReached: Event;
        + pressButton: Event;
        + movingUpGuard: Guard;
        + movingDownGuard: Guard;
        + buttonPressedGuard: Guard;
        + t1: Transition;
        + t2: Transition;
        + t3: Transition;
        + t4: Transition;
        + t5: Transition;
        + t6: Transition;
        + t7: Transition;
        + t8: Transition;
        + t9: Transition;
        + t10: Transition;
        + t11: Transition;
        <operations>
        + getUpNextFloor(): Integer;
        + getDownNextFloor(): Integer;
        + executeEvent(e: Event);
        + executeGuard(g: Guard): Boolean;
        + doTransition(t: Transition);
        + onEntry(state: StateVertex);
        + onExit(state: StateVertex);
<endclass>

<endpackage>
```

Elevator.ocl

```
package test

context Elevator:: currentState: StateVertex
init: DEACTIVATED

context Elevator:: historyState: StateVertex
init: MOVINGUP

context Elevator:: currentFloor: Integer
init: 0

context Elevator:: nextFloor: Integer
init: 0

context Door:: currentState: StateVertex
init: DEACTIVATED

context Door:: MAX_CLOSING_DURATION: Integer
init: 10

context Door:: MAX_OPENING_DURATION: Integer
init: 10

endpackage
```

Adapted Elevator class generated with Octopus:

Elevator.java (partially)

```
public class Elevator {
private StateMachine f_statemachine = null;
public Button[] buttons= new Button[10];
public Door door=new Door("Door");
private StateVertex f_dEACTIVATED = new SimpleState("DEACTIVATED");
private StateVertex f_aCTIVATED = new CompositeState("ACTIVATED");
private StateVertex f_iNITIALACTIVATED = new PseudoState("INITIALACTIVATED", "INITIAL");
private StateVertex f_mOVINGUP =  new CompositeState("MOVINGUP", f_aCTIVATED);
private StateVertex f_mOVINGDOWN = new CompositeState("MOVINGDOWN", f_aCTIVATED);
private StateVertex f_mOVING = new PseudoState("MOVING", "CHOICE");
private StateVertex f_iDLE = new CompositeState("IDLE",f_aCTIVATED);
private StateVertex f_historyState = null;
private StateVertex f_currentState = f_dEACTIVATED;
private int f_currentFloor = 0;
private int f_nextFloor = 0;
private Event f_activate = new Event("activate");
private Event f_deactivate = new Event("deactivate");
private Event f_faulty = new Event("faulty");
private Event f_floorReached = new Event("floorReached");
private Event f_pressButton = new Event("pressButton");
private Guard f_movingUpGuard = null;
private Guard f_movingDownGuard = null;
private Guard f_buttonPressedGuard = null;
private Transition f_t1 = new Transition(f_dEACTIVATED, f_aCTIVATED, f_activate);
private Transition f_t2 = new Transition(f_aCTIVATED,f_dEACTIVATED,f_deactivate);
private Transition f_t3 = new Transition(f_aCTIVATED,f_iNITIALACTIVATED,f_activate);
private Transition f_t4 =  new Transition(f_iNITIALACTIVATED,f_iDLE,f_activate);
private Transition f_t7 = new Transition(f_mOVINGUP,f_iDLE,f_floorReached);
private Transition f_t8 = new Transition(f_mOVINGDOWN,f_iDLE,f_floorReached);
private Transition f_t9 = new Transition(f_iDLE,f_mOVING,f_pressButton,f_buttonPressedGuard);
private Transition f_t10 =new Transition(f_mOVING,f_mOVINGUP,f_pressButton,f_movingUpGuard);
private Transition f_t11 =new Transition(f_mOVING,f_mOVINGDOWN,f_pressButton,f_movingDownGuard);
static private boolean usesAllInstances = false;
static private List allInstances = new ArrayList();

/** Constructor for Elevator
 *
 * @param currentFloor
 * @param nextFloor
 */
public Elevator(int currentFloor, int nextFloor) {
	super();
	this.setCurrentFloor(currentFloor);
	this.setNextFloor(nextFloor);
	this.setHistoryState( this.getMOVINGUP() );
	this.setCurrentState( this.getDEACTIVATED() );
	this.setCurrentFloor( 0 );
	this.setNextFloor( 0 );
	if ( usesAllInstances ) {
		allInstances.add(this);
	}
}

/** Default constructor for Elevator
 */
public Elevator(String name) {
	f_statemachine=new StateMachine(new State(name));
	f_buttonPressedGuard=new Guard(f_t9);
	f_movingUpGuard= new Guard (f_t10);
	f_movingDownGuard= new Guard (f_t11);
	this.setHistoryState( this.getMOVINGUP() );
	this.setCurrentState( this.getDEACTIVATED() );
	this.setCurrentFloor( 0 );
	this.setNextFloor( 0 );

	for (int i=0;i<10;i++){
		buttons[i]=new Button("Button"+i);
	}
	if ( usesAllInstances ) {
		allInstances.add(this);
	}
}

/** Implements the user defined operation '+ executeEvent( e: Event )'
```

```
 *
 * @param e
 */
public void executeEvent(Event e) throws ProtocolViolationException, InterruptedException{
	Iterator it=e.getTransitions().iterator();
	while (it.hasNext()){
		Transition t= (Transition)it.next();
		if (t.getSource()==f_currentState){
			doTransition(t);
			return;
		}
	}
	throw new ProtocolViolationException();
}

/** Implements the user defined operation '+ executeGuard( g: Guard ) : Boolean'
 *
 * @param g
 */
public boolean executeGuard(Guard g){
	boolean result=false;
	if (g==f_movingUpGuard){
		if (door.getCurrentState()==door.getCLOSED()&&f_nextFloor>f_currentFloor)
			result=true;
		else
			result=false;
	}
	else if (g==f_movingDownGuard){
		if (door.getCurrentState()==door.getCLOSED()&&f_nextFloor<f_currentFloor)
			result=true;
		else
			result=false;
	}
	else if (g==f_buttonPressedGuard){
		if (f_currentFloor!=getNextFloorUp()||f_currentFloor!=getNextFloorDown())
			result=true;
		else
			result=false;
	}
	return result;
}

/** Implements the user defined operation '+ doTransition( t: Transition )'
 *
 * @param t
 */
public void doTransition(Transition transition) throws ProtocolViolationException, InterruptedException{
	if (executeGuard(transition.getGuard())||transition.getGuard()==null){
		onExit(transition.getSource()); //executes onExit action for source State in transition
		f_currentState=transition.getTarget();
		onEntry(transition.getTarget()); //executes onEntry action for target State in transition
	}
	else
		System.out.println("Transition not executed, Guard=false");
}
public int getNextFloorUp(){
	for (int i =f_currentFloor;i<=9;i++){
		if (buttons[i].getCurrentState()==buttons[i].getPRESSED())
			return i;

	}return f_currentFloor;
}
public int getNextFloorDown(){
	for (int i =f_currentFloor;i>=0;i--){
		if (buttons[i].getCurrentState()==buttons[i].getPRESSED())
			return i;

	}return f_currentFloor;
}
/** Implements the user defined operation '+ onEntry( state: StateVertex )'
 *
 * @param state
 */
public void onEntry(StateVertex state) throws ProtocolViolationException, InterruptedException{
	if (state==f_iDLE){
		System.out.println("onEntry Action from Elevator."+state.getName());
		Thread.sleep(1000);
```

```
if((f_historyState==f_mOVINGUP&&getNextFloorUp()>f_currentFloor)||(f_historyState==f_mOVINGDOWN&&
                                        getNextFloorUp()>f_currentFloor&&getNextFloorDown()==f_currentFloor)){
                                f_nextFloor=getNextFloorUp();
                                f_historyState=f_mOVINGUP;
                                doTransition(f_t9);
                                return;
                        }
                        else
if((f_historyState==f_mOVINGDOWN&&getNextFloorDown()<f_currentFloor)||(f_historyState==f_mOVINGUP&&
                                        getNextFloorDown()<f_currentFloor&&getNextFloorUp()==f_currentFloor)){
                                f_nextFloor=getNextFloorDown();
                                f_historyState=f_mOVINGDOWN;
                                doTransition(f_t9);
                                return;
                        }
                        else
                                System.out.println("No Button Pressed");
        }
        else if (state==f_aCTIVATED){
                System.out.println("onEntry Action from Elevator."+state.getName());
                doTransition(f_t3);
                doTransition(f_t4);
        }
        else if (state==f_mOVING){
                System.out.println("onEntry Action from Elevator."+state.getName());
                Thread.sleep(1000);
                if (f_historyState==f_mOVINGUP){
                        System.out.println("Current Floor = "+f_currentFloor);
                        System.out.println("Next Floor Up= "+f_nextFloor);
                        doTransition(f_t10);
                        return;
                }
                else if (f_historyState==f_mOVINGDOWN){
                        System.out.println("Current Floor = "+f_currentFloor);
                        System.out.println("Next Floor Down= "+f_nextFloor);
                        doTransition(f_t11);
                        return;
                }
        }
        else if (state==f_mOVINGUP){
                System.out.println("onEntry Action from Elevator."+state.getName());
                Thread.sleep(1000);
                f_currentFloor=f_nextFloor;
                executeEvent(f_floorReached);
        }
        else if (state==f_mOVINGDOWN){
                System.out.println("onEntry Action from Elevator."+state.getName());
                Thread.sleep(1000);
                f_currentFloor=f_nextFloor;
                executeEvent(f_floorReached);

        }
        else if (state==f_dEACTIVATED){
                System.out.println("onEntry Action from Elevator."+state.getName());
        }

}

/** Implements the user defined operation '+ onExit( state: StateVertex )'
 *
 * @param state
 */
public void onExit(StateVertex state) throws ProtocolViolationException, InterruptedException{

        if (state==f_iDLE){
                System.out.println("onExit Action from Elevator."+state.getName());

if((f_historyState==f_mOVINGUP&&getNextFloorUp()>f_currentFloor)||(f_historyState==f_mOVINGDOWN&&
                        getNextFloorUp()>f_currentFloor&&getNextFloorDown()==f_currentFloor)){
                f_nextFloor=getNextFloorUp();
                f_historyState=f_mOVINGUP;
                if (door.getCurrentState()==door.getCLOSED()){
                        door.executeEvent(door.getOpen());
                        System.out.println("Opening door...");
                }
                Thread.sleep(500);
```

```
                        if (door.getCurrentState()==door.getOPENED()){
                                door.executeEvent(door.getClose());
                                System.out.println("Closing door...");
                        }
                }
                else
if((f_historyState==f_mOVINGDOWN&&getNextFloorDown()<f_currentFloor)||(f_historyState==f_mOVINGUP&&
                                getNextFloorDown()<f_currentFloor&&getNextFloorUp()==f_currentFloor)){
                        f_nextFloor=getNextFloorDown();
                        f_historyState=f_mOVINGDOWN;
                        if (door.getCurrentState()==door.getCLOSED()){
                                door.executeEvent(door.getOpen());
                                System.out.println("Opening door...");
                        }
                        Thread.sleep(500);
                        if (door.getCurrentState()==door.getOPENED()){
                                door.executeEvent(door.getClose());
                                System.out.println("Closing door...");
                        }
                }

        }
        else if (state==f_aCTIVATED){
                System.out.println("onExit Action from Elevator."+state.getName());
        }
        else if (state==f_mOVINGUP){
                System.out.println("onExit Action from Elevator."+state.getName());
                if (buttons[f_currentFloor].getCurrentState()==buttons[f_currentFloor].getPRESSED()){
                        buttons[f_currentFloor].executeEvent(buttons[f_currentFloor].getResetButton());
                        System.out.println("Button "+f_currentFloor+" reseted");
                }
        }
        else if (state==f_mOVINGDOWN){
                System.out.println("onExit Action from Elevator."+state.getName());
                if (buttons[f_currentFloor].getCurrentState()==buttons[f_currentFloor].getPRESSED()){
                        buttons[f_currentFloor].executeEvent(buttons[f_currentFloor].getResetButton());
                        System.out.println("Button "+f_currentFloor+" reseted");
                }
        }
        else if (state==f_mOVING){
                System.out.println("onExit Action from Elevator."+state.getName());
        }
        else if (state==f_dEACTIVATED){
                System.out.println("onExit Action from Elevator."+state.getName());
        }
}


public static void main (String[] args)throws ProtocolViolationException, InterruptedException{
        Elevator e=new Elevator("Elevator");
        e.door.executeEvent(e.door.getActivate());
        e.executeEvent(e.getActivate());
        e.buttons[5].executeEvent(e.buttons[5].getPressButton());
        e.buttons[8].executeEvent(e.buttons[8].getPressButton());
        e.executeEvent(e.getPressButton());
        e.buttons[2].executeEvent(e.buttons[2].getPressButton());
        e.executeEvent(e.getPressButton());
        e.executeEvent(e.getPressButton());
}

}
```