Chapter 2

Checking BART Test Scenarios with UML's Object Constraint Language

M. Gogolla*,
P. Ziemann

Abstract The Object Constraint Language (OCL) is part of the Unified Modeling Language (UML). Within software engineering, UML is regarded today as an important step towards development of high-quality object-oriented systems. OCL allows to sharpen UML diagrams through invariants and pre- and postconditions. This paper explains the functionality of the UML Specification Environment USE which allows to validate and verify UML and OCL descriptions.
The paper shows that central safety properties of the BART system can be expressed with OCL. Test cases embodying central aspects of the BART system can be formulated within the USE system. It can be shown that the safety properties are satisfied by the test cases examined.

1. Introduction

The Unified Modeling Language (UML) [OMG03] is regarded today as a very important standard for the development of software systems. UML originated from its main predecessor approaches Booch [Boo94], OMT [RBP+91], and OOSE [JCJO92]. UML is a graphical modeling language supporting many phases in the software development cycle by offering diagrams and language features meeting the special needs in respective phases. Many commercial tools for UML are available. The Object Constraint Language (OCL) [OMG03, WK98] is part of standard UML, but up to now OCL is not supported by most commercial tools. OCL is a specification language supporting and enriching the UML with textual details which cannot be expressed in diagramatic form. OCL allows to precisely describe system structure by invariants and system behavior by pre- and postconditions.

*Chapter responsible.

Tool support for OCL is beginning to develop. Among the first available tools was our system USE (UML Specification Environment) [Ric01]. USE is based on conceptional work on the formal semantics of the OCL, the needed relevant UML features [RG01] and work on the metamodel of OCL [RG99]. The main task of USE is to validate and verify specifications developed within UML and OCL. By validation we mean that the developer can animate the specification by providing a number of test cases and check whether the USE responses meet the intuition. Verification is supported in restricted form [GR02]. Especially for specifications involving formal aspects we regard validation systems as extremely helpful because they give feedback to developers in early development stages. By verification we mean, that the test cases are formally checked with respect to invariants and pre- and postconditions. Other tools for OCL include the commercial tool from Boldsoft [Bol02], an OCL compiler [HDF00], and the KeY tool [ABB$^+$00].
USE is now available as version 2.0.1 and has achieved more and more functionality since it was first introduced. USE is the only OCL tool allowing interactive monitoring of OCL invariants and pre- and postconditions. USE has been successfully applied in various case studies and in teaching UML. It was employed in submissions for the upcoming new version of UML and has been used for large specifications like the UML metamodel with about 100 classes and hundreds of OCL expressions. USE offers the following functionalities:

1 syntax check for and browsing through textual UML and OCL descriptions,

2 generation of system states through object, attribute and link manipulation,

3 representation of system states as object diagrams,

4 monitoring model inherent and explicit invariants in class diagrams,

5 operation execution and monitoring of pre- and postconditions,

6 representation of operation call sequences as sequence diagrams, and

7 querying system states with OCL expressions.

For realizing these functionalities, the USE system covers central UML language features. The input to USE are UML class diagrams in textual form, OCL invariants, and OCL operation descriptions through pre- and postconditions. USE produces as output UML object and sequence diagrams as well as reports on the validity of invariants and pre- and postconditions in table and text form.

The central idea of the USE tool is to check for software quality criteria like correct functionality of UML descriptions already on the design level in an implementation-independent manner. This approach takes advantage of descriptive design level specifications by expressing properties shorter and in a more abstract way. Such properties are given by invariants and pre- and postconditions, and these are checked by the USE system against the test scenarios, i.e. object diagrams, state manipulation commands, and operation calls. Such abstract design level tests are expected to be also used later in the implementation phase.

A short version of this paper has been published in [ZG03]. The structure of the rest of this chapter is as follows. Section 2 introduces UML, OCL and the USE tool. In Section 3 we declare which elements of the case study we consider and which we do not. In the central Section 4 the specification is developed and validated. The results of our approach are given in Section 5 and finally our contribution closes with a conclusion in Section 6.

2. Technical approach and method

Let us start by giving an overview on this section. In Subsection 2.1 we explain the running example we use for introducing the UML Specification Environment USE. Subsection 2.2 shows how class diagrams and constraints are represented in USE. In Subsection 2.3 we elaborate how system states can be generated automatically. Subsection 2.4 discusses how to run USE and how to get feedback from it.

2.1 The crossing system

This introductory example describes a simple traffic control application where a controller is supposed to supervise four traffic lights on a street crossing. Figure 2.1 shows a UML class diagram with the classes `Controller` and `Light`. Figure 2.2 displays two UML statechart diagrams, the left one for class `Controller` and the right one for class `Light`.

A `Controller` object can be linked to the four `Light` objects which are then accessible with the role names `north`, `south`, `west`, and `east`. The controller possesses an operation `createLights` for creating and initializing the lights and an operation `switch` for switching the crossing system. The controller is responsible for transmitting a `switch` operation to the respective lights. A `Light` object possesses three attributes `r`, `y`, and `g` for the red, yellow, and green bulb, respectively, an internal operation `ryg` for manipulating the three attributes and also an operation `switch` for switching. Furthermore, for convenience, it has three Boolean-valued operations for checking the displayed signal.

The `Light` statechart shows that we describe Italian traffic lights with three states `G`, `YG`, and `R`. In state `G` only the green bulb is on, in state `YG` only the yellow and green bulb, and in state `R` only the red bulb. The `shows` predicates, i.e. the boolean-valued operations beginning with `shows`, characterize these states. The `switch` operation in class `Light` switches between these states in a cyclic way. The `Controller` statechart has four states which are named according to the signal on the `north` and `west` light. We require that the `north` and `south` lights coincide as well as the `west` and `east` lights. When receiving a `switch` operation, the task of the controller is to transmit the `switch` to the proper `Light` objects. Essential for our purpose is that certain properties have to be satisfied. For example, for safety reasons, the `north` and the `west` light are not allowed to display only green at the same time, and they are not allowed to show only red at the same time. The purpose of the UML Specification Environment (USE) is to bring life into this description.

Figure 2.1. Class Diagram for Crossing System.

2.2 Class diagram and constraint representation in USE

Class diagrams are given to USE in textual form. In the following, the classes `Controller` and `Light` are presented together with their attributes and operations. In addition to attribute types and operation signatures, the definition of the query operations (in this case, the `shows` predicates) are specified by OCL expressions.

```
model TrafficLight

class Controller
operations
  createLights()
  switch()
end

class Light
attributes
  r : Integer
  y : Integer
```

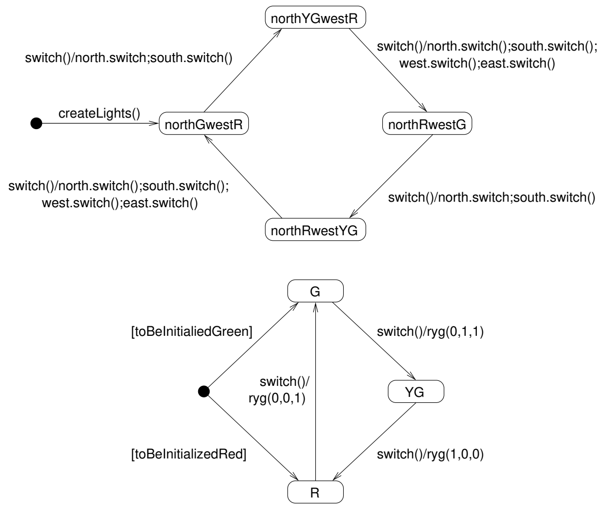

Figure 2.2. Statecharts for Crossing System.

```
  g : Integer
operations
  ryg(pr:Integer,py:Integer,pg:Integer)
  switch()
  showsR(): Boolean = r=1 and y=0 and g=0
  showsG(): Boolean = r=0 and y=0 and g=1
  showsYG():Boolean = r=0 and y=1 and g=1
end
```

Analogously to a class, an association including the association name, the participating classes, their role names and their multiplicities are described in USE textually as follows. We only show the definition of the association `North` because the associations `South`, `West`, and `East` are described analogously.

```
composition North between
  Controller[0..1] role controllerForNorth
  Light[1] role north
end
```

A central description element within USE are constraints. Constraints come in two different forms: Invariants as well as pre- and postconditions. Invariants characterize conditions which must hold during the complete object lifecycle. Pre- and postconditions characterize an operation by describing the state before and the state after the execution of the operation. Let us have a look at the `Light` invariants first. The invariant `ownedByOneController` counts the controller objects that a `Light` object is connected to and restricts this count to one. Thus, a `Light` object lives either as the north, south, west, or east `Light` object within a `Controller` object. The invariant `R_G_YG` restricts the state of a `Light` object to the allowed bulb combinations. Thus, for example, when a `Light` object has the red and green bulb both switched on, that is a forbidden state.

```
constraints -- Invariants for Light

context Light inv ownedByOneController: -- Light ownership
  controllerForNorth->size+controllerForSouth->size+
  controllerForWest->size+controllerForEast->size=1

context Light inv R_G_YG:
  -- Allowed states for single light
  Bag{showsR(),showsG(),showsYG()}
  ->select(b|b=true)->size=1
```

The invariants of the `Controller` class require that a `Controller` object is associated with exactly four `Light` objects. Furthermore, the north and the south light of the controller as well as the west and the east light of the controller have to show the same signals. Last but not least, it is required that the north and the west lights are not allowed to show only green or to show only red in the same state.

```
context Controller inv fourLightsExist:
  -- Lights are different
  Set{north,south,west,east}->size=4

-- Coincidence and non-coincidence
-- for North/South and West/East

context Controller inv northSouthCoincide:
  north.r=south.r and north.y=south.y and north.g=south.g

context Controller inv westEastCoincide:
  west.r=east.r and west.y=east.y and west.g=east.g

context Controller inv northWestNotBothG:
  not(north.g=1 and west.g=1)

context Controller inv northWestNotBothR:
```

```
  not(north.r=1 and west.r=1)
```

Apart from the above invariants, OCL expressions are used to specify operations with pre- and postconditions. Roughly speaking, the pre- and postconditions of the operation `switch` in the class `Light` and the pre- and postconditions of the operation `switch` in the class `Controller` encode the respective statecharts shown in Fig. 2.2. The details are not needed in the following, but are explained in [GR02]. We explain the basic ideas of OCL pre- and postconditions using the operation `ryg` in class `Light`. The precondition `paramsOk` checks before operation execution that the parameters to be assigned to the attributes have values which can lead in principle to a valid system state. The postcondition `assignedRYG` assures that the parameters are assigned to the respective attributes. Be aware of the fact, that the equality symbol in the postcondition is a logical equality, not an assignment. Thus, we could have written the postcondition equivalently with left and right hand sides exchanged as `pr=r and py=y and pg=g`.

```
                                                      -- ryg --
context Light::ryg(pr:Integer,py:Integer,pg:Integer)
pre paramsOk:
  0<=pr and pr<=1 and 0<=py and py<=1 and 0<=pg and pg<=1
post assignedRYG:
  r=pr and y=py and g=pg

context Light::switch() -- switch() in Light --
post R_2_G:
  (r@pre=1 and y@pre=0 and g@pre=0) implies showsG()
post G_2_YG:
  (r@pre=0 and y@pre=0 and g@pre=1) implies showsYG()
post YG_2_R:
  (r@pre=0 and y@pre=1 and g@pre=1) implies showsR()

context Controller::switch() -- switch() in Controller --
post northRwestG_2_northRwestYG:
  -- north.showsR() and west.showsG()
  ( north.r@pre=1 and north.y@pre=0 and north.g@pre=0 and
    west.r@pre=0 and west.y@pre=0 and west.g@pre=1 )
  implies ( north.showsR() and west.showsYG() )

post northRwestYG_2_northGwestR:
  -- north.showsR() and west.showsYG()
  ( north.r@pre=1 and north.y@pre=0 and north.g@pre=0 and
    west.r@pre=0 and west.y@pre=1 and west.g@pre=1 )
```

```
  implies ( north.showsG() and west.showsR() )

post northGwestR_2_northYGwestR:
  -- north.showsG() and west.showsR()
  ( north.r@pre=0 and north.y@pre=0 and north.g@pre=1 and
    west.r@pre=1 and west.y@pre=0 and west.g@pre=0 )
  implies ( north.showsYG() and west.showsR() )

post northYGwestR_2_northRwestG:
  -- north.showsYG() and west.showsR()
  ( north.r@pre=0 and north.y@pre=1 and north.g@pre=1 and
    west.r@pre=1 and west.y@pre=0 and west.g@pre=0 )
  implies ( north.showsR() and west.showsG() )

context Controller::createLights() -- createLights() --
pre lightsNotExisting:
  north->size+south->size+west->size+east->size=0
post lightsCreatedLinkedInitialized:
  north.oclIsNew and south.oclIsNew and
  west.oclIsNew and east.oclIsNew and
  north.showsG() and south.showsG() and
  west.showsR() and east.showsR()
```

2.3 USE snapshot generator and ASSL

The basic idea of USE is to validate a UML description by checking whether certain test cases meet the expected behavior. Objects can be created and destroyed, their attributes can be manipulated, and links to other objects can be established and deleted. By analysing the results of the test cases, the developers receives feedback about her or his UML description. For examples, she or he gets to know whether the invariants are too strong (too few test cases are accepted) or too weak (too many test cases are accepted). Part of the USE system is a so-called snapshot generator based on the language ASSL (A Snapshot Sequence Language). ASSL specifications try to construct with so-called procedures non-trivial system states. An example is the procedure `switch` in the following text.

```
procedure switch(theControllers:Sequence(Controller))
begin
for aController:Controller in [theControllers]
  begin
  [aController.north].r:=Try([Sequence{0,1}]);
  [aController.north].y:=Try([Sequence{0,1}]);
```

```
  [aController.north].g:=Try([Sequence{0,1}]);
  [aController.south].r:=Try([Sequence{0,1}]);
  [aController.south].y:=Try([Sequence{0,1}]);
  [aController.south].g:=Try([Sequence{0,1}]);
  [aController.west].r:=Try([Sequence{0,1}]);
  [aController.west].y:=Try([Sequence{0,1}]);
  [aController.west].g:=Try([Sequence{0,1}]);
  [aController.east].r:=Try([Sequence{0,1}]);
  [aController.east].y:=Try([Sequence{0,1}]);
  [aController.east].g:=Try([Sequence{0,1}]);
  end;
end;
```

The procedure `switch` is given a sequence of controllers. In the body of the procedure every attribute of each controller is given a value from the specified domain `{0,1}`. In the case of the procedure `switch`, after all attributes have been assigned a value, it is checked whether all invariants are valid. If this is the case, the attribute assignments are reported and can be executed. If no valid system state has been found for the attribute assignment in question, the ASSL generator backtracks and tries other assignments for the attributes. Thus, it is tried to construct a valid system state, i.e. a state where all invariants are satisfied. A successful construction yields a sequence of USE manipulation commands which applied to a current state result in a new valid state. The further employement of the generator will be explained below.

2.4 Running USE

This subsection explains how the USE system is employed to validate UML descriptions. We do so by showing commands given to USE and the responses of USE.

The first step is to read a UML description and then to initialize the crossing system. After having started USE, two windows are available for communication with USE: One window with a graphical user interface (GUI) and one window with a command line interface. Both interfaces have advantages in certain situations. Loading a model can be done through the GUI or as demonstrated below with the open command. The file `trali.use` contains the above explained textual description of the class diagram, the invariants, and the pre- and postconditions. Afterwards one `Controller` object and four Light objects are created (`!create`), the object are linked (`!insert`) and the attributes are manipulated (`!set`).

```
use> open trali.use

use> !create c:Controller
```

```
use> !create n:Light
use> !create s:Light
use> !create w:Light
use> !create e:Light

use> !insert (c,n) into North
use> !insert (c,s) into South
use> !insert (c,w) into West
use> !insert (c,e) into East

use> !set n.r=0
use> !set n.y=0
use> !set n.g=1

use> !set s.r=0
use> !set s.y=0
use> !set s.g=1

use> !set w.r=1
use> !set w.y=0
use> !set w.g=0

use> !set e.r=1
use> !set e.y=0
use> !set e.g=0
```

Figure 2.3 shows how a screenshot of the GUI after having executed the above USE commands. In the left part we see a project browser allowing to inspect the classes, the associations, the invariants, and the pre- and postconditions. The `Controller` invariant `northWestNotBothG` is selected and shown in detail below the project browser. In the right part we see four windows opened through buttons in the top part of the GUI window. We see an object diagram window, a class invariants window, a class extent window, and an OCL expression evaluation window. The object diagram window shows that in the current system state the traffic can pass in west-east direction, but the traffic has to stop in north-south direction. The class invariants window tell us that all invariants are evaluated to true in the current system state. The class extent window presents all `Light` objects together with their attribute values in table form. The OCL expression evaluation window allows to formulate an ad-hoc query against the system state and displays the result.
As shown below, checking of invariants is also supported on the command line interface via the command `check`. First, the model-inherent invariants,

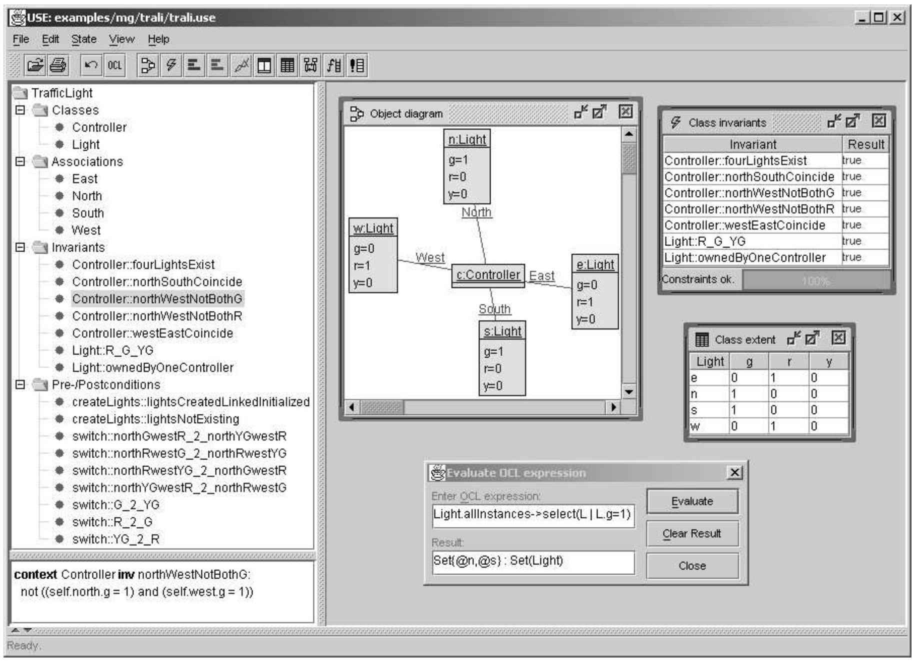

Figure 2.3. Screenshot of the USE system.

namely the multiplicity specifications from the class diagram, are checked and afterwards the explicitly given invariants. The option `-d` stands for showing a detailed result in the case that an invariant fails. We will see an example for this further below.

```
use> check -d
checking structure...
checking invariants...
checking invariant (1) ‘Controller::fourLightsExist’: OK.
checking invariant (2) ‘Controller::northSouthCoincide’:
  OK.
checking invariant (3) ‘Controller::northWestNotBothG’: OK.
checking invariant (4) ‘Controller::northWestNotBothR’: OK.
checking invariant (5) ‘Controller::westEastCoincide’: OK.
checking invariant (6) ‘Light::R_G_YG’: OK.
checking invariant (7) ‘Light::ownedByOneController’: OK.
checked 7 invariants in 0.070s, 0 failures.
```

In order to work with the snapshot generator we will need additional invariants which can be loaded dynamically and whose validity can be switched on or off. In this case, the invariants characterize the states of a controller. Note that the invariant name corresponds to the name of a respective state. Each of these invariants is placed in a separate file with a corresponding name.

```
context Controller inv northGwestR:
  self.north.showsG() and self.west.showsR()

context Controller inv northYGwestR:
  self.north.showsYG() and self.west.showsR()

context Controller inv northRwestG:
  self.north.showsR() and self.west.showsG()

context Controller inv northRwestYG:
  self.north.showsR() and self.west.showsYG()
```

These additional invariants are loaded from the respective files. Afterwards each of the invariants is switched off by setting the deactivate flag for that invariant.

```
use> gen load northYGwestR.invs
     Added invariants:
     Controller::northYGwestR
use> gen flags Controller::northYGwestR +d

use> gen load northRwestG.invs
     Added invariants:
     Controller::northRwestG
use> gen flags Controller::northRwestG +d

use> gen load northRwestYG.invs
     Added invariants:
     Controller::northRwestYG
use> gen flags Controller::northRwestYG +d

use> gen load northGwestR.invs
     Added invariants:
     Controller::northGwestR
use> gen flags Controller::northGwestR +d
```

Next we turn on exactly one of the newly added invariants, namely the one aiming for the state where the north light only shows yellow-green and the west light only shows red. Remember that we are currently in state `northGwestR`. Thus the aim is to make the first transition on the state cycle in the `Controller` statechart. The generator is started with the `gen start` command by specifying the ASSL file and the procedure in that ASSL file to be executed (there can be more than one procedure in a single ASSL file). After entering the `gen result` command, the generator reports that it has checked 1765 snapshots and was successfull in finding a valid system state. It shows the sequence of USE commands producing that state. We execute that command sequence with the `gen accept` command, turn off the temporarily switched-on invariant and demonstrate with the `check` command that all invariants are indeed satisfied.

```
use> gen flags Controller::northYGwestR -d

use> gen start trali.assl switch(Sequence{c})

use> gen result
     Random number generator was initialized with 8372.
     Checked 1765 snapshots.
     Result: Valid state found.
     Commands to produce the valid state:
     !set n.r = 0
     !set n.y = 1
     !set n.g = 1
     !set s.r = 0
     !set s.y = 1
     !set s.g = 1
     !set w.r = 1
     !set w.y = 0
     !set w.g = 0
     !set e.r = 1
     !set e.y = 0
     !set e.g = 0

use> gen result accept
     Generated result (system state) accepted.

use> gen flags Controller::northYGwestR +d

use> check -d
checking structure...
checking invariants...
checking invariant (1) ‘Controller::fourLightsExist’: OK.
checking invariant (2) ‘Controller::northSouthCoincide’:
```

```
  OK.
checking invariant (3) ‘Controller::northWestNotBothG’: OK.
checking invariant (4) ‘Controller::northWestNotBothR’: OK.
checking invariant (5) ‘Controller::westEastCoincide’: OK.
checking invariant (6) ‘Light::R_G_YG’: OK.
checking invariant (7) ‘Light::ownedByOneController’: OK.
checked 7 invariants in 0.010s, 0 failures.
```

In order to show how to work with USE in the case of invariant failure, let us look at the following command sequence. The west and east light currently show only red. We now provoke an invariant failure by switching off the red bulb of only the east light. We show the essentials of the current system state by giving a query to the USE system and afterwards check the invariants with the detail option. We learn that two invariants fail, namely the invariants `Controller::westEastCoincide` and `Light::R_G_YG`. Due to the detail option we see that controller `c` and light `e` are responsible for that invariant failure. We then reset the red bulb to the original status and again show the essentials of the current system state by evaluating a query.

```
use> !set e.r=0

use> ?Sequence{n.r,n.y,n.g, s.r,s.y,s.g, w.r,w.y,w.g,
               e.r,e.y,e.g}
-> Sequence{0,1,1, 0,1,1, 1,0,0, 0,0,0} : Sequence(Integer)

use> check -d
checking structure...
checking invariants...
checking invariant (1) ‘Controller::fourLightsExist’: OK.
checking invariant (2) ‘Controller::northSouthCoincide’:
  OK.
checking invariant (3) ‘Controller::northWestNotBothG’: OK.
checking invariant (4) ‘Controller::northWestNotBothR’: OK.
checking invariant (5) ‘Controller::westEastCoincide’:
  FAILED. -> false : Boolean
Instances of Controller violating the invariant:
  -> Set{@c} : Set(Controller)
checking invariant (6) ‘Light::R_G_YG’: FAILED.
  -> false : Boolean
Instances of Light violating the invariant:
  -> Set{@e} : Set(Light)
checking invariant (7) ‘Light::ownedByOneController’: OK.
checked 7 invariants in 0.010s, 2 failures.
```

```
use> !set e.r=1

use> ?Sequence{n.r,n.y,n.g, s.r,s.y,s.g, w.r,w.y,w.g,
              e.r,e.y,e.g}
-> Sequence{0,1,1, 0,1,1, 1,0,0, 1,0,0} : Sequence(Integer)
```

Last but not least, we show that even the answer of the USE system that no valid system state was found may be meaningful. In the following command sequence we activate two conflicting invariants, namely the invariants `Controller::northYGwestR` and `Controller::northRwestG`. Essentially, we aim at a system state where the controller is in both states from the statechart. In this case, the generator reports that it cannot find a valid system state. Indeed, it also reports that it has checked all $4096 = 2^{12}$ possible system states. The 2^{12} states originate from the four lights with each light having three attributes and each attribute having two possible values.

```
use> gen flags Controller::northYGwestR -d

use> gen flags Controller::northRwestG -d

use> gen start trali.assl switch(Sequence{c})

use> gen result
     Random number generator was initialized with 9833.
     Checked 4096 snapshots.
     Result: No valid state found.
```

3. Inputs taken from the BART case study

In this section we describe the elements of the case study we chose to specify and those we chose to drop or modify. Some subsections are named like sections of Chap. 1, making it easy to find the corresponding parts in that chapter.

3.1 Objective and general aspects

Our objective is the specification of a simplified version of the Bay Area Rapid Transit (BART) system and of an algorithm that can control the speed and acceleration of trains in this system (Par. 1–3).

A track of the BART system is partitioned into track segments. Segments may be bounded by gates (Par. 9). Station computers, which are part of the Advanced Automatic Train Control (AATC) system, control the trains in their immediate area by giving speed and acceleration commands to the trains. In each control zone, up to 20 trains can be handled (Par. 11). We abstract from communication links between computers and trains, the on-board train control

system, managing entry and exit of trains into the system, and hand-offs between stations (Par. 12). We assume that the interlocking system does not close a gate when it is to late for an approaching train to stop, not even as a signal to following trains. This is a modification to Par. 13.
The control algorithm is subject to various constraints. These include the three constraints stated in Par. 14:

- A train should not enter a closed gate.
- A train should never get so close to a train in front that if the train in front stopped suddenly the (following) train would hit it.
- A train should stay below the maximum speed that track segment can handle.

We do some major simplifications: When controlling the trains, we assume that the station computers have direct access to speed, acceleration, and position information of the trains. We do not care about uncertainty envelopes and mean and standard deviation (Par. 16). Also, we do not attach time stamps to commands (Par. 17). A commanded speed and acceleration is valid until a new commanded speed and acceleration is given, they do not expire (Par. 18). Station computers are not divided into vital and non-vital station computers (Par. 19–21).

3.2 Inputs and outputs to the control algorithm

The control algorithm has access to all relevant attributes of all trains, including position, speed, acceleration, length, and currently commanded speed and acceleration. Information of the track is also available, i.e., location and grade of segments, maximum allowable speed, and location and state (open, closed) of gates.
The output of the algorithm is a commanded speed (between 0 and 80 mph) and a commanded acceleration (-2 to -0.45 mphps in braking, 0 to 3 mphps in propulsion) for a given train.

3.3 Physical performance of the train in response to commands

When given a speed and acceleration command, the train will accelerate to 2 mph under the commanded speed at the indicated acceleration. We neglect the fact that maximum acceleration depends on speed. Once a speed 2 mph below the commanded speed is achieved, the train will attempt to maintain that speed (Par. 29).
We assume that the *open-loop* braking mode is never used. Therefore, the maximum braking rate is 2 mphps (i.e., an acceleration of -2 mphps). The

worst case profile assumes maximum braking rates below 2 mphps anyway (Par. 31–32). The minimum braking rate is 0.45 mphps (Par. 34).
Opposing to the original case study description, it takes no extra time to move from propulsion into braking and vice versa. Moreover, changing acceleration (positive as well as negative) from a_1 to a_2 always takes 0.5 seconds, no matter of the values of a_1 and a_2 (Par. 35–36). Grades have an influence on actual accelerations (Par. 37).

3.4 Worst case stopping profile

Speed and acceleration of a train has to be selected so that (1) the train does not hit a train in front of it or (2) enter a closed gate, not even in the case of very poor stopping conditions (Par. 38). We adopt the complete worst case stopping profile calculations exposed in Table 1.1 and 1.2 and described in Par. 39–49. During Validation (Sect. 4), we decided to modify the calculation in one major point: We use `v` instead of `vcm` (Par. 42), because otherwise the stopping could be too optimistic if `vcm` is lower than `v`. We adopt this worst case model, even though in our simplified system the worst case would not be such bad (mode change takes no extra time etc.).
Speed and acceleration must be consistent with worst-case safety bounds. We do not have any secondary objectives (Par. 50). Some secondary objectives of the BART case study (e.g. regarding the passengers comfort) possibly will be accomplished but we neither put any effort in it nor do we test such properties.

4. Applying the approach to the case study

In this section, we will see UML and OCL in action when working on the BART case study. We develop a specification consisting of several diffrent parts. In Sect. 4.1, the existing train system is modelled by a UML class diagram with additional OCL constraints. All side effect-free operations of the classes are specified by OCL expressions that calculate the results. The safety requirements that must be satisfied are also specified in this section. In Sect. 4.2, the physical behavior of trains is specified by a procedure that gets the watch half a second ahead and moves the trains accordingly. In Sect. 4.3 we then present a simple control algorithm, that is expected to meet the requirements in the environment set in Sect. 4.1 with trains behaving as specified in Sect. 4.2. Finally, we validate all parts of the specification by loading running tests in different scenarios in Sect. 4.4.

4.1 Specifying the problem with UML and OCL

Specifying the problem starts with the construction of the object model with class operations being described by OCL expressions as result values. OCL constraints specify the requirements each instantiation of the object model (i.e., each system state) must fulfill.

4.1.1 The basic object model. The UML class diagram in 2.4 depicts the object model all further specifications refer to. A segment belongs to a track. The attributes of class *Segment* capture begin, end and length of the segment as well as civil speed, grade and exposure. Segments are ordered by an association *SegmentOrder*. A station platform is a special segment additionally having a name. Segments can be bounded by gates at the segment end. The attribute `open` of class *Gate* indicates whether the gate is open or closed. A train is located on a track. The attributes of class *Train* hold the location of its nose, its speed `v`, acceleration `a`, commanded speed `vcm` and commanded acceleration `acm`. The length of the train is also recorded. A train is originating from a station platform and has another station platform as destination. A station computer controls trains on a part of the track. This part is determined by a segment at which it begins (role `sb`) and one at which it ends (role `se`). The operations of the classes with no side-effects are specified in the following sections.

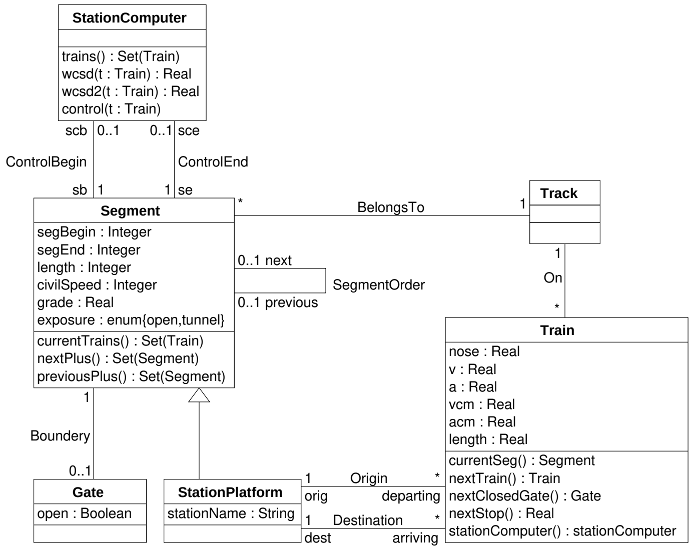

Figure 2.4. UML object model of train system.

4.1.2 Operations of class Train. The operation `currentSeg()` computes the segment on which the train is currently located. The segment belongs to the track the train is located on and the nose of the train is somewhere between begin and end of the segment. The construct `->asSequence()->first()`, occurring in the operation definition below, is often used to get the element from a set with exactly this one element in it.

```
currentSeg() : Segment =
  self.track.segment
    ->select(segBegin <= self.nose and segEnd > self.nose)
    ->asSequence()->first()
```

The operation `nextTrain()` finds the train which is in front of the given train. It is the train it has to be avoided to crash into.

```
nextTrain() : Train =
  let candidates = self.currentSeg().nextPlus()
    ->collect(currentTrains())
    ->flatten()
    ->select(t | t.nose > self.nose)
    ->asSequence() in
  if candidates->isEmpty()
  then candidates->first()
  else candidates->iterate(t:Train;
    result:Train = candidates->first()|
    if t.nose < result.nose then t else result endif)
  endif
```

The operation `nextClosedGate()` gives the next coming gate that is closed. The operation `first()` applied to an empty sequence results in `Undefined`, so this is the result of `nextClosedGate()` if there is no closed gate coming.

```
nextClosedGate() : Gate =
  let candidates = self.currentSeg().nextPlus()
    ->collect(gate)
    ->select(not open)
    ->asSequence() in
  if candidates->isEmpty()
  then candidates->first()
  else candidates->iterate(g:Gate;
    result:Gate = candidates->first()|
    if g.segment.segEnd < result.segment.segEnd
    then g else result endif)
  endif
```

The next stop of a train is either at the next closed gate or at the back of the next train or at the end of its destination segment, depending on what is nearer. The operation `nextStop()` calculates the absolute position of this place.

```
  nextStop() : Real =
    let candidates = Sequence{self.dest.segEnd,
      self.nextClosedGate().segment.segEnd,
      self.nextTrain().nose - self.nextTrain().length}
      ->reject(isUndefined) in
    candidates->iterate(s:Real;
      result:Real = candidates->first() |
      if s < result then s else result endif)
```

The station computer that is responsible for the train changes depending on the position. The operation `stationComputer()` results in the currently responsible computer.

```
stationComputer() : StationComputer =
  StationComputer.allInstances
    ->select(sc | self.currentSeg().nextPlus()
        ->includes(sc.se))
    ->select(sc |
        sc.sb.segBegin <= self.currentSeg().segBegin and
        sc.se.segEnd >= self.currentSeg().segEnd)
    ->asSequence()->first()
```

4.1.3 Operations of class Segment.

The operation `nextPlus()` applied on a segment results in a sequence that includes the segment itself and all next coming segments. It uses the auxiliary operation `nextPlusAux()`.

```
nextPlus() : Set(Segment) =
  nextPlusAux(Set{self})->reject(isUndefined)

nextPlusAux(s:Set(Segment)) : Set(Segment) =
  if s->collect(seg|seg.next)
    ->exists(seg | s->excludes(seg))
  then nextPlusAux(s->union(s->collect(seg | seg.next))
    ->asSet)
  else s
  endif
```

The operation `previousPlus()` analogously gives the previous segments.

```
previousPlus() : Set(Segment) =
  prevPlusAux(Set{self})->reject(isUndefined)

prevPlusAux(s:Set(Segment)) : Set(Segment) =
  if s->collect(seg|seg.previous)
    ->exists(seg|s->excludes(seg))
```

```
  then prevPlusAux(s->union(s->collect(seg|seg.previous))
    ->asSet)
  else s
  endif
```

The operation `currentTrains()` is the inverse of the operation `currentSeg()` of class *Train*. It gives all trains that are currently located on the segment.

```
currentTrains() : Set(Train) =
  Train.allInstances->select(currentSeg() = self)
```

4.1.4 Operations of class StationComputer.

The operation `trains()` is the inverse of the operation `stationComputer()` of class *Train*. It gives all the trains the station computer is currently responsible for.

```
trains() : Set(Train) =
  Train.allInstances->select(stationComputer() = self)
```

The operation `wcsd()` calculates the worst case stopping distance of a train `t`. The operation adheres to the formulas given in Table 1.1 on page 12 and Table 1.2 on page 13 with one exception: we use `v` instead of `vcm` in line 7 and 17. Otherwise the calculation would be to optimistic if `vcm` is lower then `v`. In line 46–49 the result is calculated by addition of `d1`, `d2` and `d7` if the train is braking, or by addition of `d1` to `d7` otherwise (see Par. 47).

```
1  wcsd(t:Train) : Real =
2    let pu:Real = 3.0 in      -- position uncertainty
3    let puf:Integer = 6 in    -- position uncertainty factor
4    let d1:Real = (puf*pu) in -- no noise
5
6    let ad:Real = 2 in        -- AATC delay
7    let d2:Real = t.v*ad in
8
9    let tjp:Real = 1.5 in     -- jerk time in propulsion
10   let ap:Real = 3 in        -- acceleration in propulsion
11   let jp:Real = ap/tjp in   -- jerk limit in propulsion
12   -- acceleration due to grade:
13   let a:Real = -21.9*t.currentSeg().grade/100 in
14     then -1.5
15   let d3:Real = t.v*tjp + 1/2*ap*tjp*tjp +
16     1/6*jp*tjp*tjp*tjp + 1/2*a*tjp*tjp in
17
18   let v3:Real = t.v + ap*tjp + 1/2*jp*tjp*tjp + a*tjp in
19   let mc:Real = 1 in -- mode change
20   let d4:Real = v3*mc + 1/2*a*mc*mc in
```

```
21
22   let v4:Real = v3+a*mc in
23   let ncar:Integer = 10 in  -- number of car in consist
24   let nfail:Integer = 2 in  -- number of failed cars
25   let nfsmc:Integer = 2 in  -- number of cars in FSMC
26   let qfsmc:Real = (ncar-nfail-nfsmc)/ncar in
27   let brk:Real = if t.currentSeg().exposure=#open
28     then -1.5
29     else -2.0
30     endif in                -- design brake rate
31   let jb:Real = -1.5 in     -- jerk limit in braking
32   let tjb:Real = brk/jb in
33   let d5:Real = v4*tjb + 1/6*jb*qfsmc*tjb*tjb*tjb
34     + 1/2*a*tjb*tjb in
35
36   let v5:Real = v4 + 1/2*jb*qfsmc*tjb*tjb + a*tjb in
37   let fsmc:Real = 8.5 in    -- fail safe mode change time
38   let t6:Real = fsmc-tjp-mc-tjb in
39   let bfs:Real = brk*qfsmc in
40   let d6:Real = v5*t6 + 1/2*bfs*t6*t6 + 1/2*a*t6*t6 in
41
42   let v6:Real = v5 + bfs*t6 + a*t6 in
43   let q:Real = (ncar-nfail)/ncar in
44   let vf=0 in               -- final speed (mph)
45   let d7:Real = (vf*vf - v6*v6)/(2*(brk*q + a)) in
46
47   if t.a < 0
48   then d1+d2+d7
49   else d1+d2+d3+d4+d5+d6+d7
50   endif
```

We do not show the OCL expression for the operation `wcsd2()` because there are only a few differences to `wcsd()`. The distance computed by `wcsd2()` is more pessimistic than the one calculated by `wcsd()`: (1) The acceleration due to grade always presumes a grade of -4%, i.e., the subexpression `t.currentSeg().grade` is replaced by `-4` in line 13. (2) Brake rate is always only 1.5, therefore, the lines 27–30 are replaced by `let brk:Real = -1.5`. (3) Acceleration is presumed to be positive so that all distances `d1` to `d7` are added up. Hence, the lines 47–50 are replaced by `d1+d2+d3+d4+d5+d6+d7`. This more pessimistic stopping distance is less fluctuating so it is more suitable for our control algorithm as we shall see.

4.1.5 OCL constraints on the train system.

The invariant `fitting` states that the end of a segment is equal to the begin of the next segment if there is a next segment.

```
context Segment inv fitting:
  self.next.isDefined implies
  self.next.segBegin=self.segEnd
```

The length of a segment has to be the difference of segment end and begin:

```
context Segment inv correctLength:
  self.segEnd-self.segBegin = self.length
```

Connected segments belong to the same track:

```
context Segment inv track:
  self.next.isDefined implies self.track = self.next.track
```

The origin and the destination of a train have to be connected by a sequence of segments:

```
context Train inv line:
  self.orig.nextPlus()->includes(self.dest)
```

A train can not be commanded to travel faster than 80 mph or slower than 0 mph (Par. 27):

```
context Train inv vcm:
  self.vcm >= 0 and self.vcm <= 80
```

The commanded acceleration is either between 0 and 3 or between -2 and -0.45 mphps(Par. 27):

```
context Train inv acm:
  (self.acm >= 0 and self.acm <= 3) or
  (self.acm >= -2 and self.acm <= -0.45)
```

A station computer is able to calculate the worst case stopping distance only of trains that are in the region the computer is responsible for:

```
context StationComputer::wcsd(t:Train) : Real
  pre: self.trains()->includes(t)
```

The same holds for the controlling of trains:

```
context StationComputer::control(t:Train)
  pre: self.trains()->includes(t)
```

A station computer is responsible for at most 20 trains (Par. 11):

```
context StationComputer::trains() : Set(Train)
  post: result->size() <= 20
```

The segments that bound the region a station computer is responsible for are connected.

```
context StationComputer inv bounderies:
  self.sb.nextPlus()->includes(self.se)
```

The last three invariants form the central part of the specification. They formalize the main constraints of the train system. The `control()` operation has to be designed in a manner that assures that these invariants do not fail. The constraint `civilSpeedSafety` demands that a "train should stay below the maximum speed that segment of track can handle" (Par. 14).

```
context StationComputer inv civilSpeedSafety:
  self.trains()->forAll(t |
    t.v <= t.currentSeg().civilSpeed)
```

„A train should not enter a closed gate" (Par. 14). The invariant with the name `closedGateSafety` says that if a next closed gate exists, the distance to it is greater than the worst case stopping distance of the train, i.e., the train can stop in time.

```
context StationComputer inv closedGateSafety:
  self.trains()->forAll(t |
    t.nextClosedGate().isDefined implies
    t.nose+self.wcsd(t) <
    t.nextClosedGate().segment.segEnd)
```

„A train should never get so close to a train in front that if the train in front stopped suddenly (e.g., derailed) the (following) train would hit it" (Par. 14). The invariant `crashSafety` says (analogously to the preceding invariant) that if a train in front exists, the distance to it is greater than the worst case stopping distance of the (following) train.

```
context StationComputer inv crashSafety:
  self.trains()->forAll(t |
    t.nextTrain().isDefined implies
    t.nose+self.wcsd(t) <
    t.nextTrain().nose-t.nextTrain().length)
```

4.2 Moving the trains

We want to validate our specification by simulating trains travelling on a track. An operation is needed that implements the physical behavior of trains as time elapses by setting the attributes of the trains as they would be after 0.5 seconds have gone by. Because such an operation obviously has side-effects, a pure OCL expression does not suffice to specify it. Therefore, `move()` is an ASSL procedure that modifies the attributes of all train according to the OCL expressions that constitute the bulk of the procedure. `move()` does not correspond with any operation of the object model.

```
procedure move()

var
  delta: Real,
  gradeacc: Real;

begin
  delta := [0.5];

  for t: Train in [Train.allInstances->asSequence()]
  begin

    gradeacc := [21.9*t.currentSeg().grade/100*-1];

    [t].nose := [
      let n = t.nose + t.v*delta + 1/2*t.a*delta*delta
        + 1/2*gradeacc*delta*delta in
      if t.v = 0 and t.vcm = 0 then t.nose
      else n endif];

    [t].v := [
      let speed = t.v + 1/2*t.a*delta
        + 1/2*gradeacc*delta in
      if (t.v = 0 and t.vcm = 0) or speed <= 0 then 0.0
      else speed endif];

    [t].a := [
      let noseAtNext = t.nose + t.v*delta
        + 1/2*t.a*delta*delta
        + 1/2*gradeacc*delta*delta in
      let grade = t.currentSeg().nextPlus()
        ->select(segBegin <= noseAtNext
```

```
33              and segEnd > noseAtNext)
34            ->asSequence()->first().grade in
35          if (t.v = 0 and t.vcm=0) then 0.0 else
36          if ((t.v > (t.vcm-2) and t.acm > 0) or
37            (t.v < (t.vcm-2) and t.acm < 0))
38          then 21.9*grade/100
39          else t.acm endif endif];
40    end;
41  end;
```

The procedure loops over all existing trains (line 10). The variable `gradeacc` holds the acceleration due to grade (line 13). In line 15–19 the position of the nose is calculated by means of the appropriate physical formula. However, if both `v` and `vcm` are zero, the position remains unchanged, i.e., the acceleration due to grade does not have any effect. The train is "fixed" as soon as it has once stopped.

In line 21–25 the new speed is computed. The train can not go backwards (Par. 66), so if the result of the computation is negative the speed is set to zero, the train stops. The speed is also set to zero if both `v` and `vcm` are zero.

The acceleration is set in line 27–39. If both `v` and `vcm` are zero, acceleration is set to zero, too, i.e., the train is "fixed" again. If the speed has achieved the commanded speed ($\pm$ 2 mph) the trains attempts to maintain this speed by compensating the acceleration due to grade. To do this, the grade of the segment the train will be located on in the next state is considered. If the commanded speed is not achieved yet, the acceleration is set to the commanded acceleration. Therefore, it takes the train 0.5 seconds to set a new acceleration, no matter how it differs from the current one.

4.3 A simple control algorithm

The ASSL procedure `control(t: Train)` implements the operation of class *StationComputer* that calculates the new commanded speed and acceleration for a train `t`. These values have to respect the constraints of the train system including `civilSpeedSafety`, `closedGateSafety` and `crashSafety`. The algorithm inspects the section of the track that begins with the nose of the train and is twice as long as the (more pessimistic but less fluctuating) worst case stopping distance `wcsd2(t)` (line 13). Only maximum speeds and obstacles within this range are considered.

```
1   procedure control(t: Train)
2
3   var
4     range: Real,
5     gradeacc: Real,
```

```
  delta: Real,
  vcmCivilSpeed: Real,
  commandedSpeed: Real,
  commandedAcc: Real;

begin
  delta := [0.5];
  range := [t.stationComputer().wcsd2(t)*2+100];
  gradeacc := [21.9*t.currentSeg().grade/100*-1];

  vcmCivilSpeed := [
    let civilSpeeds = t.currentSeg().nextPlus()
      ->select(segBegin <= t.nose+range)
      ->collect(civilSpeed)
      ->asSequence in
    civilSpeeds ->iterate(speed: Real;
      result: Real = civilSpeeds->first() |
      if speed < result then speed
      else result endif)];

  commandedSpeed := [
    if t.nextStop()-t.nose < range
    then 0.0 else vcmCivilSpeed endif];

  commandedAcc := [
    let pos = t.currentSeg().nextPlus()
      ->select(segBegin <= t.nose+range)
      ->select(civilSpeed = vcmCivilSpeed)
      ->collect(segBegin)->asSequence in
    let d1 = pos->iterate(p: Real;
      result: Real = pos->first() |
      if p < result then p
      else result endif) - t.nose in
    let acc =
      if d1 <= 0 then t.a+0.5
      else ((vcmCivilSpeed-2)*(vcmCivilSpeed-2)
        - t.v*t.v)/(2*d1) - gradeacc
      endif in
    let acmCivilSpeed =
      if (acc < 0 and acc > -0.45) then -0.45
      else acc endif in

```

```
48      let d2 = t.nextStop() - t.nose
49        - t.stationComputer().wcsd2(t) in
50      let acc= -(t.v*t.v)/(2*d2) - gradeacc in
51      let acmNextStop =
52        if t.nextStop()-t.nose > range then t.a+0.5 else
53          if (acc < 0 and acc > -0.45 and
54            d2 > t.v*delta + 1/2*gradeacc*delta*delta)
55          then 0.0
56          else
57            if (acc < 0 and acc > -0.45 and
58              d2 <= t.v*delta + 1/2*gradeacc*delta*delta)
59            then -0.45 else acc endif
60          endif
61        endif in
62
63      let acm =
64        if acmCivilSpeed < acmNextStop then acmCivilSpeed
65        else acmNextStop endif in
66
67      if acm < -2.0 then -2.0 else
68      if acm > 3.0 then 3.0 else
69      if acm < 0 and acm > -0.45 then -0.45 else
70      if t.v <= 0.5 and commandedSpeed = 0 then -2.0
71      else acm endif endif endif endif];
72
73    [t].vcm := [commandedSpeed];
74    [t].acm := [commandedAcc];
75  end;
```

The attributes `vcm` and `acm` are set in line 73–74 to `commandedSpeed` resp. `commandedAcc`. `commandedSpeed` is calculated in line 26–28. It is set to zero if the next stop (closed gate, back of next train or end of destination station platform) is within the range. Otherwise it is set to the lowest of all civil speeds within the range (`vcmCivilSpeed`, which is calculated before).
The determination of `commandedAcc` in line 30–71 is a little bit more complex. First (line 31–46), the acceleration `acmCivilSpeed` is calculated. The calculation of this acceleration only considers civil speed restrictions, other constraints can be violated. `acmCivilSpeed` is set to the acceleration that is needed to achieve a speed 2 mph below `vcmCivilSpeed` at the beginning of the next segment with `vcmCivilSpeed` as civil speed, so a violation of the invariant `civilSpeedSafety` should be avoided. If the train happens to be already on that segment, `acmCivilSpeed` is set to the current acceleration incremented

by 0.5 mph. If the resulting acceleration is between 0 and -0.45, which is not an allowed commanded acceleration (Par. 27), it is rounded off to -0.45.
In Line 48–61, `acmNextStop` is computed. If the next stop is out of the range, `acmNextStop` is set to the current acceleration incremented by 0.5 mph. Otherwise, it is set to let the train stop `wcsd2(t)` feet before the obstacle. In this way, a violation of the invariants `closedGateSafety` and `crashSafety` should be avoided. Because `wcsd2(t)` is shrinking when the train gets slower, the train finally stops in a (hopefully) reasonable distance to the obstacle. If the resulting acceleration is between 0 and -0.45, it is rounded up to zero if this does not causes a constraint violation and rounded off to -0.45 otherwise. If it would always be rounded off, the train would stop too soon, which is not desired. Note that the algorithm presumes that the new acceleration is set immediately in spite of the 0.5 seconds it takes. However, this is not critical because of the more pessimistic stopping distance it uses.
`acm` (line 63–65) is set to the minimum of `acmCivilSpeed` and `acmNextStop`. The acceleration is truncated to be in the allowed span (line 67–71). Finally, if the commanded speed is zero and the train is already slower then 0.5 mph, the train is commanded to stop with an acceleration of -2 mphps. This has be done to avoid a problem that reminds of Zeno's paradox of Achilles and the turtle: Roughly speaking, the train never reaches its stopping point because this point is slided forward a little bit each step.

4.4 Validation with USE

In this section we show how our OCL specification can be validated with the USE tool.

4.4.1 Preparation. In order to validate our control algorithm, we create a USE specification (in a file `bart.use`) that contains the object model (see Fig. 2.4) including the specifications of the side effect-free operations and the OCL constraints. The two ASSL procedures `move()` and `control(t: Train)` are put into a file `procs.assl`. At next we create a command file named `oneTrainThrough.cmd` that contains USE commands to create objects and links and to set attribute values according to the sample track (Par. 79). Grades of type *transition* are set to a proper concrete grade. In `oneTrainThrough.cmd`, which respresents our first scenario, one train with the name *Choochoo* and length 710 is set at the end of Daly City Station Platform with all attributes (other than `length`) set to zero. The resulting object diagram is shown in Fig. 2.5.
The first thing we encounter after opening the specification (`open bart.use`) and executing the command file (`read oneTrainThrough.cmd`) is that the constraint `Segment::fitting` is violated several times, i.e., the sample track

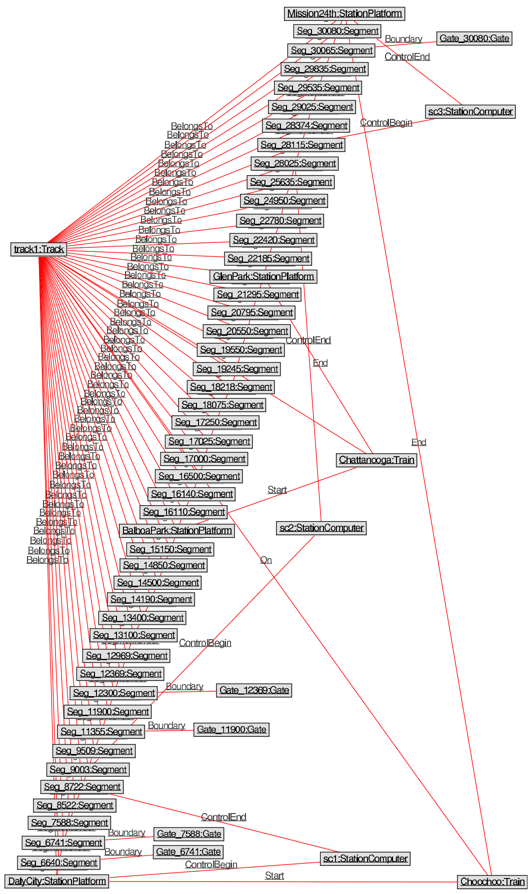

Figure 2.5. UML object model.

contains some inconsistencies: Some segments are overlapping, others have a gap between them.

```
use> check
checking structure...
checking invariants...
checking invariant (1) ‘Segment::correctLength’: OK.
checking invariant (2) ‘Segment::fitting’: FAILED.
  -> false : Boolean
checking invariant (3) ‘Segment::track’: OK.
checking invariant (4) ‘StationComputer::bounderies’: OK.
```

```
checking invariant (5) 'StationComputer::civilSpeedSafety':
 OK.
checking invariant (6) 'StationComputer::closedGateSafety':
 OK.
checking invariant (7) 'StationComputer::crashSafety': OK.
checking invariant (8) 'Train::acm': OK.
checking invariant (9) 'Train::line': OK.
checking invariant (10) 'Train::vcm': OK.
checked 10 invariants in 15.751s, 1 failure.
```

We now get more details about the violation of `Segment::fitting`. The checking mechanism of USE cancels the examination of a constraint when the first violation is detected, so we get only the first violating instance:

```
use> check -d Segment::fitting
checking structure...
checking invariants...
checking invariant (1) 'Segment::fitting': FAILED.
  -> false : Boolean
Instances of Segment violating the invariant:
  -> Set{@Seg_12300} : Set(Segment)
checked 1 invariant in 0.028s, 1 failure.
```

A look into `oneTrainThrough.cmd` reveals a gap between position 12369 and 12969. After having repaired this, three failures of the same type occur, which we all relieve either by insertion of another segment or by modification of a segment's attribute values. The inconsistencies are repaired in the final version of the sample track layout that is now included in this book (Par. 79).
In order to minimize typing when running a test, we create a command file `step.cmd` with the following content:

```
gen start procs.assl control(Choochoo)
gen result
gen result accept

gen start procs.assl control(Chattanooga)
gen result
gen result accept

gen start procs.assl move()
gen result
gen result accept
```

The generator extension of USE is told to execute `control(Choochoo)` to control the train *Choochoo*. However, the execution does not yet effect a change

of the system state–the result is a sequence of USE commands that are shown with the command `gen result` and executed with `gen accept`. The same is done for the train *Chattanooga*, which is added in other scenarios described below. In scenarios with only one train, the controlling lines for the other are deleted. The last three lines move the trains and therefore do a 0.5 seconds step into future.
To spare even more typing, we create a file `run.cmd` that contains x times the command `read step.cmd`, so we only needed to execute `run.cmd` to run x steps. The acitivity diagram in Fig 2.6 shows the overall strategy for running a test. First, the specificaton file is opened. Then, one of the scenarios is loaded by executing the corresponding command file. We proceed with the execution of `run.cmd`, which executes `step.cmd` several times. `step.cmd` in turn calls the `control` procedure to control *Choochoo* (and *Chattanooga* if neccessary) and `move()` to elapse another 0.5 seconds.

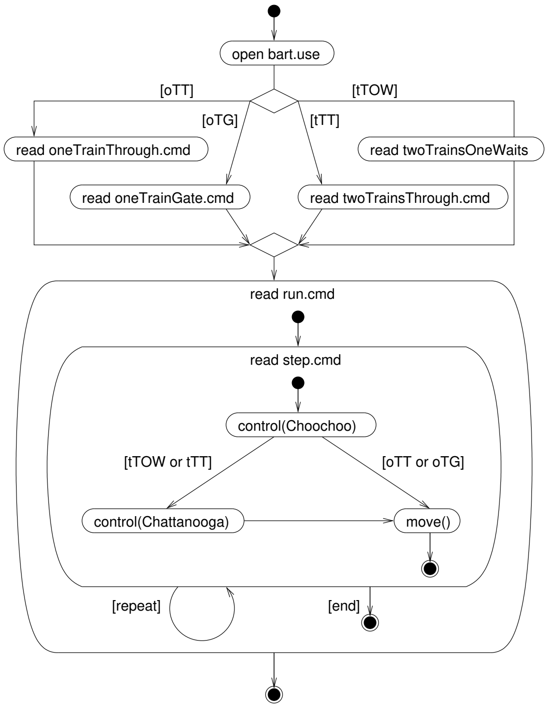

Figure 2.6. UML activity diagram showing the activities that are performed during a test run.

4.4.2 The Test Scenarios. We want to run four different test scenarios, differing from each other in the state of the gates (open or closed) or the number of existing trains and in their origin, destination and current location. These scenarios are shown in Fig. 2.7.

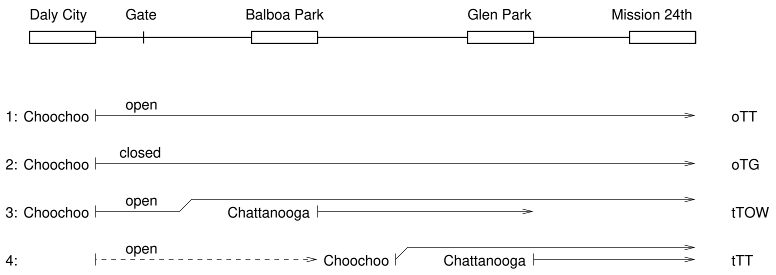

Figure 2.7. The four test scenarios. The markers mark the ends of the station platform segments and the location of one gate. 1: `oneTrainThrough.cmd` (oTT), 2: `oneTrainGate.cmd` (oTG), 3: `twoTrainsOneWaits.cmd` (tTOW), 4: `twoTrainsThrough.cmd` (tTT).

We already mentioned the first scenario that is created by executing the command file `oneTrainThrough.cmd`. Only one train *Choochoo* is created. It is put at the end of Daly City Station Platform with all attributes set to zero (except for `length` which is set to 710). The train's origin is Daly City Station Platform, its destination is 24-th Street Mission Station Platform. In this scenario, we want to validate that the constraint `civilSpeedSafety` is respected. In addition, the behavior of the train when stopping at its destination can be observed.

In the second scenario (created with `oneTrainGate.cmd`), we want to observe the train when stopping in front of a closed gate. Therefore, we set the attribute `open` of the gate at position 12369 to `false`. All other settings remain the same as in the first scenario.

Next, we want to examine the interplay of two trains, so we add another train *Chattanooga* in `twoTrainsOneWaits.cmd`. Its origin is Balboa Park Station Platform, the destination is Glen Park Station Platform. The test run starts with *Chattanooga* at the end of Balboa Park Station Platform with speed and acceleration set to zero. *Choochoo* is expected to wait behind *Chattanooga* when *Chattanooga* arrives at its destination.

In the fourth scenario, we continue with the preceding scenario by changing the destination of *Chattanooga* from Glen Park Station Platform to 24-th Street Mission Station Platform. *Choochoo* is supposed to regain speed as soon as *Chattanooga* is away and finally stop behind *Chattanooga* at 24-th Street Mission Station Platform.

The console output after a step looks like this (taken from the first scenario):

```
run.cmd> read step.cmd
step.cmd> gen start  procs.assl control(Choochoo)
step.cmd> gen result
Random number generator was initialized with 1286.
Checked 1 snapshots.
Result: Valid state found.
Commands to produce the valid state:
!set Choochoo.vcm = 36
!set Choochoo.acm = 0.3248
step.cmd> gen result accept
Generated result (system state) accepted.
step.cmd>
step.cmd> gen start procs.assl move()
step.cmd> gen result
Random number generator was initialized with 1953.
Checked 1 snapshots.
Result: Valid state found.
Commands to produce the valid state:
!set Choochoo.nose = 7323.01379999999
!set Choochoo.v = 34.6836
!set Choochoo.a = -0.1752
step.cmd> gen result accept
Generated result (system state) accepted.
```

If the execution of one of the procedures violates a constraint, the output notes that no valid state is found. In this case, the system state will not be modified. The job of the specifying engineer is to find the reason for this violation by looking into the details of the procedure execution with the `-d` option of the `gen start` command and by reproducing parts of the procedure with the USE OCL expression evaluator.

5. Results raised by this technique

The easiest way to work with USE is to first specify a UML class diagram and some OCL constraints and then on the one hand create object diagrams that represent valid system states and on the other hand create objects diagrams that represent invalid system states. Then it is checked with USE whether those system states are really valid resp. invalid in the specified system, i.e., the specification is validated.

To validate the specification in this way, it is necessary to create system states that can confidently be regarded as “has to be valid” or “must not be valid”.

Once the specification is considered as correct, more complex system states that are not easily understandable can be checked with respect to the specification. In this chapter, we specified the existing train system with a class diagram, OCL result expressions for the side effect-free class operations, and OCL constraints. The physical behavior of the trains was specified by the ASSL procedure `move()`. This constitutes our *specification*.

The ASSL procedure `control(t: Train)` also is a specification in the sense that we mainly used OCL to describe the effect of the operation but we refer to it as *implementation* that satisfies the safety requirements that are part of the specification.

Therefore, we had to validate the specification with respect to our intuition so that after that the implementation could be validated with respect to the specification. In fact we validated both at the same time by running the test scenarios. Syntax errors are of course quickly located and corrected. Several semantic errors in the specification were found immediately after loading the initial state of a scenario, e.g., inconsistencies in the sample track indicated by constraint violation. Finding errors while moving the trains can be more complicated because of the interdependence of specification and implementation. There are four different cases:

1 A constraint is violated and we judge this violation as unjustified. E.g., the constraint `civilSpeedSafety` is violated although the speed of all trains is below the allowed maximum speed. In this case, the specification is incorrect and the implementation is correct (in this context).

2 A constraint is violated and we judge this violation as justified. E.g., the constraint `civilSpeedSafety` is violated and indeed one train exceeds the civil speed. In this case, the implementation is incorrect and the specification is correct (in this context).

3 No constraint is violated but we observe that one of the system states is not allowed. E.g., the constraint `civilSpeedSafety` is not violated although one train exceeds the civil speed. In this case, both implementation and specification are incorrect.

4 No constraint is violated and we do not detect any unallowable system states. This is the aim of our validation process but there are of course some uncertainties: (1) Maybe other test scenarios lead to one of the cases above. This is a general problem of validation by test scenarios and has to be antagonized by proper selection of test scenarios. (2) Maybe the implementation is correct with respect to our demands but not with respect to the specification that allows some undesired system states. We could not detect the bug in the specification. This is critical if the specification is supposed to be used later for another implementation.

The validation exposed numerous errors of our specification and implementation. The specification/implementation shown in the previous section is the final version that resulted from several steps of refinement. The exposed errors can be categorized as follows:

1 Syntax errors. These errors of course can be found and corrected quickly.

2 Small mistakes like the inconsiderate use of < instead of <= or a forgotten algebraic sign.

3 Faults or inconsistencies in the case study description that have been adopted in the formal specification. Examples of this kind are the questionable use of `vcm` instead of `v` in the calculation of the the worst case stopping distance and the formula for acceleration due to grade, which has to be multiplied with -1 to give the correct value.

4 Constants that are not adjusted properly. An example is the constant `0.5` in line 68 of `control(t: Train)`, which previously was a `0.2`.

5 Errors in reasoning. For example, in `control(t: Train)`, setting the acceleration so that the train stops a constant distance feet behind the train ahead does not guarantee that `crashSafety` is always respected because the worst case stopping distance could temporarily be longer than the distance to the next train.

After refinement of the specification (which induced the present specification), the test runs finally yield the following results:

1 In the first scenario, *Choochoo* finally stops at position 32741, which is 240 feet before the the end of 24-th Street Mission Station Platform.

2 In the second scenario, it stops at position 12123, which is 246 feet before the closed gate.

3 In the third scenario, *Chattanooga* stops at position 21944, which is 271 feet before the end of Glen Park Station Platform. *Choochoo* stops at position 20986, which is 260 feet before the back of *Chattanooga*, which is 710 feet long.

4 When changing the destination of *Chattanooga* from Glen Park Station Platform to 24-th Street Mission Station Platform (fourth scenario), *Chattanooga* regains speed and finally stops at position 32741, which is 240 feet before the end of 24-th Street Mission Station Platform. *Choochoo* waits until the distance to *Chattanooga* has increased and then regains speed as well. *Choochoo* finally stops at 31778, 253 feet before the back of *Chattanooga*.

Two things attract attention: First, the trains stop in a distance to the obstacle that might be longer than necessary. Second, the speed of the trains is seldom higher than 36 mph, the lowest of the civil speeds within the sample track. In scenario one, the maximum speed of *Choochoo* is 39.37 mph even though maximum allowed speed is temporarily 80 mph. A reason for both is the used worst case stopping distance, which might be too pessimistic. The low maximum speed also results from the specific sample track. Trains would get faster on tracks which do not have low civil speeds on long distances. However, we rate our specification as sufficient as the putative deficiencies do not concern safety.

6. Conclusion

We have described the functionality of the UML Specification Environment USE. USE allows to validate formal specifications by providing test cases and checking whether the system's responses meet the intuition. USE covers validation and verification aspects.

We have expressed the central safety requirements from the BART case study through OCL invariants. If these invariants are satisfied, it is guaranteed that no accident can occur. In our specification, the invariants are assumed to be checked every 0.5 second. If a situation involving a closed gate or a stopped train in front of a running train is encountered, appropriate action to reduce speed of the running train is taken.

We have used four test scenarios covering central situations occurring in the BART case study. If one wants to put more resources into the BART development with USE, further test cases covering more situations could be examined in much the same way.

Future work will consider the further development of USE. The results of evaluating pre- and postconditions in the command window could be represented in the GUI. Invoking and finishing operations could also be supported interactively. In order to increase the computational power of operations one could consider programming language-like features, i.e. conditionals and loops, for the command interpretation.

References

[ABB+00] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I.P. de Guzmán, G. Brewka, and L. M. Pereira, editors, *Proc. 8th European Workshop on Logics in AI (JELIA)*, LNCS 1919, pages 21–36. Springer-Verlag, 2000.

[Bol02] Boldsoft. The Boldsoft OCL Tool Model Run. `www.boldsoft.com`, Boldsoft, Stockholm, 2002.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, Redwood City, 1994.

[GR02] Martin Gogolla and Mark Richters. Development of UML Descriptions with USE. In A Min Tjoa, Hassan Shafazand, and K. Badie, editors, *Proc. 1st Eurasian Conf. Information and Communication Technology (EURASIA'2002)*. Springer, Berlin, LNCS, 2002.

[HDF00] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 278–293. Springer, 2000.

[JCJO92] Ivar Jacobsen, Magnus Christerson, Patrik Jonsson, and G. G. Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

[OMG03] OMG, editor. *OMG Unified Modeling Language Specification, Version 1.5*. OMG, March 2003. OMG Document `formal/03-03-01`, `www.omg.org`.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[RG99] Mark Richters and Martin Gogolla. A Metamodel for OCL. In Robert France and Bernhard Rumpe, editors, *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)*, pages 156–171. Springer, Berlin, LNCS 1723, 1999.

[RG01] Mark Richters and Martin Gogolla. OCL - Syntax, Semantics and Tools. In Tony Clark and Jos Warmer, editors, *Advances in Object Modelling with the OCL*, pages 43–69. Springer, Berlin, LNCS 2263, 2001.

[Ric01] Mark Richters. A UML-based Specification Environment, 2001. `http://www.db.informatik.uni-bremen.de/projects/USE`.

[WK98] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[ZG03] Paul Ziemann and Martin Gogolla. Validating OCL specifications with the USE tool - an example based on the BART case study. In Thomas Arts and Wan Fokkink, editors, *Proc. 8th Int. Workshop Formal Methods for Industrial Critical Systems (FMICS'2003)*, volume 80 of *ENTCS*. Elsevier, 2003.