Electronic Notes in Theoretical Computer Science 80 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume80.html 13 pages

Validating OCL Specifications with the USE Tool—An Example Based on the BART Case Study

Paul Ziemann [a,1] Martin Gogolla [a,2]

[a] *Department of Computer Science*
University of Bremen
Bremen, Germany

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 as well as pre- and postconditions. This paper explains the functionality of the UML Specification Environment USE which allows to validate UML and OCL descriptions.

The paper shows that central safety properties of the train system described in the well-known BART case study can be expressed with OCL. Test cases embodying central aspects of this train system can be formulated within the USE system. It can be shown that the safety properties are satisfied by the test cases examined.

Key words: UML, OCL, tool, validation, case study, testing

1 Introduction

The Unified Modeling Language (UML) [7] is regarded today as a very important standard for the development of software systems. UML originated from its main predecessor approaches Booch [3], OMT [11], and OOSE [6]. It 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) [7,12] is part of standard UML, but up to now it is not supported by most commercial tools. OCL is a specification language supporting and enriching the UML with textual details which cannot

[1] Email: ziemann@informatik.uni-bremen.de
[2] Email: gogolla@informatik.uni-bremen.de

be expressed in diagrammatic form. OCL allows to precisely describe system structure by invariants and system behavior by pre- and postconditions.

Tool support for OCL is beginning to develop. Among the first available tools was our system USE (UML Specification Environment), which is based on conceptional work on the formal semantics of the OCL, the required UML features [10], and work on the metamodel of OCL [9]. The main task of USE is to validate and verify specifications consisting of UML class diagrams together with OCL invariants and pre- and postconditions that are kept in a USE specification file (suffix `.use`). By validation, we mean that the developer can give test cases by means of object diagrams and manipulations of them and check whether the USE responses meet the intuition. By verification we mean that the test cases are formally checked with respect to invariants and pre- and postconditions. There are special USE commands for creating and manipulating object diagrams that can be accumulated in command files (with suffix `.cmd`).

Also part of the USE system is a so-called snapshot generator based on the language ASSL (A Snapshot Sequence Language), which not only offers means to change attribute values but also offers loop constructs and other helpful features. With so-called ASSL procedures, the manipulation can be done in a more flexible way. Such a procedure can also be seen as an implementation of an operation of a class.

Especially for specifications involving formal aspects, we regard validation systems as extremely helpful because they give feedback to developers in early development stages. Other tools for OCL include the commercial tool from Boldsoft [2], an OCL compiler [5], and the KeY tool [1].

The USE tool (excluding the generator extension) is developed as part of the PhD thesis of Mark Richters [8]. The OCL 2.0 submission [13] that is developed as an answer to the UML 2.0 Request for Proposals for OCL includes a part regarding the formal semantics of UML/OCL which is based on the mentioned PhD thesis. Therefore, the upcoming version of UML/OCL does not implicate complex updates of the USE tool—the new version of UML/OCL will rather be closer to the tool than the current version.

The BART case study description [14] informally describes a portion of the Advanced Automatic Train Control (AATC) system being developed for the Bay Area Rapid Transit (BART) system. BART provides commuter rail service for part of California's San Francisco bay area. The overall objective of the case study is to construct a system within the given infrastructure, that can control the speed and acceleration of trains in the system subject to various constraints, the central ones being the following: (1) A train should not enter a closed gate. (2) A train should never get so close to another train in front that if the train in front stopped suddenly, the (following) train would hit it. (3) A train should stay below the maximum speed that segment of track can handle.

In this paper, we demonstrate some of the features of the USE tool by specifying constraints for a part of the BART case study with OCL. We also specify OCL expressions calculating speed and acceleration for trains as well as updated positions. These expressions are embedded into ASSL procedures to automate testing. The specification is validated with several test scenarios in USE.

This paper is organized as follows. In Sect. 2, we present some parts from our specification consisting of a UML class diagram and a set of OCL invariants. In Sect. 3, we describe a simple control algorithm that calculates speed and acceleration for a train. In Sect. 4, an algorithm is described that lets the trains move according to their current speed and acceleration. Section 5 shows our test scenarios and how they are performed. The results finally are outlined in Sect. 6.

2 UML/OCL specification of the train system

Specifying the problem starts with the construction of the object model. The class operations, especially the operation for controlling the trains, are specified. OCL constraints specify the requirements each instantiation of the object model (i.e. each system state) must fulfill.

2.1 The basic object model

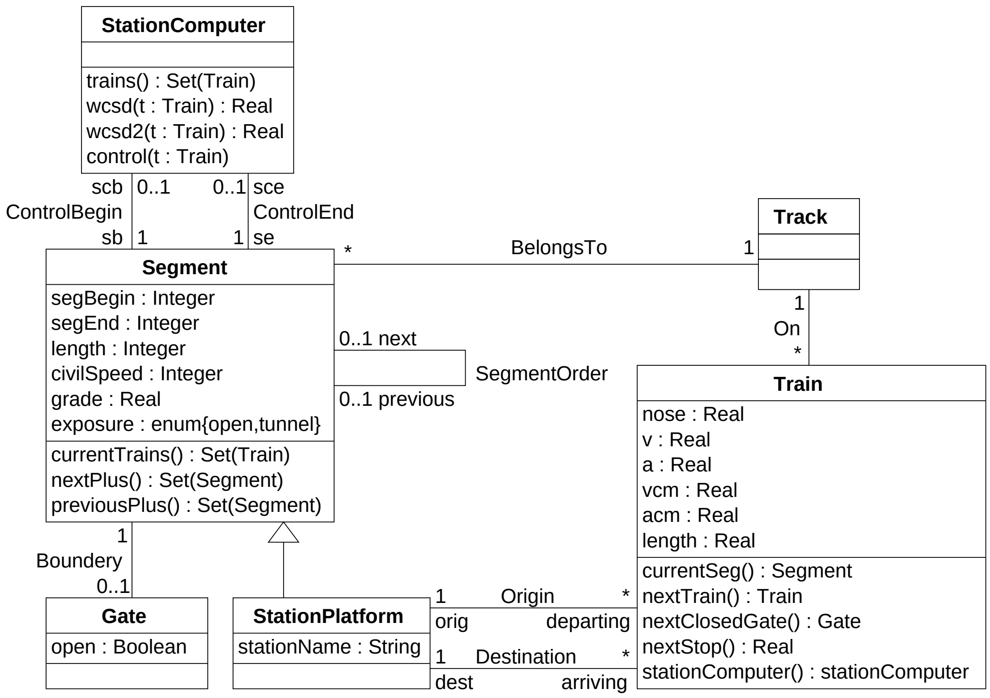

Fig. 1. UML object model of train system.

The UML class diagram in Fig. 1 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*. For the sake of simplicity, segments can only be passed by a train from the begin to the end, so trains can go only in one direction. 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` with which the commanded speed has to be achieved. 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 semantics of the operations in terms of OCL expressions will be published in [4]. Here we only give some short informal explanations.

`currentSeg()` computes the segment on which the train is currently located. `nextTrain()` finds the train which is in front of the given train. This is the train it has to be avoided to crash into. `nextClosedGate()` gives the next coming gate that is not open. 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 nearest. The operation `nextStop()` calculates the absolute position of this place. The station computer that is responsible for the train changes depending on the position. The operation `stationComputer()` results in the currently responsible computer.

The operation `nextPlus()` applied on a segment results in a sequence that includes the segment itself and all next coming segments. The operation `previousPlus()` analogously gives the previous segments. `currentTrains()` is the inverse of the operation `currentSeg()` of class *Train*. It gives all trains that are currently located on the segment.

The operation `trains()` of class *StationComputer* is the inverse of the operation `stationComputer()` of class *Train*. It gives all the trains the station computer is currently responsible for. `wcsd()` calculates the worst case stopping distance of a train `t`. The distance computed by `wcsd2()` is an even more pessimistic stopping distance. `control(t: Train)` computes a new commanded acceleration and speed for a train `t`. An algorithm for this operation is explained in Sect. 3.

2.2 OCL constraints on the train system

We define the following OCL constraints for the system.

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

```
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)
```

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 next 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", i.e. the civil speed.

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

"A train should not enter a closed gate". 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. Remember that the position of a gate is at the end of the segment it bounds.

```
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". 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)
```

3 A simple control algorithm

The ASSL procedure `control(t: Train)` implements the operation of class *StationComputer*. We do not show the code of this implementation here but rather explain the algorithm in an informal way.

The operation calculates for a train `t` a new commanded speed `vcm` and commanded acceleration `acm` with which `vcm` has to be achieved. These values are calculated by OCL expressions and have to respect the constraints of the train system including `civilSpeedSafety`, `closedGateSafety` and `crashSafety`. After calculating the values the corresponding attributes of the train are set.

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 in length) worst case stopping distance `wcsd2(t)`. Only civil speeds and obstacles within this range are considered.

The commanded speed `vcm` 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.

To determine the value of `acm`, we calculate two auxiliary values `acm-CivilSpeed` and `acmNextStop` at first:

With cs_{min} being the lowest civil speed within the range and *seg* being the first segment within the range with cs_{min} as civil speed, `acmCivilSpeed` is the acceleration needed to achieve a speed 2 mph below cs_{min} at the beginning of segment *seg*. 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 mphps.

If the next stop is out of the range, `acmNextStop` is set to the current acceleration incremented by 0.5 mphps. 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.

`acm` is set to the minimum of `acmCivilSpeed` and `acmNextStop`. The acceleration is adjusted properly to be in the span of physically possible accelerations. Finally, if the commanded speed is zero and the train is already slower than 0.5 mph, the train is commanded to stop with an acceleration of -2 mphps. Remember that `acm` is just the *commanded* acceleration *for reaching* `vcm`. When `vcm` is already reached, the actual acceleration `a` of the train is (near to) zero, no matter of the value of `acm`.

4 Moving the trains

In order to be able to validate our specification with the USE tool, we have to simulate the progression of the train system. This requires a procedure that modifies the attributes of the existing trains accordingly—the ASSL procedure `move()` serves this purpose: it sets the attributes of the trains as they would be after 0.5 seconds have gone by. `move()` does not correspond with any operation of the object model.

We do not show the complete ASSL definition of `move()` here, but the framework looks like this:

```
procedure move()
var
  delta: Real,
  gradeacc: Real;
begin
  delta := [0.5];
  for t: Train in
    [Train.allInstances->asSequence()]
  begin
    [t].nose := [<OCL-expression>];
    [t].v := [<OCL-expression>];
    [t].a := [<OCL-expression>];
  end;
end;
```

By using a `for` loop, the procedure sets the attributes `nose` (position), `v` (speed), and `a` (acceleration) of all existing trains to new values that are calculated by the OCL expressions.

The algorithm is as follows: 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" once it has stopped.

Then the new speed `v` is computed. The train cannot go backwards, so if the result of the computation is negative, the speed is set to zero and the train stops. The speed is also set to zero if both `v` and `vcm` are zero.

Finally, the acceleration is set. 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 commanded acceleration.

5 Validation with USE

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

5.1 Preparation

We create a USE specification that includes the object model (see Fig. 1), OCL constraints, and OCL specifications of the side effect-free operations. The two ASSL procedures `move()` and `control(t: Train)` are put into a file `procs.assl`. Next we create a command file `oneTrainThrough.cmd` that contains USE commands to create objects and links and to set attribute values according to the sample track that is included in [14]. In `oneTrainThrough.cmd`, 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 looks like the one shown in Fig. 2 except for the second train *Chattanooga* that is additionally included there (attributes are not shown).

The first thing we encounter after opening the specification in USE (with `open bart.use`) and executing the command file (with `read oneTrainThrough.cmd`) is that the constraint `Segment::fitting` is violated several times, i.e. the sample track 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
⋮
checked 10 invariants in 15.751s,
 1 failure.
```

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

In order to minimize typing when running a test, we create a command file `step.cmd` with the following content:

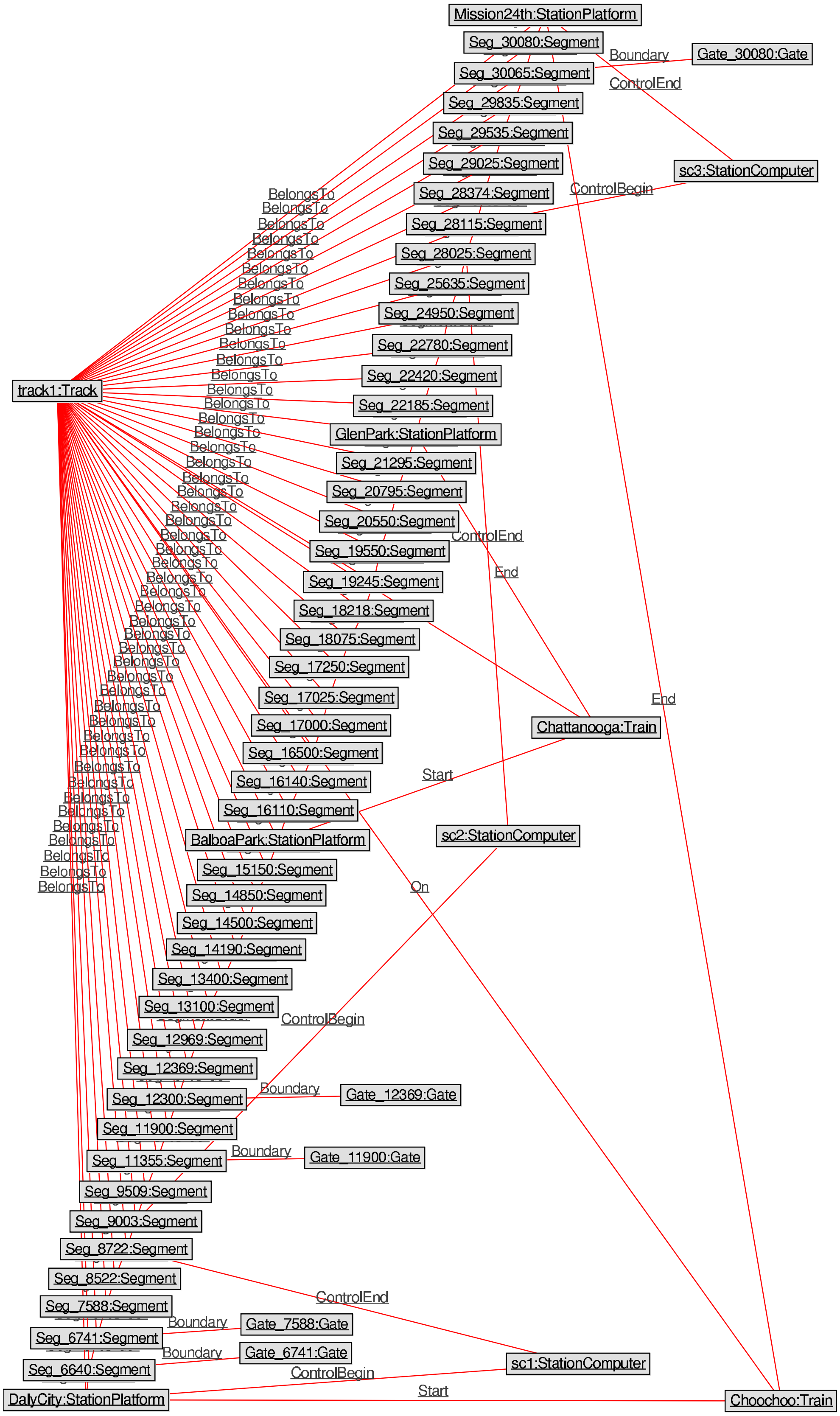

Fig. 2. Test scenario with two trains Choochoo and Chattanooga.

```
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*. In test runs with only one train, the controlling lines for the other are deleted for better efficiency. 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 several times the command `read step.cmd`, so we only need to execute `run.cmd` to run x steps.

5.2 The Test Scenarios

We want to run four different test scenarios, differing from each other in the number of existing trains and in the origin, destination and current location of the train(s). The objectdiagram in Fig. 2 shows the used portion of our test track with two trains on it. Since the diagram does not show attribute values, the positions of the trains can not be seen there.

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 mainly want to validate that the constraint `civilSpeed-Safety` 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 last 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.

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.

6 Conclusions

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

(i) Syntax errors. These errors of course can be found and corrected quickly.

(ii) Small mistakes like the inconsiderate use of `<` instead of `<=` or a forgotten algebraic sign.

(iii) Faults or inconsistencies in the case study description [14] that have been adopted in the formal specification. Examples of this kind are the faulty sample track, the questionable use of `vcm` instead of `v` in the calculation of 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.

(iv) Constants that are not adjusted properly. An example is the constant `0.5` in `control(t: Train)`. When this speed is reached while braking, the train has to perform a full brake (-2 mphps). The previous value of this constant turned out to be unsuitable.

(v) Semantic errors. 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), no constraint is finally violated in any of the test runs.

References

[1] 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. 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.

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

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

[4] M. Gogolla and P. Ziemann. Checking BART Test Scenarios with UML's Object Constraint Language. In *Formal Techniques for Embedded Distributed Systems: From Requirements to Detailed Design*. Kluwer Academic Publishers. To appear.

[5] H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In A. Evans, S. Kent, and B. 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.

[6] I. Jacobsen, M. Christerson, P. Jonsson, and G. G. Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

[7] OMG. *OMG Unified Modeling Language Specification, Version 1.5, March 2003*. Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`, 2003.

[8] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints.* PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

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

[10] M. Richters and M. Gogolla. OCL - Syntax, Semantics and Tools. In T. Clark and J. Warmer, editors, *Advances in Object Modelling with the OCL*, pages 43–69. Springer, Berlin, LNCS 2263, 2001.

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

[12] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998.

[13] J. Warmer, A. Kleppe, T. Clark, A. Ivner, J. Högström, M. Gogolla, M. Richters, H. Hussmann, S. Zschaler, S. Johnston, D. S. Frankel, and C. Bock. Object Constraint Language 2.0. Technical report, Submission to the OMG, 2001.

[14] V. L. Winter and S. Bhattacharya. *High Integrity Software.* Kluwer Academic Publishers Press., 2001.