Tool Support for Validating UML and OCL Models through Automatic Snapshot Generation

MARTIN GOGOLLA
and
MARK RICHTERS
and
JÖRN BOHLING
University of Bremen

This paper studies tool support for the testing and certification of UML and OCL models as supported by the validation tool USE. We describe the features available in the UML/OCL tool USE and extend its features by introducing a language for defining properties of desired snapshots and by showing how such snapshots are generated. We explain the functionality offered by the USE tool. In particular, we demonstrate how the diverse windows, e.g., object diagram, class invariant, class extent, or OCL expression evaluation window, support the development process in early phases of software development.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints and Procedures, functions, and subroutines*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Invariants, Mechanical verification, and Specification techniques*

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: UML Model, OCL Constraint, Snapshot, Validation, Certification

1. INTRODUCTION

Today, the Unified Modeling Language UML [OMG 2003] is regarded as the de-facto standard for software design. The Object Constraint Language (OCL) [Warmer and Kleppe 1998] plays an important part within the UML, and tool support for the OCL is developing (see the Dresden OCL compiler [Hussmann et al. 2000], the Boldsoft tool [Boldsoft 2002], work based on C++ in [Chiorean 2001], the KeY approach [Ahrendt et al. 2000], or the NEPTUNE tool [Canals et al. 2003]). One of the first systems supporting OCL was the UML Specification Environment (USE) [Richters and Gogolla 1998; 2001]. USE allows to check UML and OCL models by creating snapshots representing system states at a particular point in time with objects, attribute values, and links. These snapshots can be represented as UML object diagrams [Richters and Gogolla 2000].

However, up to now such snapshots had to be built by giving an explicit sequence of commands. The current work is motivated by the wish to produce snapshots in a more declarative style by specifying properties the snapshot has to fulfill. Therefore, we have developed the language ASSL (A Snapshot Sequence Language) allowing to form snapshots also by specifying properties and not merely by giving command sequences. Furthermore, the tool USE has been extended in order to load invariants dynamically. Thus USE and its ASSL extension allow to certify properties about UML models, i.e., developers can formally inspect properties for provided scenarios which they regard as relevant. Our ideas are introduced in the context of the USE tool, but the basic new points of our approach can also be employed in other tools.

Validation and testing in object-oriented software development has been recognized as being important for long (see, for example, [Ambler 1997]). Thus, our work has connections to other related approaches. The work in [Jézéquel et al. 1999] focuses on distribution while [Oliver and Kent 1999] has animation as its context. The approach in [Dupuy and du Bousquet 2000] emphasizes the importance of having multi-formalisms. [Muthiayen 2000] studies validation of UML models by means of theorem proving, in particular in connection with PVS.

Author Addresses: Martin Gogolla, Mark Richters, Jörn Bohling, University of Bremen, Computer Science Department, Database Systems Group, D-28334 Bremen, Germany, [gogolla|mr|joebo]@informatik.uni-bremen.de

Snapshots were also generated from dynamic models such as Use Cases or State Charts. For instance, [Fröhlich and Link 2000] derive a sequence of messages from Use Cases by transforming them into planning problems. The Alloy constraint analyzer [Jackson et al. 2000] is a tool that allows the generation of snapshots for a model based on relational logic. However, an Alloy model does not support significant UML and OCL features like attributes and its data types. Approaches concentrating on testing of UML models have been considered for special applications like smart cards [Martin 1999], and with emphasis on special UML language features like sequence diagrams [Graubmann and Rudolph 2000], collaboration diagrams [Abdurazik and Offutt 2000], or statechart diagrams [Offutt and Abdurazik 1999].

The structure of the rest of this paper is as follows. Section 2 gives an overview of an example scenario which is used throughout this paper. Section 3 introduces in detail the class diagram including operations and invariants on which we rely and the ASSL constructs needed for automatic snapshot generation. Section 4 shows how to apply ASSL elements to test cases and so-called validation cases. It illustrates how the USE tool helps in developing UML and OCL models. The paper ends with concluding remarks in Sect. 5.

2. OVERVIEW BY A DEPLOYMENT DIAGRAM

Figure 1 shows a UML deployment diagram picturing the files and their dependencies for the example scenario used in this paper. At the bottom, one recognizes the file `train.use` containing the USE model of the classes, associations, and invariants as indicated by the comments on both sides. On the next higher level, the names of two `invs` files are shown, each one containing one invariant. On the top, the file `train.assl` can be seen. An ASSL file can contain so-called procedures used for constructing snapshots consisting of objects and links. Below this file, four command files with ending `cmd` are displayed. Each command file is responsible for performing a test case or a so-called validation case. A command file first loads the `train.use` file into the USE system in order to make the classes, associations, and invariants known. A command file afterwards can call procedures from the `train.assl` file and thereby creates objects, attribute values, and links. In addition to using the invariants from the `use` file, a command file may dynamically load further invariants from `invs` files. These invariants determine the properties of the created objects and links in the snapshot.

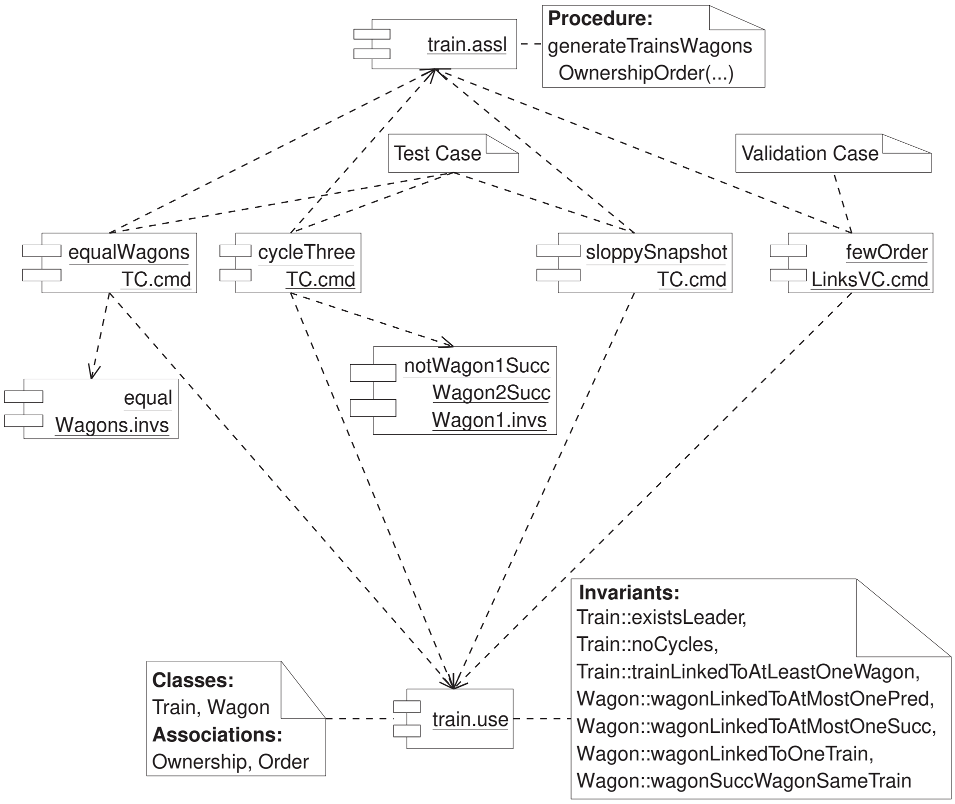

Figure 1. Deployment Diagram with `use`, `invs`, `cmd`, and `assl` Files

But let us come back to the test and validation cases. On the one hand, a test case has the task to certify that it is allowed to construct a certain snapshot on the basis of the already present invariants and the dynamically loaded invariants. On the other hand, a validation case certifies that a further property is a consequence of the already present invariants by showing that it is not possible to construct a snapshot satisfying the already present invariants under certain assumptions made. These ideas are detailed below.

3. USE TEXT, ASSL PROCEDURES, AND INVARIANTS

The USE tool and its ASSL extension are a lightweight formal method following the classification in [Agerholm and Larsen 1999]. That means, USE allows analysis of specifications rather than focusing entirely on correctness proofs. The purpose of the USE tool is to animate, test, and validate a UML class diagram and its OCL constraints and to focus on the early stages of the development process. Thus defects in a UML class diagram can be detected early before the implementation starts. Extending the USE tool with automated system state generation allows the generation of snapshots and, most important, testing of a large number of system states against additional OCL constraints. The purpose is still to find defects, thus USE and ASSL are still a lightweight formal method.

3.1 Class Diagram and Test and Validation Cases

The UML class diagram in Figure 2 shows an example from a railway context. This example is contained in the file `train.use` of Figure 1. Trains may own wagons. Wagons may be connected to other wagons thus possessing predecessor and successor wagons. We expect the following invariants to hold. A wagon is owned by exactly one train, and a train must consist of at least one wagon. A wagon has at most one predecessor and at most one successor wagon. These invariants could as well be stated as multiplicities in the class diagram. Furthermore, there should be one leading wagon in the train from which all other wagons are reachable by successor links, and there should be no cycles arising from the predecessor and successor connections. Lastly, a wagon and its successor wagon should belong to the same train.

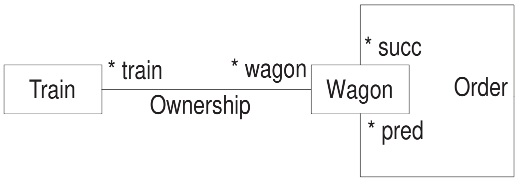

Figure 2. Class Diagram

As an example, let us consider the snapshot in the object diagram window in Figure 3. We see two trains each owning two wagons. The successor (and with this the inverse predecessor) links are determined by the OCL expression evaluation window below the object diagram window. The class invariant window shows that all the invariants hold in the depicted snapshot. This snapshot has been generated automatically by the command file to be explained below.

Coming back to the deployment diagram in Figure 1, the goal of the test case `equalWagonsTC` is to construct a simple valid snapshot where all trains have an equal number of wagons. The task of the test case `cycleThreeTC` is to show an invalid snapshot where three wagons arranged in a cyclic manner exist. The goal of the test case `sloppySnapshotTC` is to present a snapshot where most invariants fail. The validation case `fewOrderLinksVC` shows that, as a consequence from the specified invariants determining the multiplicities, not too many Order links should be generated.

3.2 USE Definitions

The following USE definitions show details not covered in Figure 2. First, two operations are defined by OCL expressions. The operation predPlus() is the transitive closure of the role pred. We only show the definition of predPlus(), the definition of succPlus() works completely analogously.

```
Wagon::predPlus():Set(Wagon)=
  self.predPlusOnSet(self.pred)
Wagon::predPlusOnSet(s:Set(Wagon)):Set(Wagon)=
  let oneStep:Set(Wagon)=s.pred->asSet in
  if oneStep->exists(w|s->excludes(w))
    then predPlusOnSet(s->union(oneStep))
    else s endif
```

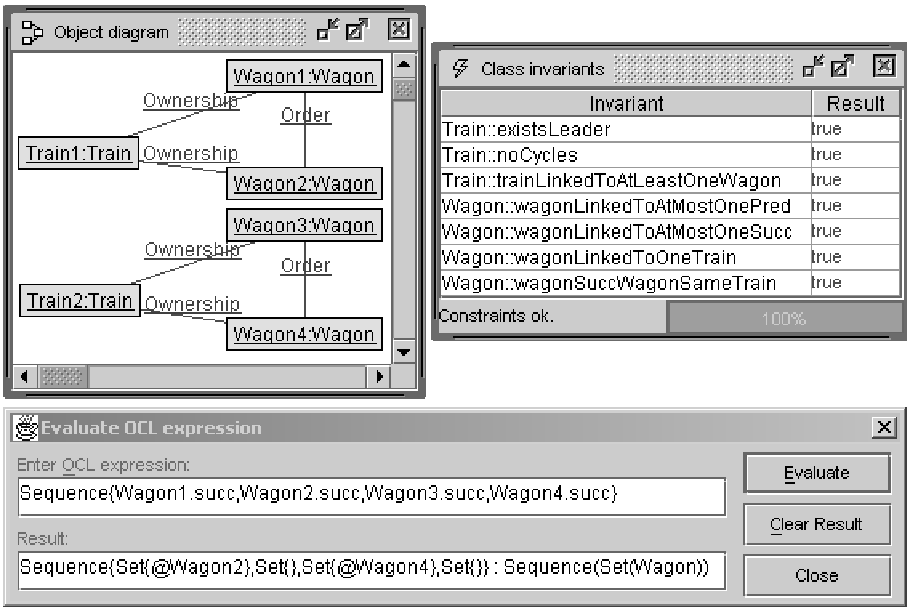

Figure 3. Snapshot for `equalWagonsTC`

Second, the invariants informally mentioned above are formally characterized by OCL formulae.

```
context Wagon inv wagonLinkedToOneTrain:
  self.train->size=1
context Train inv trainLinkedToAtLeastOneWagon:
  self.wagon->size>=1
context Wagon inv wagonLinkedToAtMostOnePred:
  self.pred->size<=1
context Wagon inv wagonLinkedToAtMostOneSucc:
  self.succ->size<=1
context Train inv existsLeader:
  self.wagon->exists(w | w.succPlus()->including(w)=self.wagon)
context Train inv noCycles:
  self.wagon->forAll(w|w.predPlus()->excludes(w))
context Wagon inv wagonSuccWagonSameTrain:
  self.succ->notEmpty implies self.train=self.succ.train->asSet
```

As mentioned above, instead of giving explicitly the first four invariants we could have used multiplicities in the class diagram. We have preferred the current solution in order to be able to explicitly show the result of evaluating these conditions, for example, in the class invariant window in Figure 3.

3.3 ASSL File

The task of an ASSL procedure is to construct a snapshot or part of a snapshot. A call to the following ASSL procedure `generateTrainsWagonsOwnershipOrder` generates trains, wagons, Ownership links and Order links. The parameters determine how many objects or links are established, for example, as many trains as indicated by the parameter `countTrains` are generated. The `CreateN` expressions return a sequence of length `countTrains` respectively `countWagons` of newly generated Train respectively Wagon objects. Afterwards, within a loop ranging over the number of links to be created, the procedure tries to generate a snapshot where all currently valid invariants hold. The first loop establishes the Ownership links, the second loop the Order links.

```
procedure generateTrainsWagonsOwnershipOrder
  (countTrains:Integer,countWagons:Integer,
   countOwnership:Integer,countOrder:Integer)
-- countOwnership<=countTrains*countWagons
-- countOrder<=countWagons*countWagons
var theTrains:Sequence(Train), aTrain:Train,
    theWagons:Sequence(Wagon),
```

```
    aWagon:Wagon, aWagon2:Wagon;
begin
theTrains:=CreateN(Train,[countTrains]);
theWagons:=CreateN(Wagon,[countWagons]);
-- generate countOwnership links in Ownership
for i:Integer in [Sequence{1..countOwnership}]
  begin
  aTrain:=Try([theTrains]);
  aWagon:=Try([theWagons->reject(w|w.train->includes(aTrain))]);
  Insert(Ownership,[aTrain],[aWagon]);
  end;
-- generate countOrder links in Order
for i:Integer in [Sequence{1..countOrder}]
  begin
  aWagon:=Try([theWagons]);
  aWagon2:=Try([theWagons->reject(w|w.pred->includes(aWagon))]);
  Insert(Order,[aWagon],[aWagon2]);
  end;
end;
```

For example, the first loop chooses a train from the newly created Train objects, and it chooses a wagon from the new Wagon objects. The `reject` expression in the Wagon assignment however guarantees that not a Wagon object is chosen which is already linked to the chosen Train object. Within the loop, the Train-Wagon pair is inserted into the Ownership association. The `Try` expressions provide a backtracking possibility: If the constructed snapshot does not satisfy all currently valid invariants, the procedure backtracks to the last Try expression offering still alternatives and takes another choice.

3.4 INVS Files

As already mentioned above, command files can dynamically load invariants in order to generate snapshots with certain properties. The command files will employ the invariant notWagon1SuccWagon2SuccWagon1 demanding that there do not exists two different wagons linked to each other in a cyclic way.

```
context Wagon inv notWagon1SuccWagon2SuccWagon1:
  not(Wagon.allInstances->exists(wagon1,wagon2|
        wagon1<>wagon2 and
        wagon1.succ->includes(wagon2) and
        wagon2.succ->includes(wagon1)))
```

The invariant equalWagons guarantees that all trains will have the same number of wagons.

```
context self:Train inv equalWagons:
  Train.allInstances->forAll(self2|self.wagon->size=self2.wagon->size)
```

4. TEST AND VALIDATION CASES

This section applies ASSL elements to test and validation cases. A test case certifies that it is allowed to construct a snapshot fulfilling the invariants. A validation case certifies that some property is a consequence of the currently valid invariants.

4.1 Test Case `equalWagonsTC`

The following protocol file shows commands given to the USE system and the responses from USE. First, the classes and invariants are made known by opening the USE definitions from file `train.use`. Afterwards, a further invariant leading the snapshot generation process into a special direction is loaded and thereby added to the currently valid set of invariants. Then the generation process is started by calling the procedure `generateTrainsWagonsOwnershipOrder` from file `train.assl` with the indicated parameters. USE is asked then to show the result. It reports that it has checked a number of snapshots and that a valid one was found. It shows the sequence of commands leading to that snapshot, and this command sequence is executed by stating the accept command.

```
use> open train.use
```

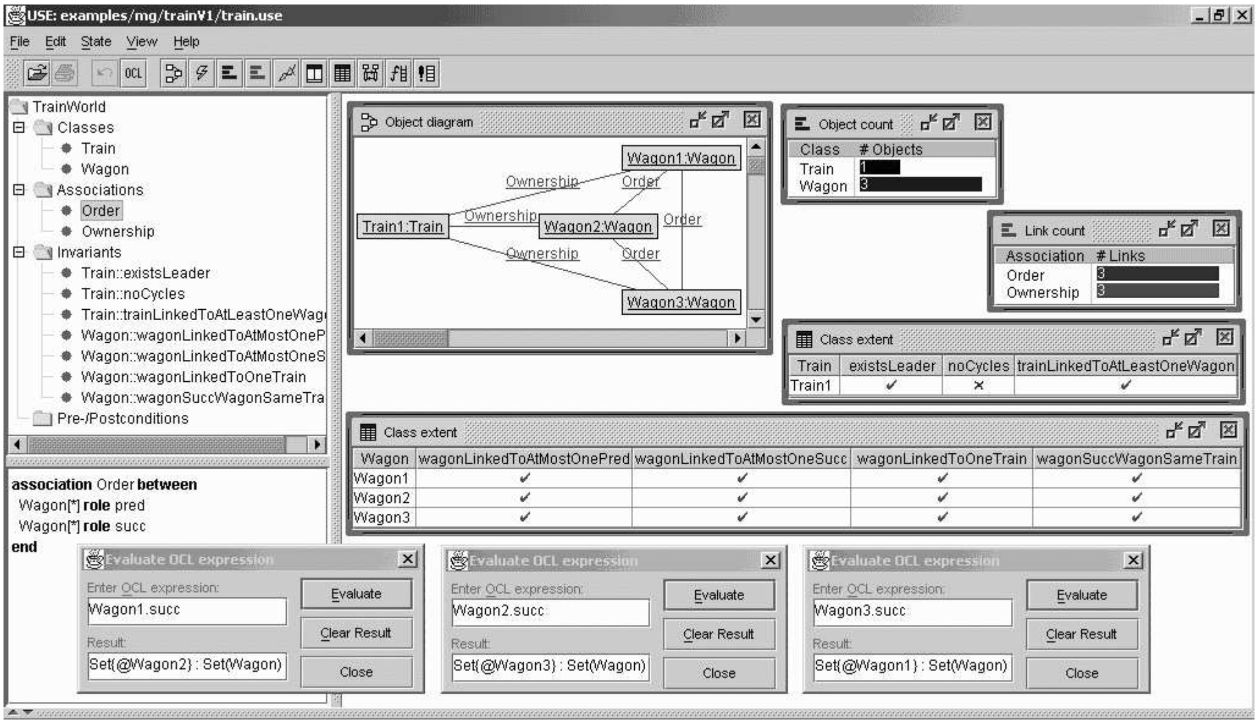

Figure 4. Snapshot for `cycleThreeTC`

```
use> gen load equalWagons.invs
     Added invariants: Train::equalWagons

use> gen start train.assl generateTrainsWagonsOwnershipOrder(2,4,4,2)
use> gen result
     Checked 5786 snapshots.
     Result: Valid state found.
     Commands to produce the valid state:
     !create Train1,Train2 : Train
     !create Wagon1,Wagon2,Wagon3,Wagon4 : Wagon
     !insert (Train1,Wagon1) into Ownership
     !insert (Train1,Wagon2) into Ownership
     !insert (Train2,Wagon3) into Ownership
     !insert (Train2,Wagon4) into Ownership
     !insert (Wagon1,Wagon2) into Order
     !insert (Wagon3,Wagon4) into Order
use> gen result accept
     Generated result (system state) accepted.
```

The reached snapshot is pictured in Figure 3. It is worth to mention that loading the additional invariant equalWagons has influence on the generated snapshot. Without loading it, one would end up with a snapshot where one train owns three wagons and the other train owns only one wagon.

4.2 Test Case `cycleThreeTC`

Let us consider a second test case. The protocol given below shows that again first the file `train.use` is opened. After this, one invariant from that file is deactivated by setting a so-called deactivate flag for that invariant. Thus for the snapshot to be generated it will be acceptable that it has cycles in the Order association. Then the before mentioned procedure is called with the indicated parameters: One train is to be generated, and three wagons, three Ownership links, and three Order links.

```
use> open train.use
```

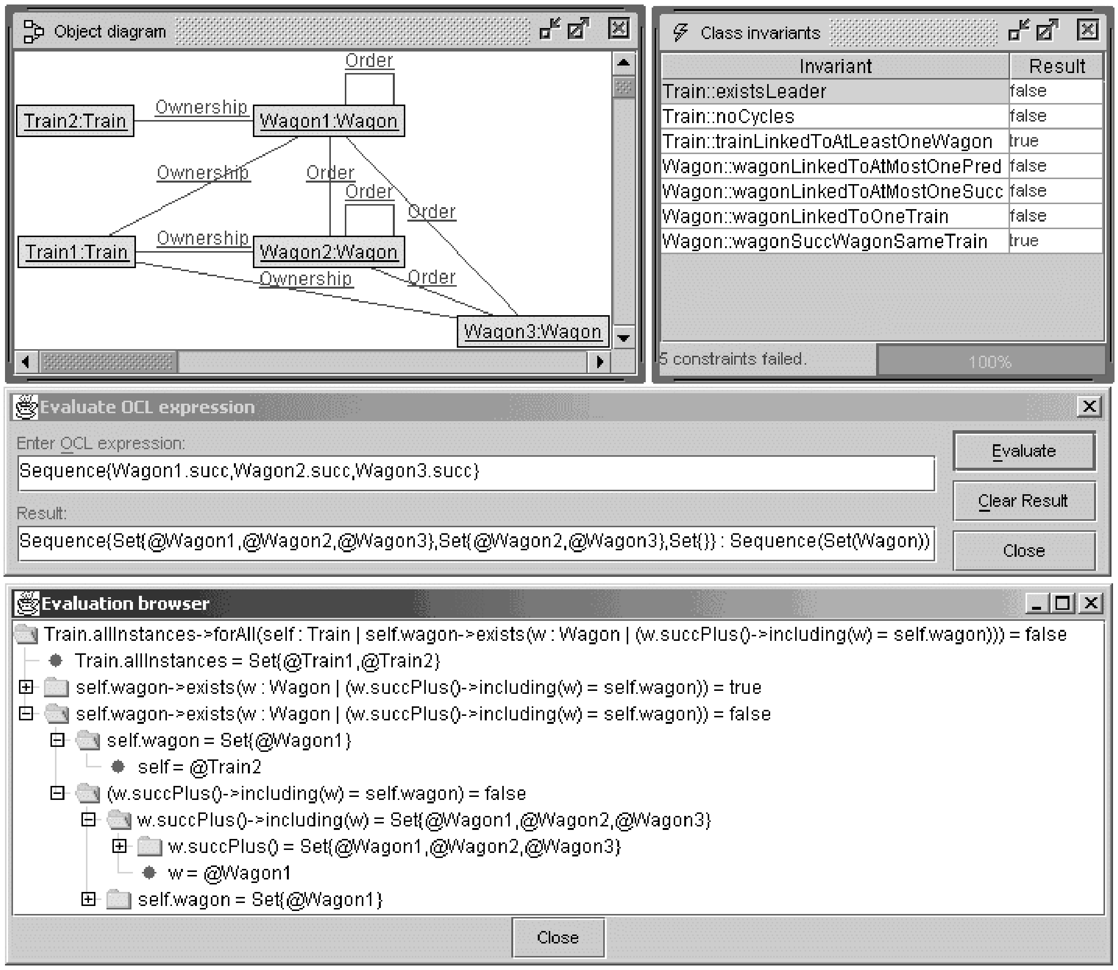

Figure 5. Snapshot for `sloppySnapshotTC`

```
use> gen flags Train::noCycles +d

use> gen start train.assl generateTrainsWagonsOwnershipOrder(1,3,3,3)
use> gen result
     Checked 89 snapshots.
     Result: Valid state found.
     Commands to produce the valid state:
     !create Train1 : Train
     !create Wagon1,Wagon2,Wagon3 : Wagon
     !insert (Train1,Wagon1) into Ownership
     !insert (Train1,Wagon2) into Ownership
     !insert (Train1,Wagon3) into Ownership
     !insert (Wagon1,Wagon2) into Order
     !insert (Wagon2,Wagon3) into Order
     !insert (Wagon3,Wagon1) into Order
use> gen result accept
     Generated result (system state) accepted.
```

Note that in a train with three wagons satisfying all invariants only two Order links can exist. Thus in order to receive a snapshot for this parameter combination it is essential to deactivate the invariant noCycles. As before, the list of commands leading to the detected snapshot is shown, and afterwards these commands are actually executed by stating the accept command.

The result is shown in the screenshot in Figure 4. This screenshot also gives an overview on central functionalities offered by USE. In the left part we see a project browser where classes, associations, invariants, and pre-

/postconditions (not described in this paper, but also supported by USE) are displayed. Classes, associations, and invariants can be shown in detail below the project browser. In the screenshot, we have selected the association Order and its details are pictured. To the right, we see a work panel where several windows have been opened and which reflect certain aspects of the reached snapshot. As before, the object diagram window directly shows the reached snapshot. The details for the Order links can be deduced from the OCL expression evaluation windows in the bottom of the screenshot: The successor of Wagon1 is Wagon2, the successor of Wagon2 is Wagon3, and the successor of Wagon3 is Wagon1. The work panel also shows two overview windows, the object count window and the link count window, giving the number of objects respectively links. Furthermore, two class extent windows show the currently existing objects in class Train and class Wagon together with the result of evaluating the class invariants. Note that all invariants except the invariant noCycles for the Train object Train1 are valid. On a color screen the ok-tick symbols are shown in green, and the violation-cross symbols are shown in red.

4.3 Test Case `sloppySnapshotTC`

The next test case introduces a quite sloppy snapshot. After opening the USE file, all invariants are deactivated, and the invariant notWagon1SuccWagon2SuccWagon1 disallowing a direct cycle in the Order association is loaded. The procedure is called by requiring two trains, three wagons, four Ownership links and five Order links to be generated. Note that under the numbers given some invariants must fail.

```
use> open train.use

use> gen flags Train::existsLeader +d
use> gen flags Train::noCycles +d
use> gen flags Train::trainLinkedToAtLeastOneWagon +d
use> gen flags Wagon::wagonLinkedToAtMostOnePred +d
use> gen flags Wagon::wagonLinkedToAtMostOneSucc +d
use> gen flags Wagon::wagonLinkedToOneTrain +d
use> gen flags Wagon::wagonSuccWagonSameTrain +d

use> gen load notWagon1SuccWagon2SuccWagon1.invs
     Added invariants: Wagon::notWagon1SuccWagon2SuccWagon1

use> gen start train.assl generateTrainsWagonsOwnershipOrder(2,3,4,5)
use> gen result
     Checked 7 snapshots.
     Result: Valid state found.
     Commands to produce the valid state:
     !create Train1,Train2 : Train
     !create Wagon1,Wagon2,Wagon3 : Wagon
     !insert (Train1,Wagon1) into Ownership
     !insert (Train1,Wagon2) into Ownership
     !insert (Train1,Wagon3) into Ownership
     !insert (Train2,Wagon1) into Ownership
     !insert (Wagon1,Wagon1) into Order
     !insert (Wagon1,Wagon2) into Order
     !insert (Wagon1,Wagon3) into Order
     !insert (Wagon2,Wagon2) into Order
     !insert (Wagon2,Wagon3) into Order
use> gen result accept
     Generated result (system state) accepted.
```

The result is displayed in Figure 5. The object diagram window shows the objects and their links. The OCL expression evaluation window allows to conclude that all Order links between different wagons go 'downward', e.g., the successor set of Wagon1 is {Wagon1, Wagon2, Wagon3}. The class invariant window reveals that in this snapshot only two from seven invariants hold. On the bottom of the screenshot we identify an evaluation browser window. This window has been opened by double-clicking the invariant existsLeader (this is indicated by the fact that existsLeader is shaded in grey). The evaluation browser window may be understood as a means for analyzing the fact that a certain invariant fails in the current snapshot. The class invariant window only states that the invariant existsLeader is false, but the evaluation browser points to the relevant parts in the snapshot that contribute to the fact that the invariant fails. In this case, we see that the object Train2 is responsible for this fact, because Train2.wagon = {Wagon1} but Wagon1.succPlus()−>including(Wagon1) = {Wagon1, Wagon2, Wagon3}. Thus the evaluation browser allows to understand and analyze invariant failure.

4.4 Validation Case `fewOrderLinksVC`

The last case whose protocol is given below tries to generate a snapshot with one train, two wagons, two Ownership links, and two Order links. The USE system reports that it has checked 24 spapshots, but that no valid one has been found.

```
use> open train.use

use> gen start train.assl generateTrainsWagonsOwnershipOrder(1,2,2,2)
use> gen result
     Checked 24 snapshots.
     Result: No valid state found.
```

Why does USE consider 24 snapshots? Let us look at procedure generateTrainsWagonsOwnershipOrder once again. We have only one train t1, so we do not get any backtracking possibility for the trains. However, within the first loop we have two possibilities for assigning the Ownership links to wagons, say w1 and w2: (1) Insert(Ownership,t1,w1); Insert(Ownership,t1,w2), and (2) Insert(Ownership,t1,w2); Insert(Ownership,t1,w1). Within the second loop, we must consider all possibilities for two Order links between wagons w1 and w2. These possibilities are pictured in Figure 6.

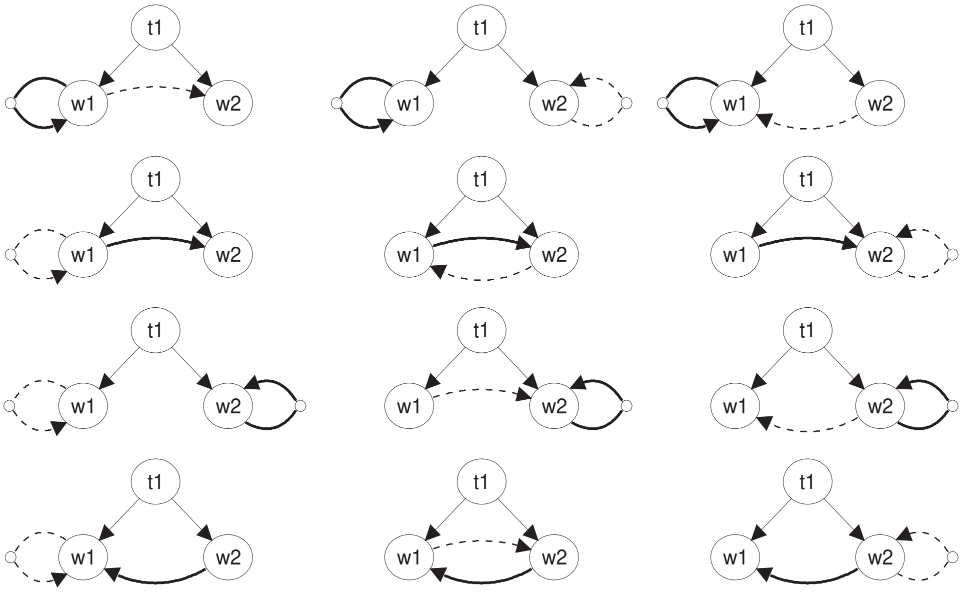

Figure 6. Illustration for `fewOrderLinksVC`

The first line in Figure 6 fixes the first link, the one between w1 and w1. This fixed link is displayed as a thick arrow. Under the assumption that this link is fixed, there are three possibilities for the second link: From w1 to w2, from w2 to w2, and from w2 to w1. Analogously, the second line fixes the first link as the one from w1 to w2, the third line fixes the link w2 to w2, and the fourth line the link from w2 to w1. Altogether, this makes 12 cases. Together with the two backtracking cases for the Ownership links, this makes up the 24 cases.
But what does this tell us? It formally shows and underpins our intuition that it is not possible to achieve a valid snapshot for one train and two wagons, if we demand the existence of two Order links and if we require all invariants to hold. In this sense, this validation case helps us in formally checking a property which we intuitively suspect. It is also possible in USE to generate these 24 cases from the given ASSL call and to represent these 24 cases as command sequences and resulting object diagrams.

5. CONCLUSION

We have extended the USE tool for automatic generation of complex snapshots. Snapshots represent system states consisting of objects possessing attribute values and links. We have employed OCL expressions for various tasks apart from its traditional tasks like definition of operations, invariants, pre- and postconditions: (1) OCL expressions are used in ASSL procedures (1.a) to reduce the search space for snapshots and (1.b) to formulate properties of the desired snapshot, and (2) OCL invariants are dynamically loaded during command file execution (2.a) to guide the snapshot search and (2.b) to certify an invariant implication. It is remarkable for us that a descriptive language like OCL can be used in task (1.a) for improving efficiency.

We have been considering larger snapshots as well and our approach seems to be scalable in this respect. We have built, for example, snapshots with about 200 objects and 300 links in acceptable execution time (about 30 seconds) with a slightly modified procedure. Our motivation for studying complex test cases in early development stages is to ensure the quality of the developed descriptions and to give very early feedback to the developer of models. We see also the possibility to carry over test and validation cases from the early phases to the later phases where the system implementation is built. For example, the 24 snapshots from the above validation case correspond to 24 negative test cases which could also be run against the implementation. Thus, we expect that the effort needed in the early phases will be rewarded by high quality outcomes in later phases.

REFERENCES

Abdurazik, A. and Offutt, J. 2000. Using UML collaboration diagrams for static checking and test generation. In *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*, A. Evans, S. Kent, and B. Selic, Eds. LNCS 1939. Springer, 383–395.

Agerholm, S. and Larsen, P. G. 1999. A Lightweight Approach to Formal Methods. In *Proc. Int. Workshop Current Trends in Applied Formal Methods (FM-TRENDS'1998)*, D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, Eds. LNCS 1641. Springer, Berlin, 168–183.

Ahrendt, W., Baar, T., Beckert, B., Giese, M., Habermalz, E., Hähnle, R., Menzel, W., and Schmitt, P. H. 2000. The KeY approach: Integrating object oriented design and formal verification. In *Proc. 8th European Workshop Logics in AI (JELIA'2000)*, M. Ojeda-Aciego, I. de Guzmán, G. Brewka, and L. M. Pereira, Eds. LNCS 1919. Springer, Berlin, 21–36.

Ambler, S. W. 1997. *Building Object Applications: Patterns, Architecture, Design, Construction, and Testing.* Prentice Hall.

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

Canals, A., Cassaing, Y., Jammes, A., Pomies, L., and Roblet, E. 2003. How You could Use NEPTUNE in the Modelling Process. *Journal of Object Technology 2,* 1, 69–83. `www.jot.fm`.

Chiorean, D. 2001. Using OCL Beyond Specifications. In *Proc. UML'2001 Workshop Rigorous Development*, A. Evans, R. France, A. Moreira, and B. Rumpe, Eds. Lecture Notes in Informatics (LNI), German Informatics Society, 57–68.

Dupuy, S. and du Bousquet, L. 2000. A multi-formalism approach for the validation of UML models. *Formal Aspects of Computing 12,* 4, 228–230.

Fröhlich, P. and Link, J. 2000. Automated Test Case Generation from Dynamic Models. In *Proc. 14th European Conf. Object-Oriented Programming (ECOOP'2000)*, E. Bertino, Ed. Springer, Berlin, LNCS 1850, 472–491.

Graubmann, P. and Rudolph, E. 2000. Hypermscs and sequence diagrams for use case modelling and testing. In *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*, A. Evans, S. Kent, and B. Selic, Eds. LNCS 1939. Springer, 32–46.

Hussmann, H., Demuth, B., and Finger, F. 2000. Modular architecture for a toolset supporting OCL. In *Proc. 3rd Int. Conference Unified Modeling Language (UML'2000)*, A. Evans, S. Kent, and B. Selic, Eds. LNCS 1939. Springer, 278–293.

Jackson, D., Schechter, I., and Shlyakhter, I. 2000. Alcoa: The Alloy constraint analyzer. In *Proc. Int. Conf. Software Engineering (ICSE'2000).* ACM, New York, 730–733.

Jézéquel, J.-M., Guennec, A. L., and Pennaneach, F. 1999. Validating distributed software modeled with the Unified Modeling Language. In *Proc. 1st Int. Workshop Unified Modeling Language (UML'1998)*, J. Bézivin and P.-A. Muller, Eds. LNCS 1618. Springer, 365–377.

Martin, H. 1999. Using test hypotheses to build a UML model of object-oriented smart card applications. In *Proc. Int. Conf. Software and Systems Engineering and their Applications (ICSSEA'1999)*, J.-C. Rault, Ed.

Muthiayen, D. 2000. Real-time reactive system development - a formal approach based on UML and PVS. Ph.D. thesis, Department of Computer Science at Concordia University, Montreal, Canada.

Offutt, J. and Abdurazik, A. 1999. Generating tests from UML specifications. In *Proc. 2nd Int. Conf. Unified Modeling Language (UML'1999)*, R. France and B. Rumpe, Eds. LNCS 1723. Springer, Berlin, 416–429.

Oliver, I. and Kent, S. 1999. Validation of object-oriented models using animation. In *Proc. EuroMicro, Vol. 2.* IEEE, Los Alamitos, 2237–2243.

OMG, Ed. 2003. *OMG Unified Modeling Language Specification, Version 1.5.* OMG. OMG Document `formal/03-03-01`, `www.omg.org`.

Richters, M. and Gogolla, M. 1998. On Formalizing the UML Object Constraint Language OCL. In *Proc. 17th Int. Conf. Conceptual Modeling (ER'1998)*, T.-W. Ling, S. Ram, and M. L. Lee, Eds. Springer, Berlin, LNCS 1507, 449–464.

Richters, M. and Gogolla, M. 2000. Validating UML Models and OCL Constraints. In *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*, A. Evans and S. Kent, Eds. Springer, Berlin, LNCS 1939, 265–277.

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

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