Consistency, Independence and Consequences in UML and OCL Models

Martin Gogolla, Mirco Kuhlmann, Lars Hamann

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
{gogolla|mk|lhamann}@informatik.uni-bremen.de

Abstract. Properties in UML models are frequently formulated as OCL invariants or OCL pre- and postconditions. The UML-based Specification Environment (USE) supports validation and to a certain degree verification of such properties. USE allows the developer to prove the consistency and independence of invariants by building automatically generated test cases. USE also assists the developer in checking consequences and making deductions from invariants by automatically constructing a set of test cases in form of model scenarios. Suspected deductions are either falsified by a counter test case or are shown to be valid in a fixed finite search space.

1 Introduction

In recent years, many proposals for improving software quality have put forward model-centric development in contrast to code-centric development. Modeling languages and standards like UML [15] including the OCL [19] and QVT [16] are cornerstones of model-driven approaches. Our work concentrates on OCL and UML class diagram features. Our tool USE (UML-based Specification Environment) supports the validation of UML models by building prototypical test cases in form of scenarios comprising UML object or sequence diagrams [10, 11]. One goal of USE is to derive properties of a UML design from these test scenarios.

In particular, USE is able to support the consistency and the independence of OCL constraints. USE also supports deductions from OCL constraints. *Consistency* of OCL invariants can be shown in USE by constructing a positive test case in form of an object or sequence diagram such that all invariants do hold. *Independence* of invariants means that no single invariant can be concluded from other stated invariants. This a property which helps to keep UML models small and focussed. Independence can be shown in a systematic way within USE by the construction of counter test cases. Checking *consequences* and drawing conclusions from OCL invariants is often needed when only basic properties are formulated as invariants and other more advanced properties are consequences from the more basic ones. Checking consequences is supported in USE also by building counter test scenarios or by showing that a property is valid in a fixed search space consisting of a possibly large number of UML object diagrams.

Our work has connections to other relevant approaches. Basic concepts for formal testing can be found in the pioneering paper [8]. It introduces essential concepts, e.g., the formal definition of test data selection. The tool presented in [17] allows the developer to perform static and dynamic model checking with focus on UML statecharts. UMLAnT [18] allows designers and testers to validate UML models in a way similar to xUnit tests. UMLAnT delegates the validation of invariants, pre- and postconditions to USE [11]. Automatic test generation based on subsets of UML and OCL are examined in [5] and [2], whereas runtime checking of JML assertions transformed from OCL constraints is treated in [1]. Approaches for static UML and OCL model verification without generating a system state can be found in [14], [4] and [6]. In [14], UML models and corresponding OCL expressions are translated into B specifications and, for OCL expressions, into B formal expressions. The transformation result is analyzed by an external tool which provides information about specification errors, e.g., a contradiction between invariants. The work in [4] focuses on static verification of dynamic model parts (sequence diagrams) with respect to the static specification. The KeY approach [6] targets Java Card applications which can be verified against UML and OCL models. Our approach for checking model properties and in particular for finding models has many similarities with the Alloy [13] approach. The approach in [12] is closely related to ours because it also deals with state spaces for testing purposes.

The structure of the rest of the paper is as follows. Section 2 introduces the running example we are going to use throughout. Section 3, 4 and 5 discuss in detail consistency, independence, and checking consequences within USE, respectively. The paper is finished with concluding remarks and future work.

2 Running Example

Our considerations regarding completeness, consistency and checking consequences in UML and OCL models are illustrated and motivated by an example model [3, 9] which describes trains, wagons and their formation. The model allows that trains can consist of ordered wagons, i.e., a wagon may have predecessor and successor wagons.

Figure 1 displays the user interface of USE which is divided into a project browser (left frame), a main window and a log window (lower frame). The project browser displays all model elements, i.e., UML classes and associations as well as OCL constraints which can be invariants or pre- and postconditions. Details of items selected in the project browser are shown below the project browser.

The main window may contain several views for inspecting the model and particularly its system states. The class diagram view depicts the structural part

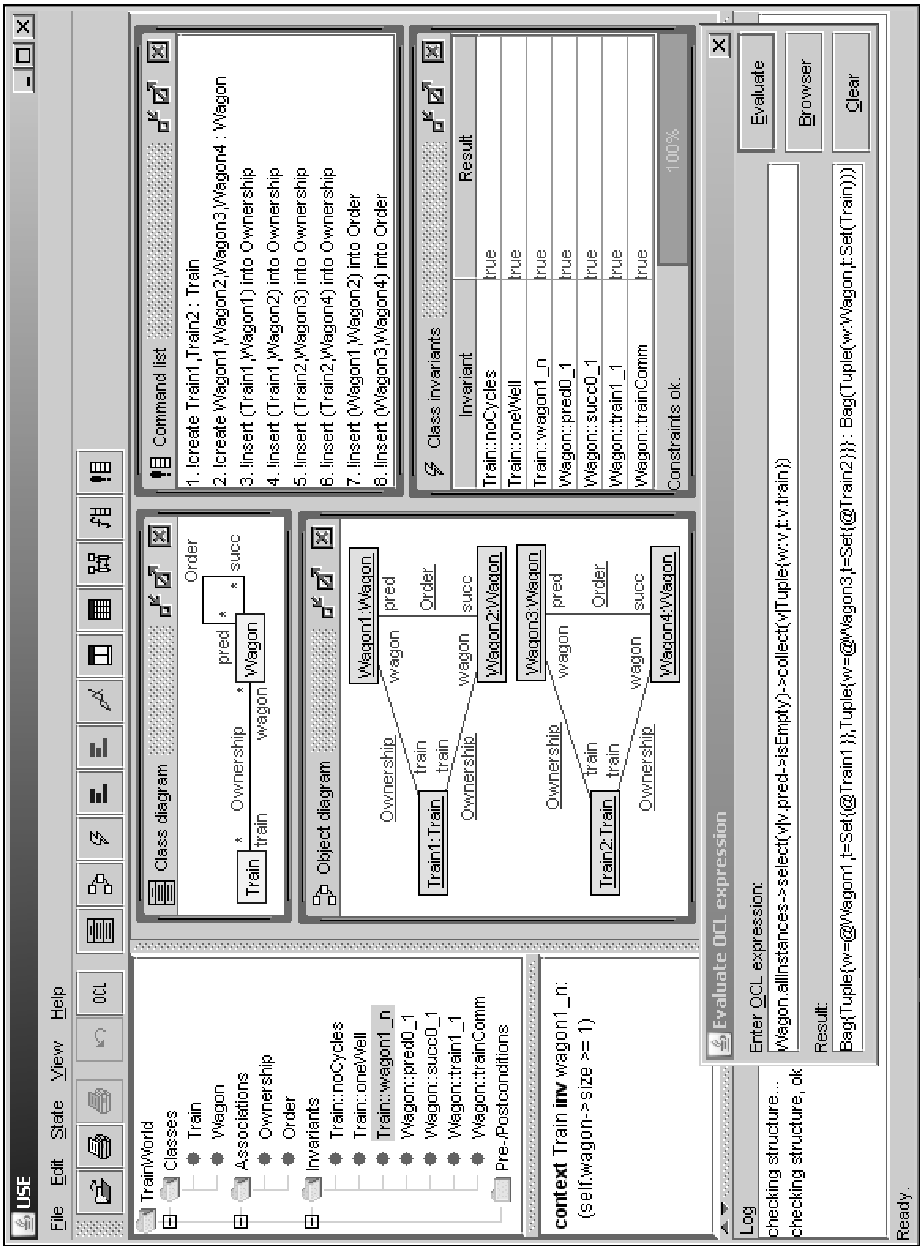

Fig. 1. USE Scenario Proving Consistency in the Example

of the model. It gives an overview on the defined classes, their associations and the constraining multiplicities.

The object diagram view visualizes the system state resulting from the commands shown right beside it. In this case it presents a valid state of the train model, because no constraint is violated. Each invariant expression evaluates to true and all links between the objects comply with the UML multiplicities which is stated in the log window.

USE allows the developer to query the system state interactively by entering an OCL expression into the OCL evaluation window. The result is directly reported. The example query asks for all wagons which do not have preceding wagons and collects their object identifiers together with the train identifiers which the wagon is part of.

As shown in the class diagram the class `Train` is related to the class `Wagon` via the association `Ownership`. A reflexive association `Order` permits connections between wagons. There are no restrictions concerning the multiplicities, although some of the invariants explained below express properties which could be stated in the class diagram as multiplicities. We have preferred to denote them explicitly as invariants, because this gives us the possibility to compare these multiplicities with other stated properties. The valid system state comprises two trains each connected with two ordered wagons.

Seven OCL invariants enrich the class diagram. All constraints are evaluated in context of a specific class. Accordingly an invariant is fulfilled if its body expression evaluates to true for each instance of the respective class, e.g., the invariant `wagon1_n` has to be fulfilled for the objects `Train1` and `Train2`.

The first four invariants add restrictions to the most general allowed multiplicities stated in the class diagram. Trains own one or more wagons (invariant `Train::wagon1_n`). Wagons belong to exactly one train (invariant `Wagon::train1_1`) and have at most one successor resp. one predecessor (invariants `Wagon::succ0_1` resp. `Wagon::pred0_1`).

```
context Train inv wagon1_n: self.wagon->size>=1
context Wagon inv train1_1: self.train->size=1
context Wagon inv succ0_1: self.succ->size<=1
context Wagon inv pred0_1: self.pred->size<=1
```

Three additional invariants characterize and restrict the train formation. Each train must have exactly one wagon (variable `well`) whose direct and indirect successors (operation `succPlus`) form a superset of the direct and indirect successors of all wagons in the train (invariant `Train::oneWell`). No wagon is allowed to appear in the set of its direct or indirect predecessors (invariant `Train::noCycles`). Finally, a commutativity property for wagons has to hold for each pair of wagons, i.e., if a wagon succeeds another, both have to be owned by the same train (invariant `Wagon::trainComm`).

```
context Train inv oneWell:
  self.wagon->one(well| self.wagon->forAll(w|
    well.succPlus()->includesAll(w.succPlus())))
```

```
context Train inv noCycles:
  self.wagon->forAll(w|w.predPlus()->excludes(w))
context w1:Wagon inv trainComm:
  Wagon.allInstances->forAll(w2|
    w1.succ->includes(w2) implies w1.train=w2.train)
```

All invariants make use of side effect free operations described within the classes `Train` and `Wagon`. The operation `succPlus` computes the direct and indirect wagon successors, i.e., the transitive closure of the successors of a wagon. It invokes the recursive auxiliary operation `succPlusOnSet` which collects the successors step-by-step. The operation `predPlus` is defined analogously. Later on we will introduce further, dynamically loaded invariants using the operation `allWagons` in the class `Train` which returns all wagons reachable from the owned wagons.

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

Train::allWagons():Set(Wagon)=
  self.wagon->union(self.wagon.predPlus()->asSet())->
    union(self.wagon.succPlus()->asSet())
```

In practice such models, especially more complex ones, are developed incrementally, i.e., the system properties are added one after the other, possibly by different developers. Ideally, the developer has to clarify after each addition or modification whether the model is still consistent and whether each invariant adds new information. Emerging correlations are often overlooked. This might result in models without clear borders between the model elements. In the following sections we discuss these elementary questions and the possibilities to answer them automatically by taking advantage of the USE tool.

3 Consistency

A consistent model is a desirable premise for further considerations. Consistency characterizes a model with no conflicts between any constraints, i.e., no constraint or set of constraints rules out other constraints. Within this paper we concentrate on relationships between OCL invariants.

We will continue with a more formal description. Each model M defines a set $I = (i_1, \dots i_n)$ of invariants. An invariant is evaluated in the context of a specific system state σ, i.e., the evaluation $\sigma(i)$ of invariant i yields true or false. We denote the set of all possible system states by $\sigma(M)$. A consistent model satisfies the following property.

$$\neg\exists i \in I \ (\exists J \subseteq I(\forall \sigma \in \sigma(M)(\bigwedge_{j \in J} \sigma(j) \Rightarrow \neg\sigma(i) \ )))$$

There must be no invariant i and no set of invariants J such that the invariant i is false whenever the invariants in J are true. The formula is equivalent to $\forall i \in I(\forall J \subseteq I(\exists \sigma \in \sigma(M)(\bigwedge_{j \in J} \sigma(j) \wedge \sigma(i))))$ which can be simplified to the basis of the subsequent proof of consistency.

$$\exists \sigma \in \sigma(M)(\sigma(i_1) \wedge \ldots \wedge \sigma(i_n))$$

Consequently we have to find a system state which fulfills all invariants in order to prove consistency. Due to the fact that each invariant expression is implicitly universally quantified, the empty system state always fulfills all invariants. Thus the formula is never false. A common way for avoiding this problem is to narrow the set of possible states by demanding the existence of at least one object for each class in each system state [7].

With the aid of the USE system and in particular the USE snapshot generator, it is possible to search a given state space for a non-empty state which fulfills all invariants. The generator executes user-defined ASSL (A Snapshot Sequence Language) procedures which make it possible to build up system states in a (pseudo) non-deterministic way. This feature can be applied for specifying a well-delimited set of system states (the set of states which can possibly be reached after invoking the procedure). These general considerations are made now more concrete by means of examples.

USE constructs a collection of system states (search space) as follows: After creating one system state the USE generator checks whether the state is valid in context of the UML and OCL constraints; if not, it backtracks and tries another way until a valid state was found or all possible search paths were considered. In our example, the following procedure is used for proving the consistency of the train model.

```
procedure genTrainsWagonsOwnershipOrder
 (countTrains:Integer,countWagons:Integer,
  countOwnership:Integer,countOrder:Integer)
var theTrains:Sequence(Train), aTrain:Train,
    theWagons:Sequence(Wagon),
    aWagon:Wagon, aWagon2:Wagon;
begin
theTrains:=CreateN(Train,[countTrains]);
theWagons:=CreateN(Wagon,[countWagons]);
for i:Integer in [Sequence{1..countOwnership}]
  begin
  aTrain:=Try([theTrains]);
```

```
13   aWagon:=Try([theWagons->reject(w|w.train->includes(aTrain))]);
14   Insert(Ownership,[aTrain],[aWagon]);
15   end;
16 for i:Integer in [Sequence{1..countOrder}]
17   begin
18   aWagon:=Try([theWagons]);
19   aWagon2:=Try([theWagons->reject(w|w.pred->includes(aWagon))]);
20   Insert(Order,[aWagon],[aWagon2]);
21   end;
22 end;
```

The procedure `genTrainsWagonsOwnershipOrder` has parameters for the number of trains, wagons, ownership links and order links to be created. After defining local variables (lines 4–6) the executable code begins. The generator is directed to create the specified number of trains and wagons (lines 8–9). With the following `for` loop, the specified number of ownership links is created. The generator selects a train and a wagon and inserts an ownership link between them in each iteration step.

The keyword `Try` indicates a backtrack point. The generator backtracks to it if the selected values prohibit the construction of a valid state. If the procedure results in an invalid state the generator backtracks to the last `Try` statement and chooses another value. If all alternative values for a `Try` were checked and no valid state was found, the generator backtracks to the next (outer) `Try`, e.g., when no wagon allows the creation of a valid system state (line 13) the generator jumps back to the upper `Try` and chooses another train (line 12).

For avoiding multiple link insertions between the same (train,wagon) pairs, wagons which are already connected to the train are rejected before the selection. Order links are inserted analogously. Here the links are established between wagon pairs.

The USE generator provides the possibility to load invariants which are not part of the model before starting the generation process. Through this it is possible to direct the result by adding respective constraints. The invariant `trainSizeBalanced` forbids system states with unbalanced trains: Two trains must own the same number of wagons or the numbers of wagons in two trains differ by one.

```
context t1:Train inv trainSizeBalanced: Train.allInstances->forAll(t2|
  t1<>t2 implies (t1.wagon->size-t2.wagon->size).abs<=1)
```

We use this constraint during the generation process to achieve a more attractive object diagram. If the generator does not find any valid state, we must start a run without the additional constraint to ensure that our considerations were not influenced by this additional constraint. A USE protocol explaining the invocation of the procedure `genTrainsWagonsOwnershipOrder` is shown below. The lines starting with `use>` indicate user input, the rest are responses from USE.

```
use> open train_wagon.use

use> gen load trainSizeBalanced.invs
     Added invariants: Train::trainSizeBalanced

use> gen start train_wagon.assl genTrainsWagonsOwnershipOrder(2,4,4,2)
use> gen result
     Random number generator was initialized with 6315.
     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.
```

After loading the train model, the USE generator is instructed to load the external invariant. Thereafter the procedure is invoked. The state space is controlled by the arguments. In the above case, the generator should only consider states with two trains, four wagons, four ownership links and two order links. When the generation process has finished the result can be checked. In this case, the generator has found a state fulfilling all model invariants in conjunction with the external one after checking 5786 states. The command sequence creating the state is shown afterwards. Once the result is accepted the state can be displayed in the object diagram view as pictured in Fig. 1. By this, the consistency of the train model has been proven.

This approach can be carried over to any UML model concentrating on a class diagram and OCL invariants. Depending on the state of knowledge about the constraints and their relations, an adequate procedure may be more general or more specific. The procedure above incorporates some knowledge about the model and its purpose. In the next section we use a more general procedure which results in a larger state space.

4 Independence

During development, the possibility of creating dependencies in the set of the constraints cannot be excluded. Dependent constraints, i.e., constraints whose fulfillment or violation depends on other constraints, can occur through the work of different developers or even through a single developer when the model evolves over a long time period.

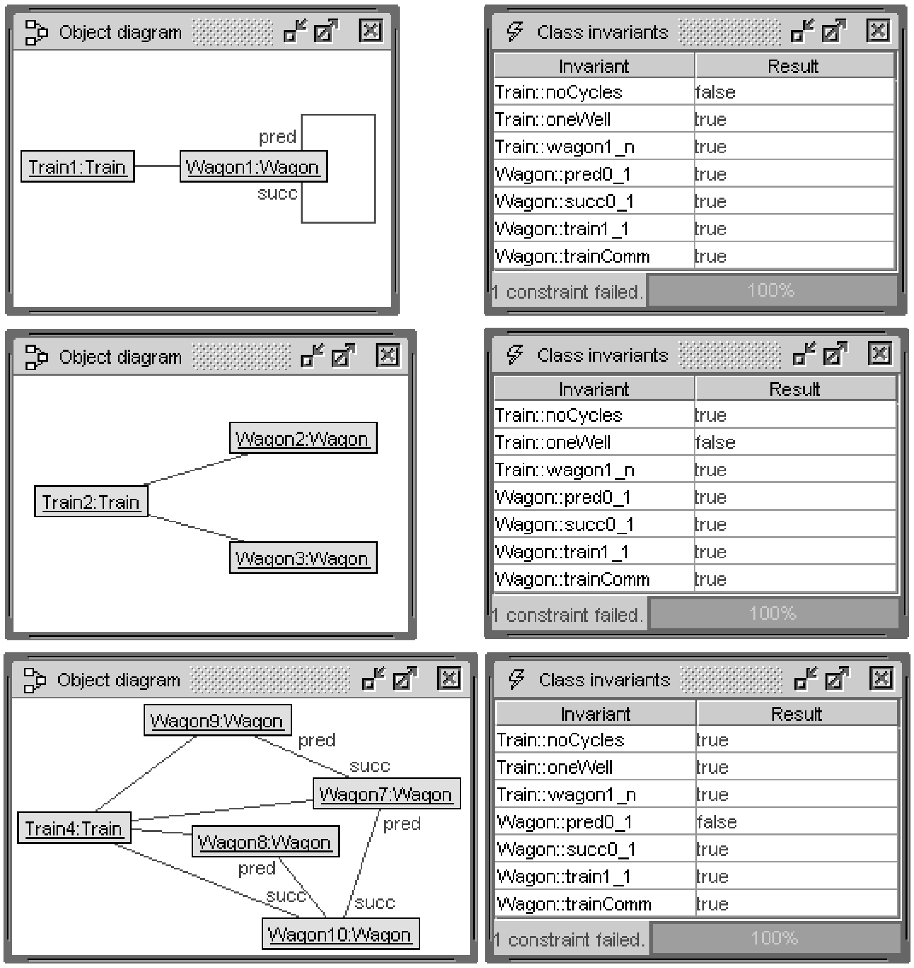

Fig. 2. USE Scenarios Proving Independence in the Example (Part A)

Developers should aim for independent constraints in their models, because this allows them to concentrate on the essential properties. Regarding OCL constraints, independence implies that each invariant is essential, i.e., the constraint cannot be removed from the model without loss of information. Formally, an invariant $i_k \in I$ is independent if and only if there is no set $J \subseteq I$ of invariants which implies i_k. Naturally, the invariant i_k which has to be proved to be independent should not be part of J.

$$\neg\exists J \subseteq I\ (i_k \notin J \wedge \forall\sigma \in \sigma(M)(\ \bigwedge_{j\in J} \sigma(j) \Rightarrow \sigma(i_k)\ ))$$

This is equivalent to $\forall J \subseteq I\ (i_k \in J \vee \exists \sigma \in \sigma(M)(\bigwedge_{j \in J} \sigma(j) \wedge \neg\sigma(i_k)))$ which can again be simplified to the following statement.

$$\exists \sigma \in \sigma(M)\ (\sigma(i_1) \wedge \ldots \wedge \sigma(i_{k-1}) \wedge \sigma(i_{k+1}) \wedge \ldots \wedge \sigma(i_n) \wedge \neg\sigma(i_k))$$

This means, we have to find a system state which conforms to all invariants except i_k and which also conforms to $\neg i_k$. If we are not sure whether an invariant is independent, it is not easy to find such a state. The USE generator can help again by searching a given state space.

Figures 2 and 3 show the proof results for six invariants of the train model. The generator has been configured to find system states violating one invariant at a time. The states are visualized on the left side and the invariant evaluation on the right side.

As mentioned before, the underlying ASSL procedure `genMaxCountTrainsMaxCountWagons` represents a generalized form of `genTrainsWagonsOwnershipOrder`. The formal parameters determine the maximum number of trains and wagons. The number of links cannot be affected manually.

```
 1 procedure genMaxCountTrainsMaxCountWagons
 2   (maxCountTrains:Integer,maxCountWagons:Integer)
 3 var theWagons:Sequence(Wagon), theTrains:Sequence(Train),
 4     actualCountTrains:Integer, actualCountWagons:Integer;
 5 begin
 6 actualCountTrains:=Try([Sequence{1..maxCountTrains}]);
 7 actualCountWagons:=Try([Sequence{1..maxCountWagons}]);
 8 theTrains:=CreateN(Train,[actualCountTrains]);
 9 theWagons:=CreateN(Wagon,[actualCountWagons]);
10 Try(Ownership,[theTrains],[theWagons]);
11 Try(Order,[theWagons],[theWagons]);
12 end;
```

First, the generator has to choose the actual number of trains and wagons, i.e., a number between one and the specified maximum number (lines 6–7). The generator takes different values if the chosen numbers turn out to be inappropriate for finding a valid state. The corresponding sets of trains and wagons are created in lines 8 and 9. Afterwards the keyword `Try` is used for the link insertion process. With line 10 all possible `Ownership` link configurations are tried. The generator starts with the empty link set, then it tries all link sets with one link, then all links sets with two links and so on. Line 11 describes the same approach for `Order` links. These `Try` statements are crucial, because in the worst case the generator has to try all possible combinations.

Before we turn to further considerations concerning the ASSL procedure, we use it for proving in an exemplary way the independence of the invariant `Train::noCycles` and thus demonstrate the general approach.

```
use> open train_wagon.use
```

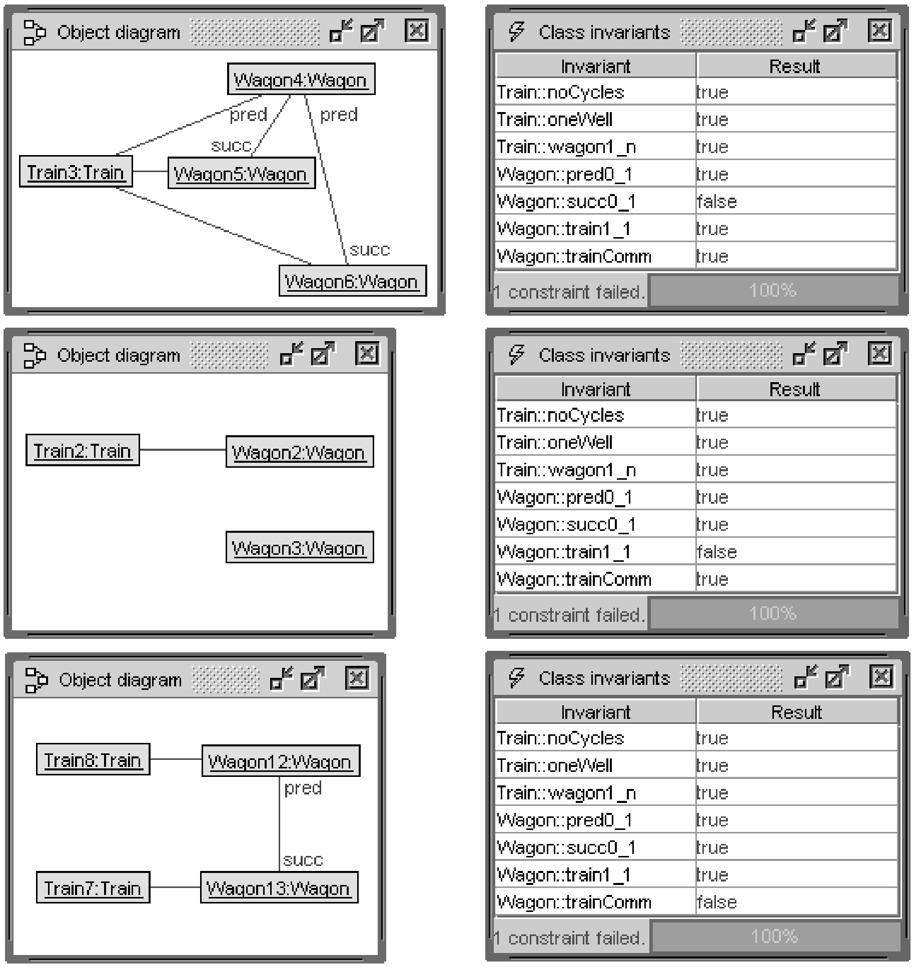

Fig. 3. USE Scenarios Proving Independence in the Example (Part B)

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

use> gen start train_wagon.assl genMaxCountTrainsMaxCountWagons(2,4)
use> gen result
     Random number generator was initialized with 9864.
     Checked 4 snapshots.
     Result: Valid state found.
     Commands to produce the valid state:
     !create Train1 : Train
     !create Wagon1 : Wagon
     !insert (Train1,Wagon1) into Ownership
     !insert (Wagon1,Wagon1) into Order
```

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

The first generator command sets a flag which negates the invariant `Train::noCycles`. After that, the generator is invoked with a small state space (at most two trains and four wagons). The result shows that this space was adequate for the proof as shown in the upper object diagram in Fig. 2. The invariant `Train::noCycles` is independent, because it is the only invariant which forbids the created system state (a train with one wagon which is its own predecessor). The invariant adds essential information to the model.

The independence proofs for five other invariants are treated analogously to `Train::noCycles`. The ASSL procedure is always invoked with the same arguments. Figures 2 and 3 show all proof results in form of specific system states.

For the invariant `Train::wagon1_n` it is not possible to find an appropriate system state with the aforementioned configuration. The generator has checked 17862988 states representing the defined state space.

```
use> open train_wagon.use

use> gen flags Train::wagon1_n +n

use> gen start train_wagon.assl genMaxCountTrainsMaxCountWagons(2,4)
use> gen result
     Random number generator was initialized with 5785.
     Checked 17862988 snapshots.
     Result: No valid state found.
```

The size of the state space results from the procedure's arguments. See the formula below for an exact calculation. The generator tries all numbers for trains and wagons from `1` to `maxCountTrains` resp. `maxCountWagons`. The variables t and w represent the actual number of trains and wagons. There are 2^w possibilities to connect a train with w wagons. t trains result in a total of 2^{wt} combinations. Analogously each wagon can be a predecessor of another one. Altogether this fact results in 2^{w^w} possible combinations.

$$\sum_{t=1}^{t=maxCountTrains} \left(\sum_{w=1}^{w=maxCountWagons} 2^{wt} \cdot 2^{w^w} \right)$$

The fact that no valid state was found does not prove a dependence in general, but for the given state space. So we get a strong indication for the dependence between `Train::wagon1_n` and a subset of the other invariants.

A closer examination of the invariants reveals a direct implication of `Train::wagon1_n` by `Train::oneWell`. The latter requires exactly one wagon of the current train to fulfill a specific boolean OCL expression. This implies the existence of at least one wagon linked to the train. It is possible to check this dependence by running again the procedure with the following generator configuration.

```
use> open train_wagon.use

use> gen flags Train::wagon1_n  +n

use> gen flags Train::noCycles  +d
use> gen flags Wagon::pred0_1   +d
use> gen flags Wagon::succ0_1   +d
use> gen flags Wagon::train1_1  +d
use> gen flags Wagon::trainComm +d

use> gen flags Train::oneWell   -d

use> gen start train_wagon.assl genMaxCountTrainsMaxCountWagons(2,4)

use> gen result
     Random number generator was initialized with 4575.
     Checked 17862988 snapshots.
     Result: No valid state found.
```

The invariant `Train::wagon1_n` was negated and the rest of the invariants were deactivated except for `Train::oneWell`. After the generation process the generator shows the same result as in the former run in which all invariants were embraced. This result strongly supports the assumption that the considered invariant is not independent, i.e., that `Train::wagon1_n` is a consequence of `Train::oneWell`.

5 Checking Consequences

The USE generator feature for loading external OCL invariants can be utilized yet in a different way. Besides using it to load constraints for guiding the search, it can be used for checking consequences from the model.

In general, only essential constraints, which have no dependencies, should be formulated within a model. But the constraints do not necessarily represent all important properties of the model directly. Instead, other important properties may follow from the defined constraints. For example, the important property written down below as the invariant `Train::distinctTrainsDistinctWagons` is assumed to be always fulfilled when a valid state is produced. The constraint forbids wagons from being shared between two trains.

```
context t1:Train inv distinctTrainsDistinctWagons:
  Train.allInstances->forAll(t2| t1<>t2 implies
    t1.allWagons()->intersection(t2.allWagons())->isEmpty())
```

The following USE protocol shows how the relationship between the explicit model constraints and the property under consideration can be explored with the USE generator. The corresponding additional invariant is negated directly after it has been loaded. After starting the search, the generator reports that it does not find a valid state.

```
use> open train_wagon.use

use> gen load distinctTrainsDistinctWagons.invs
     Added invariants: Train::distinctTrainsDistinctWagons

use> gen flags Train::distinctTrainsDistinctWagons +n

use> gen start train_wagon.assl genMaxCountTrainsMaxCountWagons(2,4)
use> gen result
     Random number generator was initialized with 9261.
     Checked 17862988 snapshots.
     Result: No valid state found.
```

Therefore, within the scope of a state space with at most two trains and four wagons, the added invariant has been proven to be not independent, i.e., to be implied by the model. In this example, this conclusion may be generalized, because additional trains and wagons do not entail significant changes to the state space except for its size.

Standard OCL does not distinguish between a basic invariant and an invariant which is an implication from other invariants. However, in order to label such implications one could introduce a special stereotype to express this.

6 Conclusion

This paper explains how consistency, independence and checking consequences in UML and OCL models can be handled with the USE tool on the basis of test scenarios. The OCL plays a central role in our approach: OCL is used for formulating constraints, for reducing the test search space (in ASSL procedures), for formulating search space properties (by employing dynamically loaded invariants) and for focusing deductions (by switching off unneeded invariants). Our approach is based on an interaction between building scenarios (through test cases) and studying system properties (through formulating properties and giving proofs).

A number of open questions remain for future work. We have to improve our approach by reducing the search space. Currently we are investigating how to guide the ASSL search by allowing constraints to be stated so that the ASSL search can be finished earlier in negative cases. Furthermore it is necessary to show more information about the search space as well as valid and invalid invariants during the search. The user interface of the ASSL search could be improved by allowing for a direct interaction. General ASSL procedures like `genMaxCountTrainsMaxCountWagons` with a universal scheme for state constructing could be generated in an automatic way. Last but not least, we want to employ efficient SAT solver technology for checking properties like consistency or independence.

Acknowledgement

The constructive remarks of the referees have helped us to improve and to polish our work.

References

[1] C. Avila, G. Flores, and Y. Cheon. A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking. In H.R. Arabnia and H. Reza, editors, Software *Engineering Research and Practice*, 403-408. CSREA Press, 2008.

[2] B.K. Aichernig and P.A. Salas. Test Case Generation by OCL Mutation and Constraint Solving. In *QSIC*, 64-71. IEEE Computer Society, 2005.

[3] J. Bohling. Snapshot Generation for Validating UML Class Diagrams (In German). *Diploma Thesis*, University of Bremen, Computer Science Department, 2001.

[4] A. Baruzzo and M. Comini. Static Verification of UML Model Consistency. In D. Hearnden, J.G. Süß, B. Baudry, and N. Rapin, editors, *Proc. 3rd Workshop Model Design and Validation*, 111-126. University of Queensland, October 2006.

[5] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A Subset of Precise UML for Model-Based Testing. In *A-MOST*, 95-104. ACM, 2007.

[6] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. Verification of Object-Oriented Software: The KeY Approach. *LNCS 4334*. Springer, 2007.

[7] J. Cabot and R. Clarisó. UML/OCL Verification in Practice. In S. Van Baelen, R. Van Der Straeten, and T. Mens, editors, ChaMDE 2008, *1st Int. Workshop Challenges in Model Driven Software Engineering*, 31-35, 2008. http://ssel.vub.ac.be/ChaMDE08/.

[8] M.-C. Gaudel. Testing can be Formal, Too. In TAPSOFT '95, *Proc. 6th Int. Joint Conf. CAAP/FASE Theory and Practice of Software Development*, 82-96. Springer, 1995.

[9] M. Gogolla, M. Richters, J. Bohling. Tool Support for Validating UML and OCL Models through Automatic Snapshot Generation. In J. Eloff, A. Engelbrecht, P. Kotze, M. Eloff, editors, *Proc. Annual Research Conf. South African Institute of Computer Scientists and Information Technologists*, 248-257. ACM, 2003.

[10] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Journal on Software and System Modeling*, 4(4):386-398, 2005.

[11] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27-34, 2007.

[12] J. Hughes. QuickCheck Testing for Fun and Profit. *Proc. 9th Int. Symp. Practical Aspects of Declarative Languages*, LNCS 4354, 1-32, 2007.

[13] D. Jackson. Software Abstractions: Logic, Language, and Analysis. *MIT Press*, 2006.

[14] R. Marcano-Kamenoff and N. Lévy. Transformation Rules of OCL Constraints into B Formal Expressions. In *CSDUML'2002, Workshop Critical Systems Development with UML*. 5th Int. Conf. Unified Modeling Language, Dresden, Germany, September 2002.

[15] OMG, editor. OMG Unified Modeling Language Specification, Version 2.0. *OMG*, 2004.

[16] OMG, editor. OMG Query, View and Transformation Specification (QVT). *OMG*, 2005.

[17] W. Shen, K. Compton, and J. Huggins. A Toolset for Supporting UML Static and Dynamic Model Checking. In *Proc. 16th IEEE Int. Conf. Automated Software Engineering (ASE)*, 147-152. IEEE Computer Society, 2001.

[18] T.D. Trong, S. Ghosh, R.B. France, M. Hamilton, and B. Wilkins. UMLAnT: An Eclipse Plugin for Animating and Testing UML Designs. In Eclipse'05, *Proc. 2005 OOPSLA Workshop Eclipse Technology eXchange*, 120-124, New York, NY, USA, 2005. ACM Press.

[19] J. Warmer and A. Kleppe. The Object Constraint Language: Precise Modeling with UML. *Addison-Wesley*, 2003. 2nd Edition.