Proving and Visualizing OCL Invariant Independence by Automatically Generated Test Cases

Martin Gogolla, Lars Hamann, and Mirco Kuhlmann

Universität Bremen, Informatik, AG Datenbanksysteme, D-28334 Bremen

Abstract. Within model-driven development, class invariants play a central role. An essential property of a collection of invariants is the independence of each single invariant, i.e., the invariant at hand cannot be deduced from the other invariants. The paper explains with three example models the details of an approach for automatically proving and representing invariant independence on the basis of a script constructing large test cases for the underlying model. Analysis of invariant independence is visualized by means of several diagrams like a 'test configuration and result' diagram, an 'invariant dependence detail' diagram, and an 'invariant dependence overview' diagram. The paper also discusses how to build the test case construction script in a systematic way. The test case construction script is written by the model developer, but a general construction frame for the script is outlined.

1 Introduction

Software development with model-driven techniques and languages like the Unified Modeling Language (UML) and the Object Constraint Language (OCL) has attracted many practitioners and researchers in recent years. In object-oriented approaches, class invariants play a central role. A crucial property of a collection of invariants is the independence of each single invariant, i.e., a considered invariant cannot be deduced from the other stated invariants. This property may be viewed as guaranteeing the absence of redundancy in an invariant set.

This paper explains an approach for automatically proving and representing invariant independence on the basis of a script constructing large test cases for the underlying model. When we use the notion script here, we always refer to such a test case construction script which belongs to the modeling level and not to a detail of the implementation. The test case construction script is written by the model developer, but a general construction frame for the script is outlined here in order to formulate a primary version of the script. Development is supported and visualized by means of several diagrams allowing to represent test configurations and results. The proposed diagrams can be generated in an automatic way from the executed test cases. This paper also explains how to write a test case construction script in general. Our work is done within the UML-based Specification Environment (USE), a modeling and validation system [GBR05,GBR07].

The USE details of all test cases in this paper may be found in [GHK10]. One motivation for introducing the various diagram forms here was to abstract away from test case details.

Invariant independence in models is an important property in object-oriented approaches, because if it holds, then all invariants have to be respected directly in an implementation; if it does not hold, then it is desirable to know, why it does not hold and which invariants contribute to redundancy. Our proposal helps in answering such questions and can be used to detect which subsets of the given invariants are independent. Invariant redundancy often occurs when important system properties are consequences of other technical requirements. Therefore techniques to detect dependencies are valuable.

The rest of the paper is structured as follows. Section 2 introduces the necessity for invariant negation, affirmation, and deactivation in test case construction scripts and shows the different diagram forms. Section 3 explains with a more complex example test case construction and shows more involved invariant dependencies. Section 4 discusses an example from the literature and puts forward the general template for a test case construction script. The paper ends with a discussion of related work and a conclusion. The generated test cases in [GHK10] may be seen as appendix-like add-on for the paper.

2 Basics of Invariant Independence

As an extension to the research presented in [GKH09] we here propose a more sophisticated way of producing test cases which prove the independence of invariants. This paper employs invariant deactivation, a technique which we did not apply before for showing invariant independence. By doing so, we are able to obtain more independence relationships. Furthermore we represent the different test case configurations and test results in diagrammatic form by means of so-called 'test configuration and result', 'invariant dependence detail', and 'invariant dependence overview' diagrams.

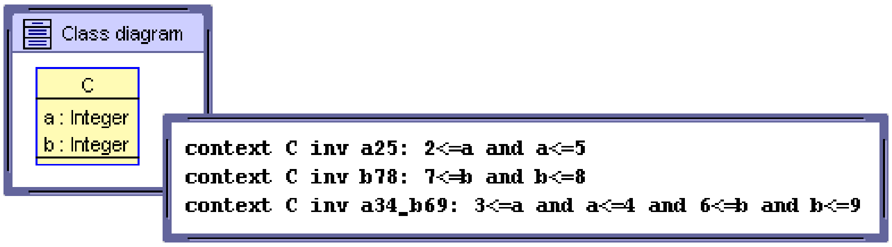


Fig. 1. Example Model CAB

The artificial example model shown in Fig. 1 possessing a single class C and two integer attributes a and b demonstrates the new features. The three invariants restrict the attribute values and partly overlap.

The following test case construction script enumerates possibilities for attribute values until a system state satisfying the current invariant configuration is found. A system state is also called snapshot because it represents the state of the system at a particular point during the dynamic development of the system. With the UML, object diagrams are employed for representing snapshots. An invariant can be negated, affirmed, or deactivated. The script language ASSL (A Snapshot Sequence Language [GBR05]) is a hybrid language which uses elements of imperative programming and OCL. OCL and ASSL are decidable because quantification is applied only on finite domains. In this example, the bounds for the attribute values can even be derived systematically from the constraints which use the constant 2 as the minimum integer and the constant 9 as the maximum integer. The bounds in the script extend the used bounds in the constraints by 1 on both interval ends in order to cover positive and negative test cases.

```
procedure genWorld()
var c: C;
begin
c:=[C.allInstances()->any(true)];
[c].a:=Try([Sequence{1..10}]);
[c].b:=Try([Sequence{1..10}]);
end;
```

The first step for the analysis of invariant independence is to call the test case generation script as often as determined by the number of invariants: If there exist n invariants, then the script is called n times; each call sets exactly one invariant as negated and all other invariants as affirmed:[1]

```
gen flags [a25 +n] [b78 -n] [a34_b69 -n]; gen start genWorld()
gen flags [a25 -n] [b78 +n] [a34_b69 -n]; gen start genWorld()
gen flags [a25 -n] [b78 -n] [a34_b69 +n]; gen start genWorld()
```

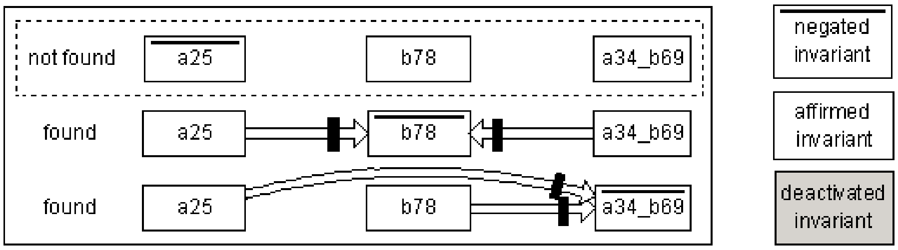


Fig. 2. Test Configuration and Test Result Diagram (CAB-A)

[1] In our current implementation, the syntax for invariant affirmation, negation and deactivation is slightly different; the flags for invariant affirmation, negation and deactivation must be stated for each each single invariant (see [GHK10]).

The different calls are represented in the Test Configuration and Test Result diagram in Fig. 2. Each call is shown as a line of invariants together with an indication expressing whether a snapshot has been found or not. Affirmed, negated and deactivated invariants are displayed differently (in the calls in Fig. 2 there are no deactivations, but such deactivations will show up below). Furthermore, if a snapshot has been found, an arrow indicating independence resp. non-implication may be drawn from each affirmed invariant to a negated invariant. For example, the upper-left non-implication arrow in Fig. 2 expresses that we have formally shown the fact (theorem): `not(a25 implies b78)`.

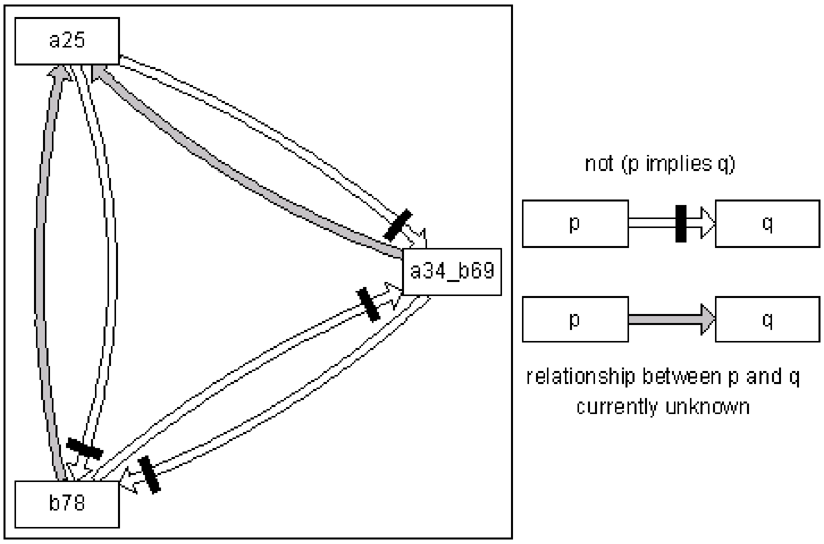


Fig. 3. Invariant Dependence Detail Diagram (CAB-B)

The currently known relationships between invariants are displayed in the Invariant Dependence Detail diagram in Fig. 3. All known non-implications are shown with non-implication arrows, and unknown relationships are shown as grey arrows.

In the second step for the analysis of invariant independence we study those test cases in more detail where the first step did not find snapshots. If a call with a negated invariant and with all other invariants affirmed was not successful, the configuration is extended so that all those test case generation calls are performed where at least one of the other invariants is deactivated and at least one is negated:

```
gen flags [a25 +n] [b78 +d] [a34_b69 -n]; gen start genWorld()
gen flags [a25 +n] [b78 -n] [a34_b69 +d]; gen start genWorld()
```

In the example this leads to the Test Configuration and Test Result diagram and the Invariant Dependence Detail diagram show in Fig. 4. The grey arrow in

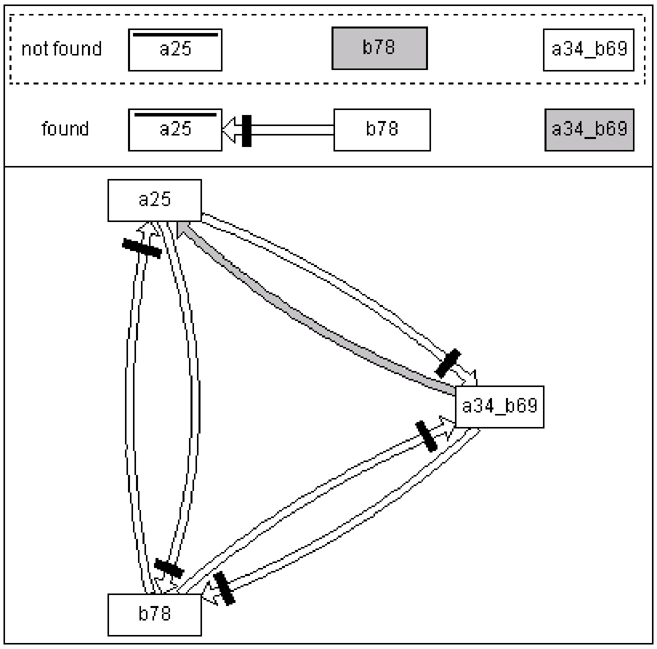


Fig. 4. Test Configuration and Test Result Diagram (CAB-C) as well as Updated Invariant Dependence Detail Diagram (CAB-D)

Fig. 3 from b78 to a25 has been updated to a non-implication arrow. This could only be achieved by invariant deactivation which was not considered in [GKH09].

It is worth to discuss why deactivation finds more counter examples: We assume one fixed ASSL script is employed; this script enumerates a particular finite set of system states, called the search space; first, the script is called as many times as invariants exist with exactly one invariant being negated; each single call may produce as result either a counter example or the answer that no system state exists in the search space under the current configuration for affirmed and negated invariants; in the last case the complete finite search space has been considered; only in this situation deactivation is used as a second step; within the search space, there may be counter examples which satisfy the current configuration of affirmed and negated invariants except the deactivated invariant which is violated; in order to find these counter examples, deactivation is beneficial.

The current knowledge about the invariant dependence is now aggregated into a single Invariant Dependence Overview diagram as shown in Fig. 5. All invariants are displayed with solid or dashed rectangles within a rounded rectangle. If an invariant is independent from all other invariants, it is also displayed outside the rounded rectangle with a non-implication arrow from the rounded rectangle to it (for example, `b78`). This arrow means that this invariant is independent from the complete set of invariants except the invariant itself: The independent invariant is shown with a dashed rectangle within the rounded rectangle (again, for example, `b78`) in order to distinguish it from an invariant which is not independent from the other invariants (for example, `a25`); naturally, we have, for example, `b78 implies b78` and we do not have `not(b78 implies b78)`. Further non-implications may be shown within the rounded rectangle (for example, the non-implication arrow from `b78` to `a25`). This kind of abstracting details of edges is similar to the abstraction made in statecharts [Har87] for displaying different transitions as a single transition.

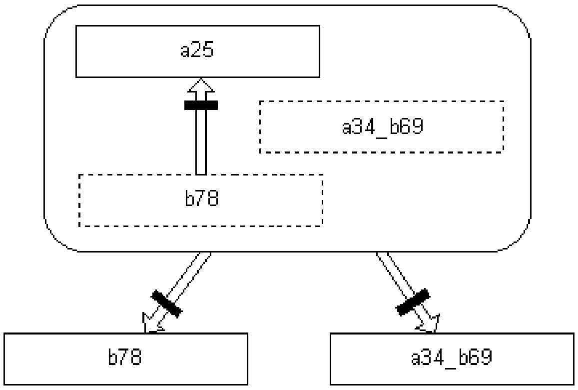


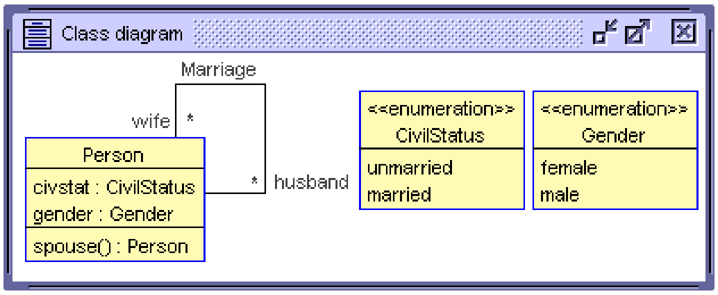
Fig. 5. Invariant Dependence Overview Diagram (CAB-E)

3 Invariant Independence in More Complex Models

Up to now we did not handle associations in test case generation scripts. The example in this section treats associations, and it shows that the fact that non-implications cannot be established is a good indicator that mutual dependencies between invariants do exist.

The example model in Fig. 6 handles the civil status of persons and constrains the attributes and links. The first two constraints are requirements which could also be expressed as multiplicity restrictions in the class diagram. However, we have formulated them explicitly in OCL because we want to check these constraints with respect to independence to the other constraints. The following test case generation script restricts the number of persons to three.

```
procedure genWorld()
var num: Integer,
    persons: Sequence(Person);
begin
num:=[3];
persons:=CreateN(Person,[num]);
Try(Marriage,[persons],[persons]);
for i:Integer in [Sequence{1..num}] begin
```




```
context Person inv atMostOneWife:                    -- aMOW
  wife->size()<=1
context Person inv atMostOneHusband:                 -- aMOH
  husband->size()<=1
context Person inv husbandForFemale:                 -- hFF
  husband->notEmpty() implies gender=#female
context Person inv wifeForMale:                      -- wFM
  wife->notEmpty() implies gender=#male
context Person inv civstatEqualsMarriage:            -- cEM
  (civstat=#married)=
  (husband->notEmpty or wife->notEmpty())
context Person inv spouseDifferentGender:            -- sDG
  gender<>spouse().gender

spouse():Person=                                     -- spouse
  if wife->size()=1 then
    wife->any(true)
  else
    if husband->size()=1 then
      husband->any(true)
    else
      oclUndefined(Person)
    endif
  endif
```

Fig. 6. Example Model CIVSTAT

```
  [persons->at(i)].civstat:=Try([Sequence{#married,#unmarried}]);
  [persons->at(i)].gender:=Try([Sequence{#female,#male}]);
 end;
end;
```

The first Test Configuration and Test Result diagram in Fig. 7 shows independence of three invariants. In each of the six calls to the test case generation script exactly one invariant is negated and the other invariants are affirmed. The test case generation script does not find examples for the three other invariants. The achieved Invariant Dependency Overview diagram is shown in Fig. 8.

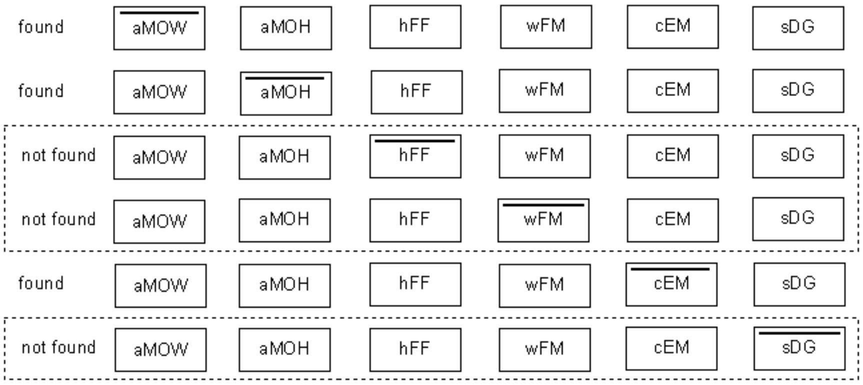


Fig. 7. Test Configuration and Test Result Diagram (CIVSTAT-A)

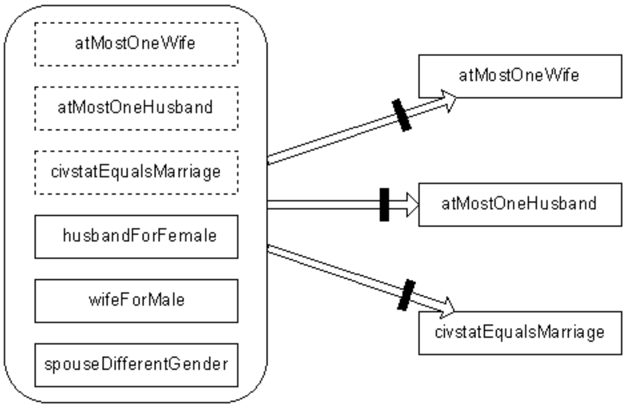


Fig. 8. Invariant Dependence Overview Diagram (CIVSTAT-B)

The second Test Configuration and Test Result diagram in Fig. 9 with deactivations adds six non-implications. These are pictured in the Invariant Dependence Detail diagram in Fig. 10. In contrast to the example from the previous section, the non-implications have a single invariant as source and two invariants as target[2]. Furthermore, the three invariants husbandForFemale, wifeForMale and spouseDifferentGender, whose independence cannot be proved, show symmetrical properties in Fig. 10. This could be seen as an indication for dependencies between these invariants. Indeed, we have `(hFF and wFM) implies sDG`. This could formally be proved with approaches like [BW09] or [BHS07].

This example also shows our general strategy for tackling invariant independence. One should first show as many as possible independencies resp. non-implications.

[2] Now we have `not(A implies B)` as well as `not(A implies C)`, but previously we had `not(B implies A)` as well as `not(C implies A)`.

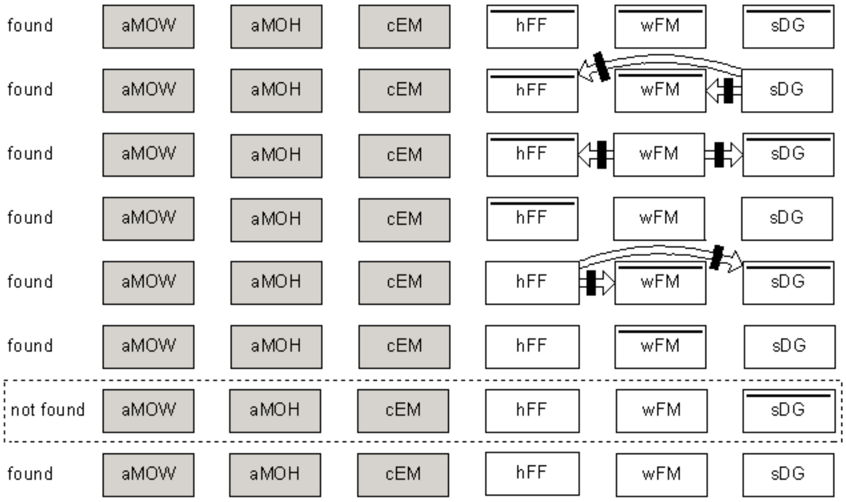


Fig. 9. Test Configuration and Test Result Diagram (CIVSTAT-C)

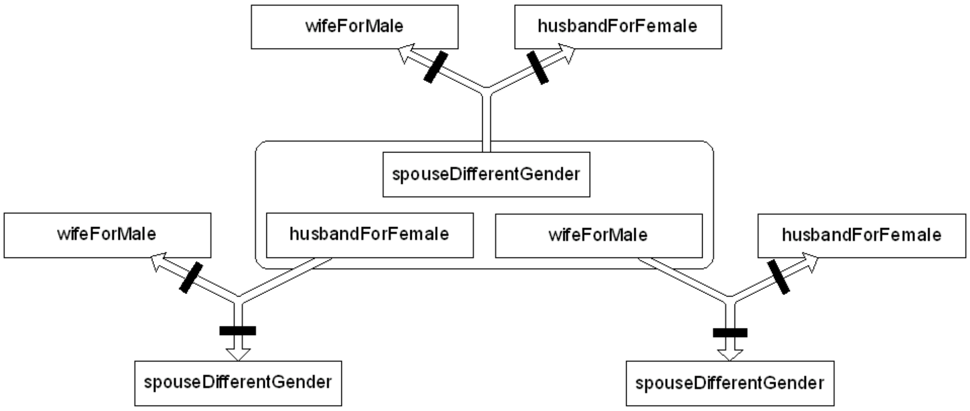


Fig. 10. Invariant Dependence Detail Diagram (CIVSTAT-D)

Invariant dependencies must be in the complement of the shown independencies. Afterwards, the hopefully small complement can be tackled with explicit proof techniques like [BHS07,BW09].

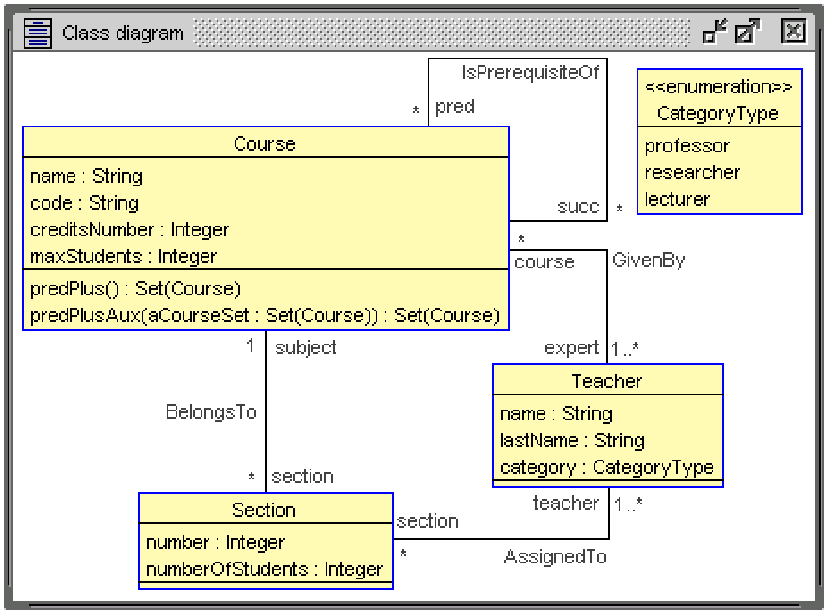
4 Sanity-Check and Test Case Generation Template

The last model is an example which has not been developed by us, but which can be found in the literature [CGQ+08]. The example is a sanity-check that our proposal works for models which are developed not in our group. The specific test case generation script also shows the general structure for such a script. The script has to be written by the model developer for each individual model, but the general structure can be employed for many models and helps to formulate a primary version of the script. If this version is not efficient, it may be taken as the starting point for further improvements, for example, by using restricting OCL expressions in ‘Any’ or ‘Try’ statements which reduce the search space.

The model in Fig. 11 is a bit more complex with three classes, four associations and nine invariants. In the original paper, independence of invariants is discussed under the notion ‘absence of redundancy’ and sufficient criteria for it are stated.

According to the increased complexity of the underlying model, the test case construction script also becomes more involved. Objects, links, and attribute values are generated in that order as follows.

```
procedure genWorld()
var num: Integer, numC: Integer, numT: Integer, numS: Integer,
    courses: Sequence(Course),
    teachers: Sequence(Teacher),
    sections: Sequence(Section);
begin
num:=[2];
numC:=Try([Sequence{1..num}]);
numT:=Try([Sequence{1..num}]);
numS:=Try([Sequence{1..num}]);
courses:=CreateN(Course,[numC]);
teachers:=CreateN(Teacher,[numT]);
sections:=CreateN(Section,[numS]);
Try(IsPrerequisiteOf,[courses],[courses]);
Try(GivenBy,[courses],[teachers]);
Try(BelongsTo,[courses],[sections]);
Try(AssignedTo,[teachers],[sections]);
for i:Integer in [Sequence{1..numC}] begin
  [courses->at(i)].name:=Try([Sequence{'DBS Course','SWE Course'}]);
  [courses->at(i)].code:=Try([Sequence{'dbs','swe'}]);
  [courses->at(i)].maxStudents:=Try([Sequence{40}]);
  [courses->at(i)].creditsNumber:=Try([Sequence{10,12}]);
  end;
for i:Integer in [Sequence{1..numT}] begin
  [teachers->at(i)].name:=Try([Sequence{'Ada','Bob'}]);
  [teachers->at(i)].lastName:=Try([Sequence{'Smith'}]);
  [teachers->at(i)].category:=Try([Sequence{#lecturer,#researcher}]);
  end;
for i:Integer in [Sequence{1..numS}] begin
  [sections->at(i)].number:=Try([Sequence{100,110}]);
  [sections->at(i)].numberOfStudents:=Try([Sequence{30,50}]);
  end;
end;
```




```
context Course inv nameUnique:
  Course.allInstances()->isUnique(name)
context Course inv codeUnique:
  Course.allInstances()->isUnique(code)
context Teacher inv nameLastNameUnique:
  Teacher.allInstances()->isUnique(Sequence{name,lastName})
context Section inv courseNumberUnique:
  Section.allInstances()->isUnique(Sequence{subject,number})
context Course inv isPrerequisiteOfIsAcyclic:
  self.predPlus()->excludes(self)
context Course inv limitOnCredits:
  self.creditsNumber<=10
context Course inv sectionTeacherMustBeExpert:
  self.expert->includesAll(self.section.teacher)
context Section inv limitOnNumberOfStudents:
  self.numberOfStudents<=self.subject.maxStudents
context Course inv atLeastALecturer:
  self.expert.category->includes(#lecturer)

Course::predPlus():Set(Course)=
  self.predPlusAux(self.pred)
Course::predPlusAux(aCourseSet:Set(Course)):Set(Course)=
  let oneStep=aCourseSet.pred->asSet() in
  if oneStep->exists(c|aCourseSet->excludes(c))
    then self.predPlusAux(aCourseSet->union(oneStep))
    else aCourseSet endif
```

Fig. 11. Example Model CST (Course-Section-Teacher)

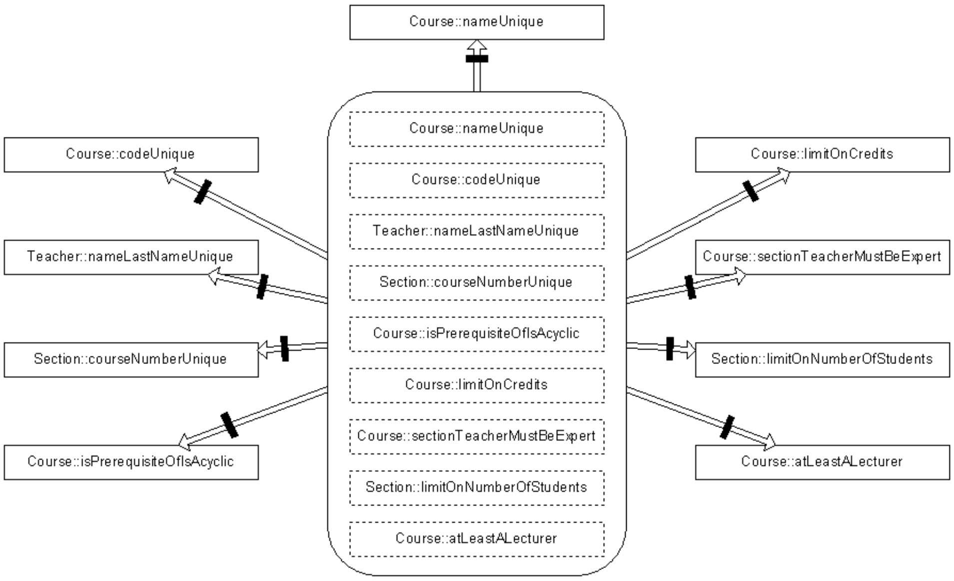


Fig. 12. Invariant Dependence Overview Diagram (CST-A)

According to our approach, we first call the above test case generation script nine times. In each call, exactly one invariant is negated and all other are affirmed. In this case, all invariants are independent and this can be proved already with these first nine calls. The result is represented as an Invariant Dependence Overview diagram in Fig. 12. The found counter examples are shown in Fig. 13 and 14. In our view, these counter examples show the invariant independence, but they also demonstrate nicely how the models work and what the effects of the model look like in term of system states resp. object diagrams.

From the above script we can distill a general format for a test case generation script (Fig. 15). The general format defines a script which works as follows: For each class determine the maximal number of objects in that class and create new objects accordingly; thus the test case generation process will first try to populate the class with one object, then with two objects and so forth; for each association try all possible link sets in that association; for each attribute in each class try all possibilities from an explicitly given sequence of attribute values. In general, one should be conservative with regard to the maximal number of objects. One should start with lowest values and try to increase these numbers with experiments.

Finally, we want again point to the fact that the diagrams which we have presented in this paper are abstractions of the numerous test cases resp. snapshots which were generated in an automatic way. As is detailed in [GHK10], for the

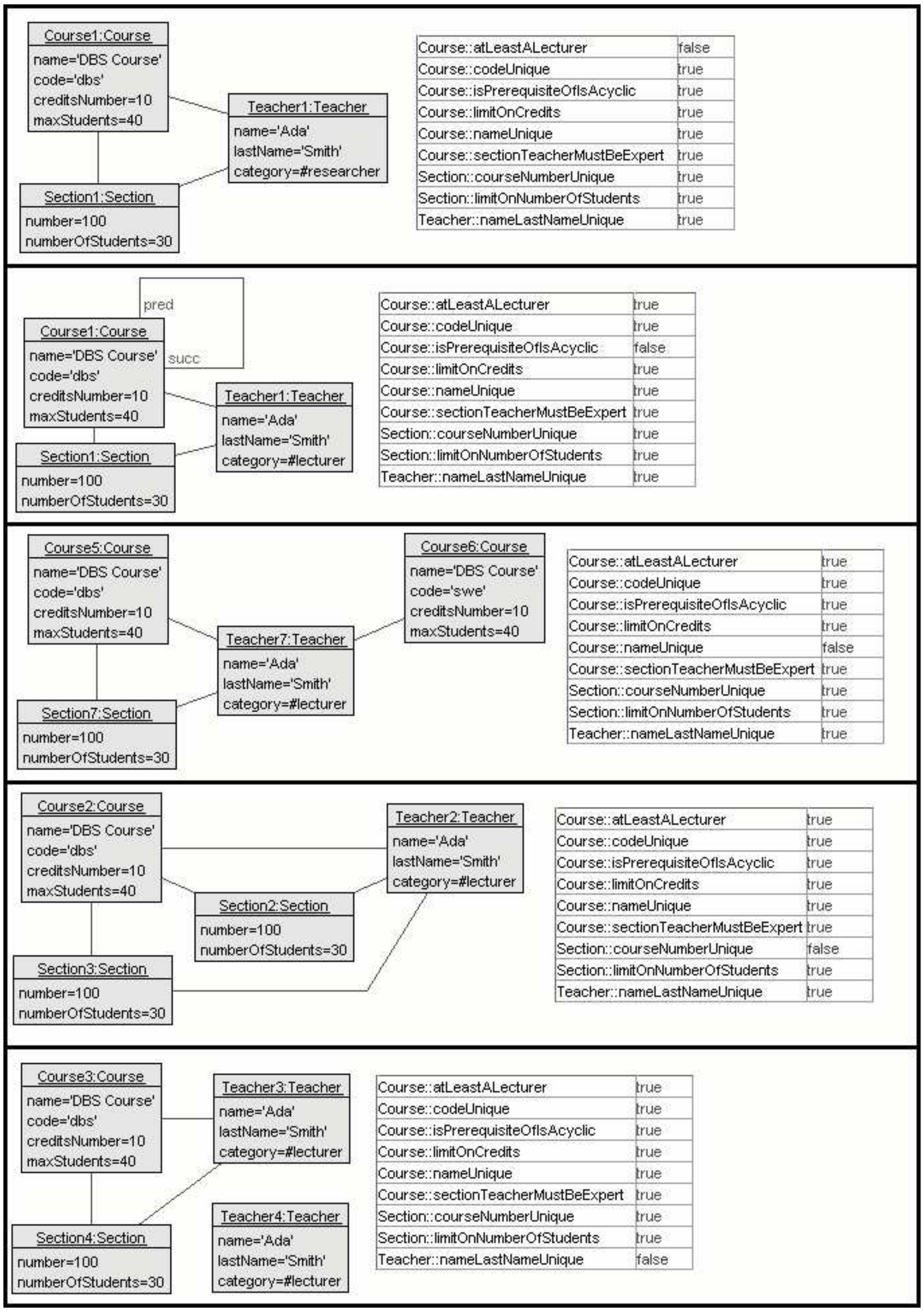


Fig. 13. Counter Examples for Model CST Proving Full Independence (Part 1)

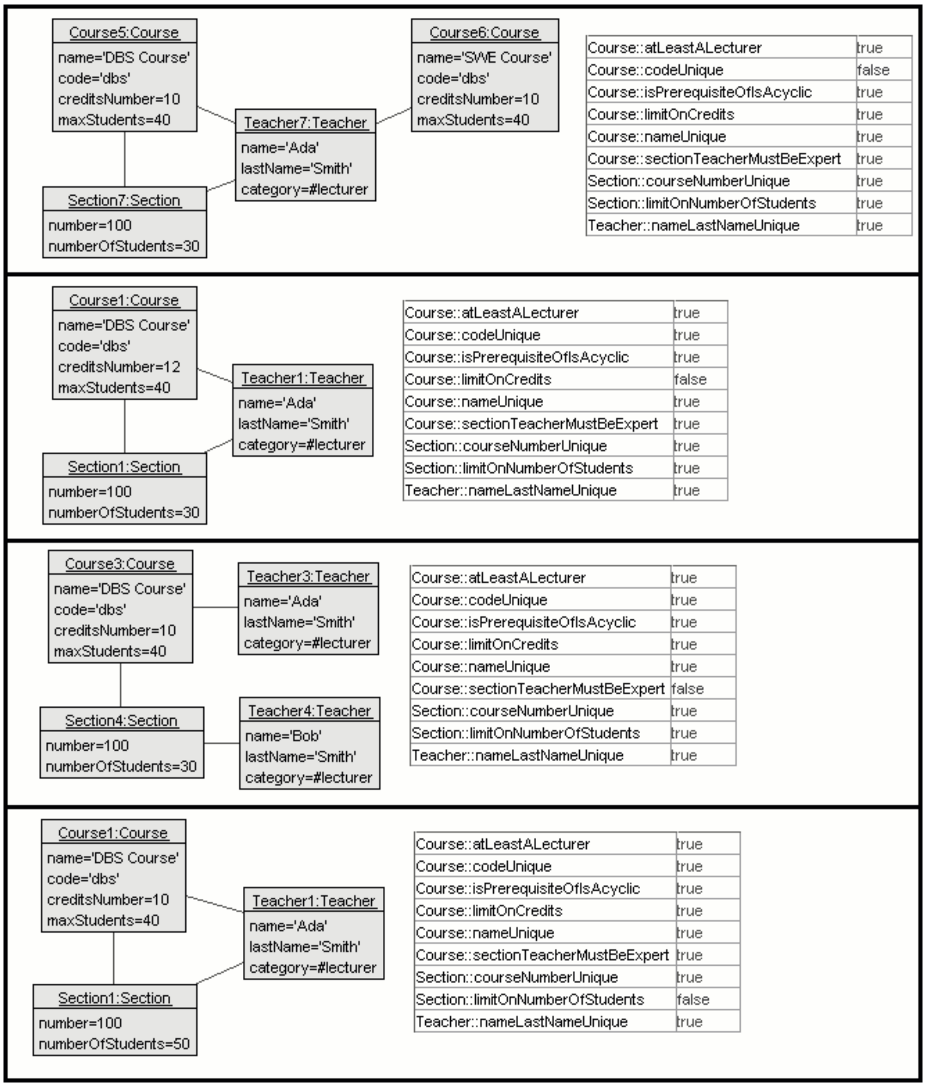


Fig. 14. Counter Examples for Model CST Proving Full Independence (Part 2)

CAB model 250 snapshots, for the CIVSTAT model 101.001 snapshots, and for the CST model 2.393.734 snapshots were considered.[3]

In our examples, the test generation script produces the results within acceptable answer times on a standard laptop. However, for larger examples with more classes SAT-based techniques can produce the counter examples with faster answer times [SWK+10]. As future work, we will combine the proposed visualiza-

[3] This can be traced by considering the messages 'Checked ... snapshots' in [GHK10].

```
procedure genWorld()
var <declaration-of-variables-for-object-sequences>,
    <declaration-of-auxiliary-variables>;
begin
---------------------------------------------------------------- objects
numClass_1:=Try([Sequence{1..numClass_1Max}]);   ...
numClass_k:=Try([Sequence{1..numClass_kMax}]);
objectsClass_1:=CreateN(Class_1,[numClass_1]);   ...
objectsClass_k:=CreateN(Class_k,[numClass_k]);
------------------------------------------------------------------ links
Try(Assoc_1,[objectsClass_i],[objectsClass_j]);   ...
Try(Assoc_m,[     ....     ],[     ....     ]);
------------------------------------------------------- attribute values
for i:Integer in [Sequence{1..numClass_1}] begin
  [objectsClass_1->at(i)].att_1:=Try([Sequence{value_1, ..., value_p}]);
  ...
  [objectsClass_1->at(i)].att_q:=Try([             ....             ]);
  end;
<attribute-handling-for-further-classes>
end;
```

Fig. 15. General Format for Test Case Generation Script

tion techniques of this paper with the more efficient SAT-like counter example production.

5 Related Work

Proving constraint independence has connections to other relevant approaches. Basic concepts for formal testing can be found in the pioneering paper [Gau95]. Independence has been studied also in connection with relational database schemata [Orm97,OS98]. The presented tool in [SCH01] allows static and dynamic model checking with focus on UML statecharts. UMLAnT [TGF$^+$05] allows designers and testers to validate UML models in a way similar to xUnit tests. UMLAnT delegates the validation of invariants, pre- and postconditions to USE [GBR07]. Automatic test generation based on subsets of UML and OCL are examined in [BGL$^+$07] and [AS05], whereas runtime checking of JML assertions transformed from OCL constraints is treated in [AFC08]. Approaches for static UML and OCL model verification without generating a system state can be found in [MKL02], [BC06] and [BHS07]. In [MKL02], UML models and corresponding OCL expressions are translated into B specifications. The transformation result is analyzed by an external tool. The work in [BC06] focuses on static verification of dynamic model parts (sequence diagrams) with respect to the static specification. The KeY approach [BHS07] 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 [Jac06] approach. As our approach, the ALLOY approach detects model properties by setting up a search space for model snapshots and exhaustively stepping through this search space [SYC+04]. The achieved results are always relative to this search space. Thus exhaustive testing techniques on the code level [JLDM09] with methods like sparse test generation, structural test merging or oracle-based test clustering may contribute to the improvement of our approach.

6 Conclusion

We have proposed an approach for detecting independencies among class invariants. We were able to trace more independence facts by using invariant deactivation. The paper discussed the representation and documentation of test case generation with diagrams offering different degrees of detail. A general template for a test case generation script has been put forward.

Future work will extend our approach for handling OCL invariants to handling OCL pre- and postconditions. Special interest must be paid to the interplay between invariants and pre- and postconditions. One can improve the efficiency of our proposal with SAT-based techniques. An implementation of the proposed diagrams and case studies giving user-feedback have to be carried out. In general, we are interested to study further relationships between proving and testing in connection with UML and OCL models.

References

[AFC08] Carmen Avila, Guillermo Flores, and Yoonsik Cheon. A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 403–408. CSREA Press, 2008.

[AS05] Bernhard K. Aichernig and Percy Antonio Pari Salas. Test Case Generation by OCL Mutation and Constraint Solving. In *QSIC*, pages 64–71. IEEE Computer Society, 2005.

[BC06] Andrea Baruzzo and Marco Comini. Static Verification of UML Model Consistency. In David Hearnden, Jörn Guy Süß, Benoît Baudry, and Nicolas Rapin, editors, *Proc. 3rd Workshop on Model Design and Validation*, pages 111–126. University of Queensland, 2006.

[BGL+07] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, Fabien Peureux, Nicolas Vacelet, and Mark Utting. A Subset of Precise UML for Model-Based Testing. In *A-MOST*, pages 95–104. ACM, 2007.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.

[BW09] Achim D. Brucker and Burkhart Wolff. Semantics, Calculi, and Analysis for Object-Oriented Specifications. *Acta Informatica*, 46(4):255–284, 2009.

[CGQ+08] Dolors Costal, Cristina Gómez, Anna Queralt, Ruth Raventós, and Ernest Teniente. Improving the Definition of General Constraints in UML. *Journal of Software and System Modeling*, 7(4):469–486, 2008.

[Gau95] Marie-Claude Gaudel. Testing can be Formal, too. In *Proc. 6th Int. Conf. Theory and Practice of Software Development (TAPSOFT '95)*, pages 82–96. Springer, 1995.

[GBR05] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Journal on Software and System Modeling*, 4(4):386–398, 2005.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.

[GHK10] Martin Gogolla, Lars Hamann, and Mirco Kuhlmann. Detailed Test Cases for 'Proving OCL Invariant Independence by Automatically Generated Test Cases'. http://www.db.informatik.uni-bremen.de/publications/intern/GHK2010ADDON.pdf, 2010.

[GKH09] Martin Gogolla, Mirco Kuhlmann, and Lars Hamann. Consistency, Independence and Consequences in UML and OCL Models. In Catherine Dubois, editor, *Proc. 3rd Int. Conf. Test and Proof (TAP'2009)*, pages 90–104. Springer, Berlin, LNCS 5668, 2009.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[JLDM09] Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov. Reducing the costs of bounded-exhaustive testing. In Marsha Chechik and Martin Wirsing, editors, *FASE*, pages 171–185. Springer, LNCS 5503, 2009.

[MKL02] Rafael Marcano-Kamenoff and Nicole Lévy. Transformation Rules of OCL Constraints into B Formal Expressions. In *Workshop Critical Systems Development (CSDUML'2002) with UML at 5th Int. Conf. Unified Modeling Language*, 2002.

[Orm97] Levent V. Orman. Relational Database Constraints as Counterexamples. *Acta Informatica*, 34(3):167–189, 1997.

[OS98] Hussien Oakasha and Gunter Saake. Foundations for Integrity Independence in Relational Databases. In *FMLDO*, pages 143–165, 1998.

[SCH01] Wuwei Shen, Kevin Compton, and James Huggins. A Toolset for Supporting UML Static and Dynamic Model Checking. In *Proc. 16th IEEE Int. Conf. Automated Software Engineering (ASE)*, pages 147–152. IEEE, 2001.

[SWK+10] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL Models Using Boolean Satisfiability. In WolfgangMüller, editor, *Proc. Design, Automation and Test in Europe (DATE'2010)*. IEEE, 2010.

[SYC+04] Kevin J. Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. In George S. Avrunin and Gregg Rothermel, editors, *ISSTA*, pages 133–142. ACM, 2004.

[TGF+05] Trung Dinh Trong, Sudipto Ghosh, Robert B. France, Michael Hamilton, and Brent Wilkins. UMLAnT: An Eclipse Plugin for Animating and Testing UML Designs. In *Proc. OOPSLA Workshop on Eclipse Technology Exchange (Eclipse'05)*, pages 120–124. ACM Press, 2005.