Evaluating and Debugging OCL Expressions in UML Models

Jens Brüning[1], Martin Gogolla[2], Lars Hamann[2], and Mirco Kuhlmann[2]

[1] University of Rostock
[2] University of Bremen

Abstract. This paper discusses the relationship between tests and proofs with focus on a tool for UML and OCL models. Tests are thought of as UML object diagrams and theorems or properties which are to be checked are represented as OCL constraints, i.e., class invariants or operation pre- and postconditions. The paper shows for the UML and OCL tool USE (UML-based Specification Environment) how to trace and debug the validity of an expected theorem (an OCL constraint) within a given test case (a state model in the form of a UML object diagram).

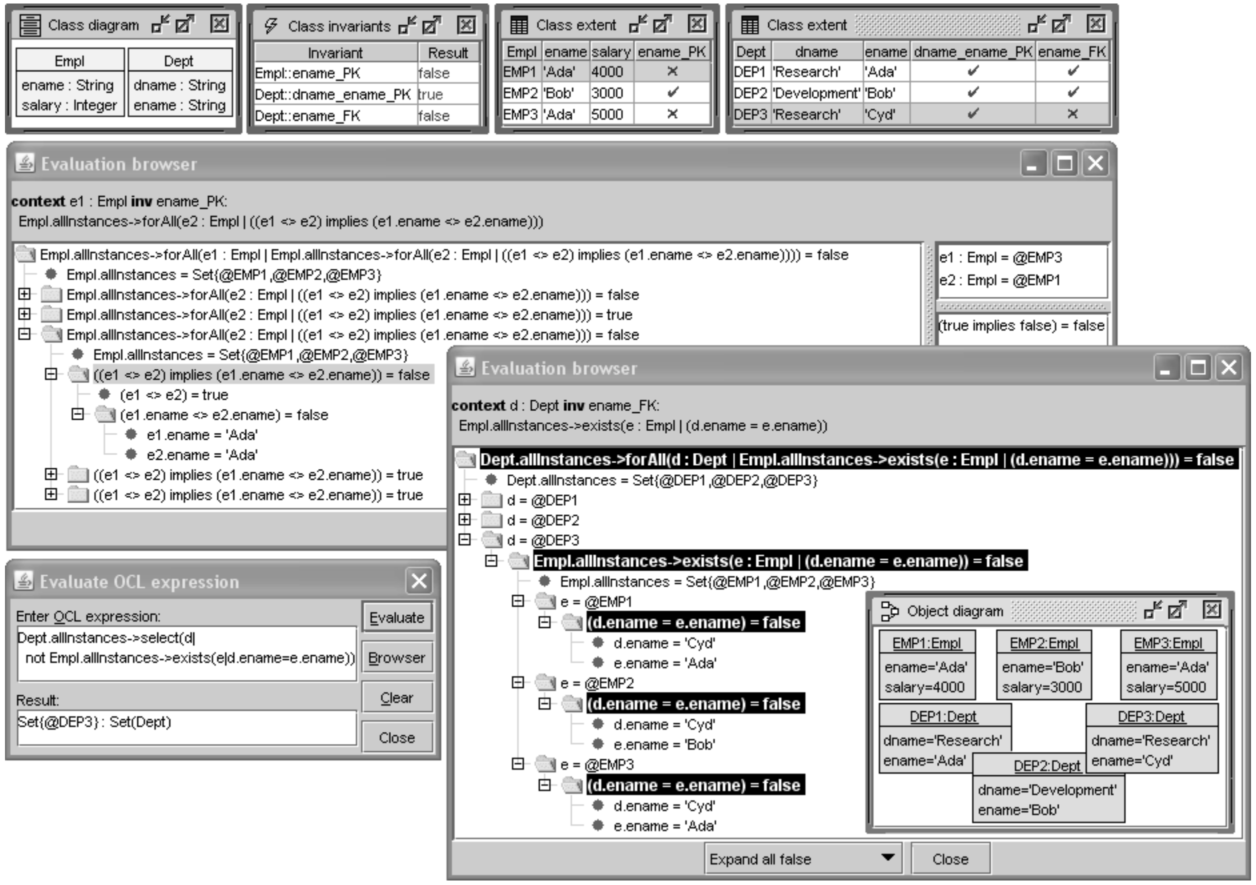
1 Introduction

A central issue in the relationship between tests and proofs is the question which part of a test affects which part of a proof or a theorem to be proven. Tests as well as proofs and the underlying theorems are highly structured entities with many important relationships, not all being relevant in a specific situation during development. For example, for proof counter-examples it is important to know which part of the expected proof or theorem is falsified by the counter-example, and it is important for the developer to find the respective parts of the test and the proof or theorem in an adequate way.

This paper discusses this general question with focus on a tool for UML and OCL models. Tests are thought of as UML object diagrams and theorems or properties which are to be checked are represented as OCL constraints, i.e., class invariants or operation pre- and postconditions. The paper shows for the UML and OCL tool USE (UML-based Specification Environment) [GKH09] how to trace and debug the validity of an expected theorem (an OCL constraint) within a given test case (a constructed state model in form of a UML object diagram). The technical realization in the tool is done by a so-called evaluation browser which allows the developer to debug the evaluation of a complex OCL expression and its subexpressions with respect to a given system state in a user-friendly way with the aim of better understanding, for example, invariant failure.

2 Basic Evaluation Browser Concepts by Example

Let us introduce the basic idea of our approach by means of an example. The USE screenshot in Fig. 1 shows in the upper row an OCL and UML model with



Class diagram

Empl	Dept
ename : String salary : Integer	dname : String ename : String

Class invariants

Invariant	Result
Empl::ename_PK	false
Dept::dname_ename_PK	true
Dept::ename_FK	false

Class extent

Empl	ename	salary	ename_PK
EMP1	'Ada'	4000	✗
EMP2	'Bob'	3000	✓
EMP3	'Ada'	5000	✗

Class extent

Dept	dname	ename	dname_ename_PK	ename_FK
DEP1	'Research'	'Ada'	✓	✓
DEP2	'Development'	'Bob'	✓	✓
DEP3	'Research'	'Cyd'	✓	✗

Evaluation browser

context e1 : Empl **inv** ename_PK:
Empl.allInstances->forAll(e2 : Empl | ((e1 <> e2) implies (e1.ename <> e2.ename)))

```
Empl.allInstances->forAll(e1 : Empl | Empl.allInstances->forAll(e2 : Empl | ((e1 <> e2) implies (e1.ename <> e2.ename)))) = false
  Empl.allInstances = Set{@EMP1,@EMP2,@EMP3}
  Empl.allInstances->forAll(e2 : Empl | ((e1 <> e2) implies (e1.ename <> e2.ename))) = false
  Empl.allInstances->forAll(e2 : Empl | ((e1 <> e2) implies (e1.ename <> e2.ename))) = true
  Empl.allInstances->forAll(e2 : Empl | ((e1 <> e2) implies (e1.ename <> e2.ename))) = false
    Empl.allInstances = Set{@EMP1,@EMP2,@EMP3}
    ((e1 <> e2) implies (e1.ename <> e2.ename)) = false
      (e1 <> e2) = true
      (e1.ename <> e2.ename) = false
        e1.ename = 'Ada'
        e2.ename = 'Ada'
    ((e1 <> e2) implies (e1.ename <> e2.ename)) = true
    ((e1 <> e2) implies (e1.ename <> e2.ename)) = true
```

```
e1 : Empl = @EMP3
e2 : Empl = @EMP1
```

```
(true implies false) = false
```

Evaluate OCL expression

Enter OCL expression:

```
Dept.allInstances->select(d|
 not Empl.allInstances->exists(e|d.ename=e.ename))
```

Result:

```
Set{@DEP3} : Set(Dept)
```

Evaluation browser

context d : Dept **inv** ename_FK:
Empl.allInstances->exists(e : Empl | (d.ename = e.ename))

```
Dept.allInstances->forAll(d : Dept | Empl.allInstances->exists(e : Empl | (d.ename = e.ename))) = false
  Dept.allInstances = Set{@DEP1,@DEP2,@DEP3}
  d = @DEP1
  d = @DEP2
  d = @DEP3
    Empl.allInstances->exists(e : Empl | (d.ename = e.ename)) = false
      Empl.allInstances = Set{@EMP1,@EMP2,@EMP3}
      e = @EMP1
        (d.ename = e.ename) = false
          d.ename = 'Cyd'
          e.ename = 'Ada'
      e = @EMP2
        (d.ename = e.ename) = false
          d.ename = 'Cyd'
          e.ename = 'Bob'
      e = @EMP3
        (d.ename = e.ename) = false
          d.ename = 'Cyd'
          e.ename = 'Ada'
```

Object diagram

```
EMP1:Empl          EMP2:Empl          EMP3:Empl
ename='Ada'        ename='Bob'        ename='Ada'
salary=4000        salary=3000        salary=5000

DEP1:Dept          DEP2:Dept               DEP3:Dept
dname='Research'   dname='Development'     dname='Research'
ename='Ada'        ename='Bob'             ename='Cyd'
```

Fig. 1. Basic Use of Evaluation Browser

invariants and a corresponding state in form of 2 Class extent windows (these 2 windows determine a UML object diagram displayed in the lower right). The class diagram represents a simple relational database schema with two tables (`Empl[oyee]`, `Dep[artmen]t`), two primary key constraints ({`ename`} is primary key in `Empl`, {`dname, ename`} is primary key in `Dept`) and one foreign key constraint (`Dept.ename` references `Empl.ename`). The OCL details of one primary key constraint and the foreign key constraint are shown in the top part of the 2 Evaluation browser windows, respectively.

As shown in the Class invariants window, the (database) state represented in the Class extent windows does violate 2 of the specified constraints. In order to understand the reason for the violation, USE allows to open so-called Evaluation browser windows. In the screenshot we see one Evaluation browser window for the failing primary key constraint `ename_PK` and one for the failing foreign key constraint `ename_FK`. The windows have been configured in different ways to demonstrate the possibilities of our approach. For example, the first window shows variable substitutions in a subwindow in the very right, the second window shows the variable substitutions inside the main Evaluation browser window. OCL expressions evaluating to `false` are highlighted in the second window in a white-on-black style, whereas they are displayed without special indication in the first window. In this simple situation, the analysis could be done by simple inspection without the evaluation browser, but we want to demonstrate the approach with an easy understandable example. We will show below an involved situation hard to understand by simple inspection.

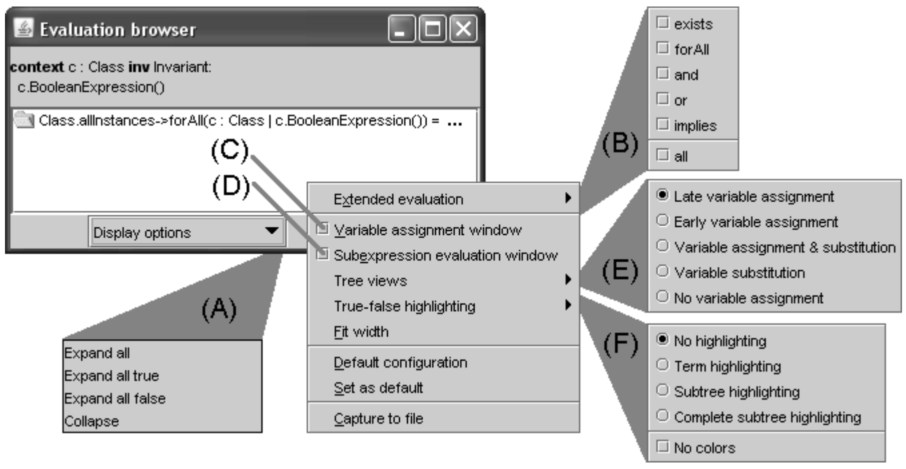
In the first Evaluation browser window, which can be opened by double-clicking in the Class invariants window the failing primary key invariant `ename_PK`, one subformula which evaluates to `false` is highlighted in grey. Three lines below the highlighted grey line, we see that the OCL terms `e1.ename` and `e2.ename` both evaluate to `'Ada'`. The variable substitutions responsible for this evaluation are stated in the right subwindow (basically stating `e1=EMP3, e2=EMP1`) and the evaluation of the selected and highlighted subexpression is displayed below the substitutions on the right (`(true implies false)=false`). Thus, this Evaluation browser window displays one concrete counter-proof for the expected primary key property `ename_PK`: `EMP3` and `EMP1` are distinct, but their ename values coincide, and this violates the primary key requirement.

In the second Evaluation browser window, the foreign key constraint is analyzed. Only those subformulas evaluating to `false` are displayed and are pictured in a white-on-black style. Below the central highlighted exists subformula it is shown that for the `Dept` object `DEP3` having ename attribute value `'Cyd'` no corresponding `Empl` object exists having the ename attribute value `'Cyd'`. Thus, the Evaluation browser window again displays one concrete counter-proof for the expected foreign key property `ename_FK`.

3 General Features Available in the Evaluation Browser

Our so-called evaluation browser pictures the evaluation of an OCL term in a graphical style as a tree. The tree nodes show OCL terms or subterms of the original term together with values of subterms and substitutions for occurring variables. Tree subbranches may be opened or closed interactively through the user or by setting particular configuration parameters. Particular tree parts may be highlighted in color or in a white-on-black style. The aim of the evaluation browser is to offer an intuitive, highly configurable, and flexible tool for analyzing the evaluation of complex OCL terms. As indicated in Fig. 2, there are basically six central configuration parameters for the OCL evaluation browser (basically available by right-clicking into the Evaluation browser's pane):

(A) Determination of opened subbranches of the tree.
(B) Turning the extended OCL formula evaluation on or off.
(C) Turning an additional variable assignment subwindow on or off.
(D) Turning an additional subexpression evaluation subwindow on or off.
(E) Positioning of variable assignments in the main evaluation term.
(F) Determining the highlighting of subformulas evaluating to particular values.



```
context c : Class inv Invariant:
 c.BooleanExpression()
```

```
Class.allInstances->forAll(c : Class | c.BooleanExpression()) = ...
```

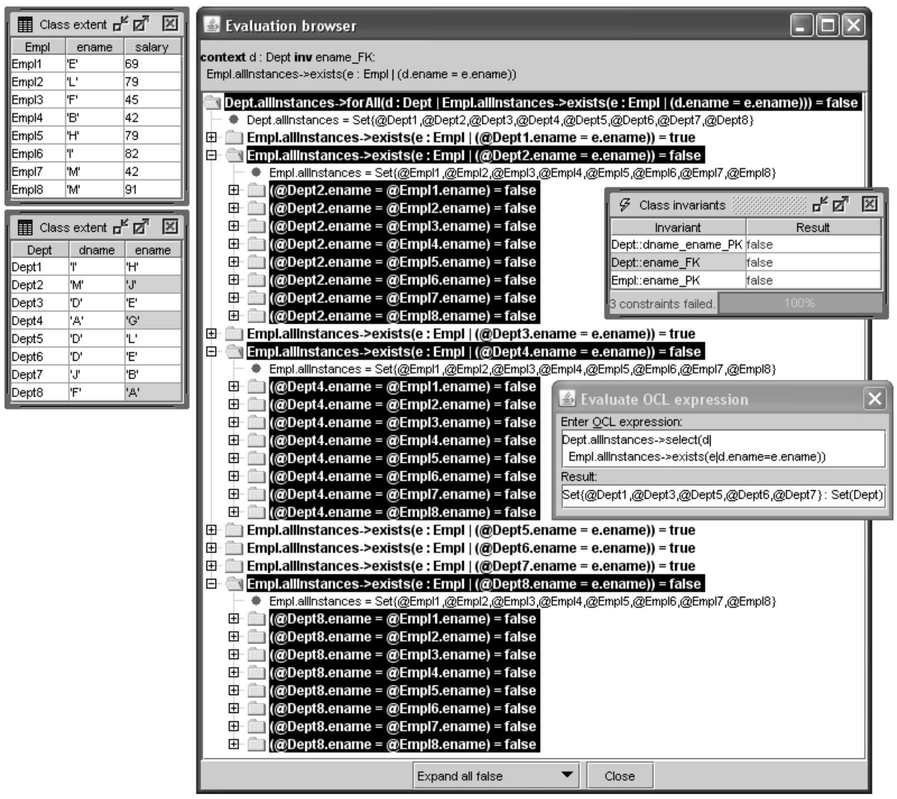
Fig. 2. General Features Available in Evaluation Browser

In (A) the developer determines the basic structure of opened or closed tree subbranches. Either all subbranches, all subbranches evaluating to `true`, all subbranches evaluating to FALSE are opened or no subbranch is opened. In (B) an extended evaluation of OCL subformulas is configured. In the standard evaluation of OCL for the exists quantifier the evaluation stops with `true`, if the first satisfying element is found. However, one frequently wants to know all elements satisfying the exists predicate. This can be accomplished by turning on the extended evaluation for exists. Analogous possibilities are provided for the other logical operations. In (C) an explicit subwindow for the variable assignments is opened. In (D) an explicit subwindow for the subexpression evaluation is opened. In (E) the position of variable assignments in the tree is fixed.

The variable assignments may be placed at the tree leafs ('Late') or inside the tree as early as they appear ('Early'). More options are available. In (F) the highlighting of `false` subformulas resp. `true` subformulas is determined.

4 Further Features Available in the Evaluation Browser

The second USE screenshot in Fig. 3 shows the evaluation browser being used for an automatically generated object diagram (test case) which is the result of an ASSL procedure [GKH09]. The randomly generated object diagram represented by 2 Class extent windows in the left upper part of the screenshot involves 16 objects with respective attribute values. As the class invariants window in the right shows, all invariants fail. The Evaluation browser window has been opened through double-clicking the failing invariant `Dept::ename_FK`. This window shows all details, i.e., all reasons, why this invariant fails. The complete evaluation tree has 3 subbranches evaluating to `false` and exactly these 3 subbranches have been opened and are displayed as white-on-black. The 3 subbranches indicate that the 3 objects `Dept2`, `Dept4`, and `Dept8` are the reason for



Empl	ename	salary
Empl1	'E'	69
Empl2	'L'	79
Empl3	'F'	45
Empl4	'B'	42
Empl5	'H'	79
Empl6	'I'	82
Empl7	'M'	42
Empl8	'M'	91

Dept	dname	ename
Dept1	'I'	'H'
Dept2	'M'	'J'
Dept3	'D'	'E'
Dept4	'A'	'G'
Dept5	'D'	'L'
Dept6	'D'	'E'
Dept7	'J'	'B'
Dept8	'F'	'A'

```
context d : Dept inv ename_FK:
 Empl.allInstances->exists(e : Empl | (d.ename = e.ename))

Dept.allInstances->forAll(d : Dept | Empl.allInstances->exists(e : Empl | (d.ename = e.ename))) = false
  Dept.allInstances = Set{@Dept1,@Dept2,@Dept3,@Dept4,@Dept5,@Dept6,@Dept7,@Dept8}
  Empl.allInstances->exists(e : Empl | (@Dept1.ename = e.ename)) = true
  Empl.allInstances->exists(e : Empl | (@Dept2.ename = e.ename)) = false
    Empl.allInstances = Set{@Empl1,@Empl2,@Empl3,@Empl4,@Empl5,@Empl6,@Empl7,@Empl8}
    (@Dept2.ename = @Empl1.ename) = false
    (@Dept2.ename = @Empl2.ename) = false
    (@Dept2.ename = @Empl3.ename) = false
    (@Dept2.ename = @Empl4.ename) = false
    (@Dept2.ename = @Empl5.ename) = false
    (@Dept2.ename = @Empl6.ename) = false
    (@Dept2.ename = @Empl7.ename) = false
    (@Dept2.ename = @Empl8.ename) = false
  Empl.allInstances->exists(e : Empl | (@Dept3.ename = e.ename)) = true
  Empl.allInstances->exists(e : Empl | (@Dept4.ename = e.ename)) = false
    Empl.allInstances = Set{@Empl1,@Empl2,@Empl3,@Empl4,@Empl5,@Empl6,@Empl7,@Empl8}
    (@Dept4.ename = @Empl1.ename) = false
    (@Dept4.ename = @Empl2.ename) = false
    (@Dept4.ename = @Empl3.ename) = false
    (@Dept4.ename = @Empl4.ename) = false
    (@Dept4.ename = @Empl5.ename) = false
    (@Dept4.ename = @Empl6.ename) = false
    (@Dept4.ename = @Empl7.ename) = false
    (@Dept4.ename = @Empl8.ename) = false
  Empl.allInstances->exists(e : Empl | (@Dept5.ename = e.ename)) = true
  Empl.allInstances->exists(e : Empl | (@Dept6.ename = e.ename)) = true
  Empl.allInstances->exists(e : Empl | (@Dept7.ename = e.ename)) = true
  Empl.allInstances->exists(e : Empl | (@Dept8.ename = e.ename)) = false
    Empl.allInstances = Set{@Empl1,@Empl2,@Empl3,@Empl4,@Empl5,@Empl6,@Empl7,@Empl8}
    (@Dept8.ename = @Empl1.ename) = false
    (@Dept8.ename = @Empl2.ename) = false
    (@Dept8.ename = @Empl3.ename) = false
    (@Dept8.ename = @Empl4.ename) = false
    (@Dept8.ename = @Empl5.ename) = false
    (@Dept8.ename = @Empl6.ename) = false
    (@Dept8.ename = @Empl7.ename) = false
    (@Dept8.ename = @Empl8.ename) = false
```

Invariant	Result
Dept::dname_ename_PK	false
Dept::ename_FK	false
Empl::ename_PK	false

3 constraints failed. 100%

Enter OCL expression:

```
Dept.allInstances->select(d|
 Empl.allInstances->exists(e|d.ename=e.ename))
```

Result:

```
Set{@Dept1,@Dept3,@Dept5,@Dept6,@Dept7} : Set(Dept)
```

Fig. 3. Further Use of Evaluation Browser

the invariant failure. Checking these objects against all Dept objects in the second Class extent window one learns that the respective ename values (`'J', 'G', 'A'`) indeed cannot be found as ename values in the first Class extent window for Empl objects. In this evaluation browser configuration, variable substitutions are not displayed, but variables have been substituted by their values.

The Evaluate OCL expression window on the right is a cross-check against the found result. This OCL expression retrieves all Dept objects which possess a corresponding Empl object having the same name. It returns the complement `Dept` object set `Dept1`, `Dept3`, `Dept5`, `Dept6`, and `Dept7`. This screenshot is an explanation why the 'theorem' (i.e., the invariant `Dept::ename_FK`) fails in this test case (i.e., in this object diagram). Such a detailed analysis is needed during development when unexpected results in form of failing constraints occur in order to understand the reason for constraint failure.

5 Related Work

We only point to a few works on debugging in the context of model-based development and declarative languages. In [SSJ$^+$03] counter-example generation is understood as debugging. [GHMGB07] discusses model-level debugging for software architectures. Initial ideas towards model-based debugging are proposed in [MS08]. [KSWR09] discusses a connection between debugging and QVT. The work in [RVM10] proposes a debugger for the specification language Maude.

6 Conclusion

This paper has made a proposal for debugging OCL invariants in UML models. Debugging works by means of term evaluation. The display of the evaluation term may be adjusted by the developer in various ways. The ideas of the proposal could be used for other declarative languages as well. Up to now there are too few proposals for user-friendly debugging in the context of theorem proving or theorem checking. Our approach currently works for invariants only and has to be extended for pre- and postconditions. Further options for configuring the evaluation tree are imaginable, for example, grouping of subbranches with similar results. Larger case studies must give feedback on the usability of the proposal.

References

[GHMGB07] Graf, P., Hübner, M., Müller-Glaser, K.D., Becker, J.: A Graphical Model-Level Debugger for Heterogenous Reconfigurable Architectures. In: Bertels, K., Najjar, W.A., van Genderen, A.J., Vassiliadis, S. (eds.) FPL, pp. 722–725. IEEE (2007)

[GKH09] Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In: Dubois, C. (ed.) TAP 2009. LNCS, vol. 5668, pp. 90–104. Springer, Heidelberg (2009)

[KSWR09] Kusel, A., Schwinger, W., Wimmer, M., Retschitzegger, W.: Common Pitfalls of using QVT Relations - Graphical Debugging as Remedy. In: ICECCS, pp. 329–334. IEEE Computer Society (2009)

[MS08] Mayer, W., Stumptner, M.: Evaluating Models for Model-Based Debugging. In: ASE, pp. 128–137. IEEE (2008)

[RVM10] Riesco, A., Verdejo, A., Martí-Oliet, N.: A Complete Declarative Debugger for Maude. In: Johnson, M., Pavlovic, D. (eds.) AMAST 2010. LNCS, vol. 6486, pp. 216–225. Springer, Heidelberg (2011)

[SSJ$^+$03] Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., Taghdiri, M.: Debugging Overconstrained Declarative Models using Unsatisfiable Cores. In: ASE, pp. 94–105. IEEE Computer Society (2003)