Extensive Validation of OCL Models by Integrating SAT Solving into USE

Mirco Kuhlmann, Lars Hamann, and Martin Gogolla

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

Abstract. The Object Constraint Language (OCL) substantially enriches modeling languages like UML, MOF or EMF with respect to formulating meaningful model properties. In model-centric approaches, an accurately defined model is a requisite for further use. During development of a model, continuous validation of properties and feedback to developers is required, since many design flaws can then be directly discovered and corrected. For this purpose, lightweight validation approaches which allow developers to perform automatic model analysis are particularly helpful. We provide a new method for efficiently searching for model instances. The existence or non-existence of model instances with certain properties allows significant conclusions about model properties. Our approach is based on the translation of UML and OCL concepts into relational logic and its realization with SAT solvers. We explain various use cases of our proposal, for example, completion of partly defined model instances so that particular properties hold in the completed model instances. Our proposal is realized by integrating a model validator as a plugin into the UML and OCL tool USE.

1 Introduction

Unlike traditional code-centric approaches, model-centric and model-driven software and hardware development relies on comprehensive and in particular flawless models. Modeling languages like UML, MOF or EMF are substantially enriched by the Object Constraint Language (OCL) which allows precise definitions of model properties. However, complex models increase the need for extensive validation and verification. Our work focuses on discovering implied model qualities and properties early in the development cycle, i. e., properties which follow from the given UML and OCL model constraints. During the development, continuous validation of properties and feedback to the developers is required, since many design flaws can then be directly discovered and corrected. Lightweight validation approaches which allow developers to perform automatic model analysis are thus particularly helpful (in contrast to strongly interactive theorem proving approaches). In our work, we automatically confirm wanted and find unwanted model properties (features and bugs) by examining the existence or non-existence of specific model instances, i. e., snapshots representing concrete states of the modeled system.

For this purpose, we developed a method which we call OCL2Kodkod. By translating UML and OCL concepts into the relational logic of Kodkod [23],

our method enables an efficient SAT-based search and generation of partial and complete snapshots with user-specified properties. The user has not to deal with any details of relational logic, but employs UML and OCL concepts only. We integrated OCL2Kodkod as a plugin into our UML and OCL tool USE (UML-based Specification Environment) [10] adding various new validation options.

In order to realize a modular architecture, the USE system was recently modified in order to support a flexible incorporation of new functionality via plugins. The plugin architecture is well-suited for research and teaching projects like student or PhD theses. The first plugin which realizes the method OCL2Kodkod is called 'model validator'.

The rest of the paper is structured as follows. Sect. 2 introduces the tools and methods handled within the paper. In Sect. 3 we present the principles of the USE plugin architecture and the particularities of the model validator plugin. Section 4 discusses the principles of model validation wrt. the model validator and explains the principles in the context of a concrete UML and OCL model. Section 5 gives an overview of the translation from UML and OCL to Kodkod. Related work is presented in Sect. 6. We conclude with Sect. 7.

2 Background: The Tools USE and Kodkod

In order to assist developers in model-driven techniques, our group puts forward the tool USE [10]. USE is basically an interpreter for a subset of UML and MOF and OCL. The main task of USE is to validate and verify specifications consisting of class diagrams together with OCL invariants and pre- and postconditions. USE further supports object diagrams and sequence diagrams. Commands for atomic snapshot manipulation (object and link creation/deletion and attribute modification) allow to construct states. Also part of the USE system is a so-called snapshot generator based on 'a snapshot sequence language' (ASSL) [9]. We distinguish the built-in snapshot generator from the new model validator plugin in Sect. 4.

Kodkod [23] implements relational logic [13] which is purely based on relations, that is, sets of tuples. Independent from their nature, all values are uniformly handled as relations. Atomic values like integer values are represented by unary singleton relations (e. g., $[(2)]$). Sets of atomic values are relations with possibly more than one unary tuple (e. g., $[(a),(b),(c)]$). Relationships between atoms are specified with n-ary relations (e. g., a function $[(a,2),(b,8),(c,2)]$). The structure of a relation can be freely defined. Kodkod's logic provides a large set of operations for handling relations, including common set operations like union and subset. The most important means for extracting information and merging different relations is the so called 'join' expressed by a dot (e. g., $[(a,2),(b,3)].[(2,c)] = [(a,c)]$). Often, the join emulates a function call (e. g., $[(a)].[(a,2),(b,8),(c,2)] = [(2)]$). In order to perform calculations based on integer values, the relational logic additionally includes the corresponding operations. Naturally, the relational logic brings boolean operations for building relational formulas which are evaluated in the context of concrete relation instances.

Kodkod is designed as an API allowing developers of other tools to easily use its features. Its most important feature is the possibility to efficiently find specific

instantiations of the underlying relations which fulfill the specified relational formulas. This task is achieved by translating the formulas from the relational logic into boolean logic. The resulting boolean formulas can in turn be explored by proprietary solvers for boolean satisfiability (SAT). If the applied SAT solver finds a solution, the solution is translated back into respective relation instances.

3 Integrating SAT Solving into the USE Tool

The USE tool provides a way to add new features to the USE system without altering the internal USE sources. In the following subsection we dwell on the principles of this plugin architecture. After that, we consider the new 'model validator' plugin and its features.

3.1 The USE Plugin Architecture

Since the first release of USE, several extensions to the system were made. Some of these extensions were the result of diploma theses supervised in our group. One example of such an extension is the snapshot generator mentioned before. When unexperienced developers are involved — which is the common and natural case with students — it is hard to maintain several concurrent feature branches. Students are normally not interested in the overall USE development road map, nor do they care about system changes done during their diploma work. This makes the (re)integration of their results a labor-intensive task. Therefore, we enriched USE to support extensions to the system by plugins which allows us to decouple the various extensions from the system core maintained by our group. Also the users of our tool will benefit from the plugin architecture, once a greater variety of USE plugins well be available. Users can then customize USE to their needs focussing the functionality required to solve their specific tasks.

Several commonly used applications support extensions by plugins, AddOns or AddIns, respectively. Eclipse may be seen as an application in which nearly every feature is realized by a plugin whereas modern browsers provide more core functionality but allow the user to customize and extend them via plugins. We evaluated some popular plugin frameworks (Eclipse, NetBeans and Mozilla) to possibly reuse them but came to the conclusion that many of their features are unnecessary in the context of USE. Therefore, we newly designed a small and less complex but sufficiently powerful framework. Like other frameworks, the USE plugin framework provides so-called extension points which plugin developers can use to add features to the system. The following extension points are available:

Action This extension point allows a plugin to add new information and interactive elements to the graphical user interface of USE, e. g., for providing new menu commands to open windows or for executing functions offered by a plugin.

Shell In USE, most of the tasks can be invoked by shell commands. The respective extension point enables plugins to add new commands, for example, with respect to validation tasks.

Model In order to integrate new model elements into the model browser of USE, this extension point can be used.

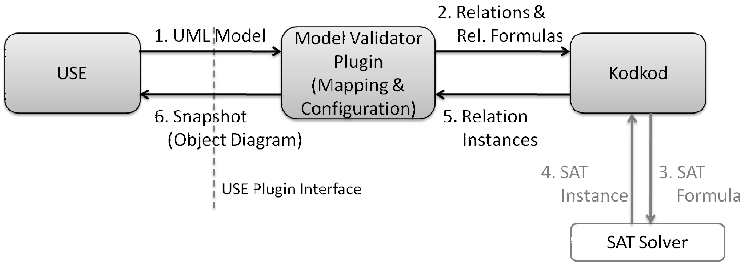

Fig. 1. Translation process involving the USE model validator plugin

These extension points can be seen as controlled hooks to the system which are automatically invoked at runtime to add new functionality and to allow the plugins to execute calculations. This kind of controlled execution is often called the Hollywood Principle ("don't call us, we'll call you" or applied to a framework "don't call the framework, the framework will call you") which distinguishes the plugin framework from a library which does not take care of any execution process. USE plugins can rely on other plugins which is supported by services a plugin may provide. The registration and initialization process of the plugins at the system start up requires that each plugin provides a configuration file with meta data about it. The latter includes information about the main class of the plugin, the provided extensions to the described extension points and information about the offered services. Whereas USE does not know anything about the concrete plugins, a plugin itself can make use of all public features of the USE system which are accessible in a specific context, e. g., the current system state (snapshot) or a parent window.

3.2 The Model Validator Plugin

The model validator plugin connects the UML and OCL side with the world of pure relations and the related logic. On the one hand, it includes Kodkod for the relational part. On the other hand, it utilizes the USE plugin interface to access the UML and OCL part as well as to extend the USE GUI. Generally speaking, the plugin is used to find snapshots fitting the given UML class diagram and fulfilling all further specified OCL constraints.

The resulting translation process is shown in Fig. 1. The plugin is first requesting the model information from USE. Then it maps the UML model including the OCL constraints into relations and relational formulas. It also adds configuration information given by the user within the USE GUI. This information determines the search space as it needs specific bounds. Kodkod performs the task of translating the given relations and relational formulas into boolean formulas which are handled by a SAT solver. The concrete SAT solver to be used can be determined via the plugin. A possibly returned SAT instance is translated into a set of relation instances which fulfill the relational formulas. The plugin maps these relations to objects, attribute values and links, i. e., a snapshot which is forwarded to the USE system and in the end is displayed in form of an object diagram. The UML and OCL concepts currently supported by the model validator are shown in Tab. 1.

Table 1. UML and OCL features supported by the USE model validator

Language	Concepts
UML	classes and attributes (enumeration, object, integer, boolean, string type) associations and association classes (binary, n-ary) inheritance and abstract classes
OCL	attribute access and navigation sets, bags, sequences, ordered sets Set, OclAny, Boolean, String and Integer operations

4 Automatic Validation with the USE Model Validator

Within this section we examine multiple alternatives for analyzing a UML and OCL model with the aid of the USE tool and in particular the new USE model validator which considerably enhances the existing USE validation scope.

4.1 Validation Principles

USE helps developers during design and analysis of their models. Common validation purposes in the context of the USE system are the following:

1. Create and check positive model snapshots which are valid in the eyes of the developer. If USE does not accept the given snapshot as valid system state in the context of the OCL constraints, the model is too restrictive, i. e., the constraints are too strong.
2. Create and check negative (i. e., invalid) snapshots. If USE accepts them as valid model instances although the developer does not, the given constraints are too weak.
3. Check model consistency in general. If there are conflicting constraints, it is not possible to create a meaningful snapshot at all, i. e., all snapshots which are valid in the developers' eyes are forbidden by the UML and OCL model.
4. Check constraint independency, i. e., if each OCL constraint adds essential information to the model and is not implied by other constraints (avoid redundancy). When a constraint is independent from the others, there must be at least one snapshot which fulfills all but the considered constraint.
5. Check user-defined properties of the model, i. e., if the model implies a user-specified property given as an OCL expression.

The built-in USE snapshot generator which executes user-defined ASSL procedures adds further validation possibilities, since it allows to automatically generate new system states. It also helps to perform the aforementioned validation aspects by searching system states with specific properties. In [9] we introduce the ASSL language and the generator features. In [11] we apply the generator for checking model consistency, constraint independency and further model properties (consequences).

An ASSL procedure must be manually defined. Through 'try' statements a procedure can describe a complete set of snapshots, that is, a snapshot space. Starting such a procedure with the USE generator, it creates one snapshot conforming to the procedure after another until a valid one was found (invoking backtracking if the found snapshot was invalid). The main disadvantage of the

built-in generator is its enumerative nature, as it has to create and check each snapshot if the procedure's snapshot space does not include a snapshot fulfilling all model constraints. As a consequence, larger snapshot spaces which for instance comprise more than a few objects and attribute values or all possible link constellations cannot be handled.

We remove this problem with the model validator which substitutes the enumerative search by an efficient SAT-based approach. In the context of the plugin, a snapshot space to be searched is not determined by an ASSL procedure, but by general bounds to the number of objects, links and attribute values, making the configuration considerably less time-consuming. In the following, we present the main features of the model validator.

Completion of partial snapshots In many validation scenarios, a developer has concrete situations of the modeled system and related properties in mind. The model validator enables an automatic analysis based on concrete situations which are manually specified as partial model snapshots. Given a snapshot and an optional property to be checked explicitly (specified in form of an OCL expression), the model validator can automatically complete the snapshot so that all model constraints and the optionally examined property are fulfilled (provided that the searched snapshot space includes a fitting snapshot), while leaving the predefined snapshot elements unchanged.

Complete snapshot generation The model validator also allows the users to generate complete snapshots which do not base on a partial solution. This way, the generated snapshots purely depend on the user-specified snapshot search space, the model constraints and given OCL properties to be examined.

Search space configuration The search space of the model validator needs to be bounded with respect to the number of the snapshot elements. More precisely speaking,

- the number of objects for each class,
- the number of links for each association, and
- the range of values for each class attribute

can be restricted through lower and upper bounds. Whereas all classes need an explicit upper bound, the maximum number of links can be left open (indicated by the value -1), since the number of all possible link combinations is restricted by the number of existing objects. If a partial snapshot is predefined, the lower bounds are automatically adapted and existing attribute values are kept. Furthermore, the existing OCL invariants can be individually activated, deactivated and negated before invoking the snapshot search.

4.2 Explanation of the Model Validator Features

Our following examinations base on a small but nontrivial UML model shown in Fig. 2. We play the role of a developer who created the model with USE and aims to validate the current version.

Running Example. The model declaratively describes a model transformation. The classes Person, Female and Male as well as the associations Parenthood and Marriage represent the source side (source meta model) of the transformation.

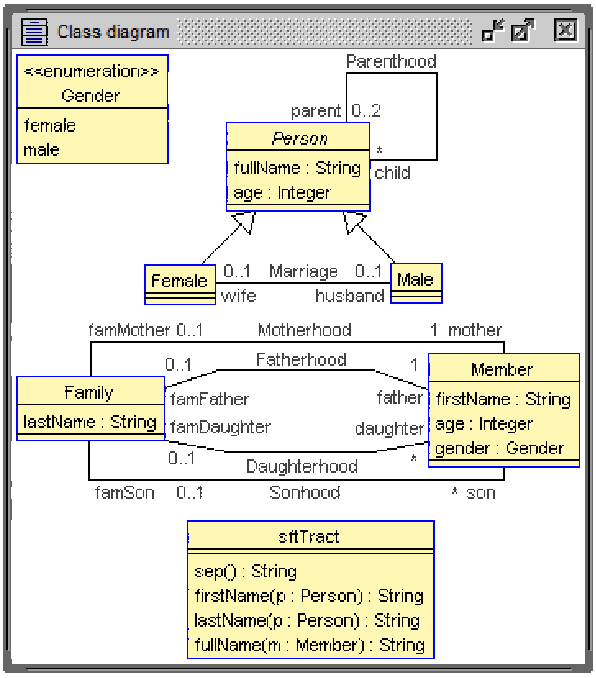

Fig. 2. Class diagram of the example UML model

Instances of the source side are to be transformed into instances of the target meta model consisting of a family and family members who can take four different roles. The full name of a person (source side) is thereby split into a last name (Family) and a first name (Member). Furthermore, the gender of a member is given by an attribute in the target model. The third part of the UML model is a 'simple family tree' (sft) class which observes the source and target models and checks them with respect to the transformation's correctness, i. e., it enforces the transformation contract. The class diagram is supplemented by the following OCL invariants. The first constraint applies to the class Person of the source meta model. We have already explained it in Sect. 2.

```
context Person inv SMM_parentsFM:
  parent->size()=2 implies
    parent->select(oclIsTypeOf(Female))->size()=1 and
    parent->select(oclIsTypeOf(Male))->size()=1
```

The second invariant constraints the target meta model. Mothers and fathers must be female or male, respectively. This invariant has the same purpose as the former invariant at the source model side.

```
context Family inv TMM_mumFemale_dadMale:
  mother.gender=#female and father.gender=#male
```

The transformation contract class adds constraints regarding the transformation. A source model must have the following property. All full names must consist of a first name and a last name separated by a blank character. The

invariant makes use of user-defined OCL query operations (sep, firstName and lastName).

```
sep():String= ' '
firstName(p:Person):String=
  p.fullName.substring(1,p.fullName.indexOf(sep())-1)
lastName(p:Person):String=
  p.fullName.substring(p.fullName.indexOf(sep())+1,p.fullName.size)

inv SRC_fullName_EQ_firstSepLast:
  Person.allInstances->forAll(p|
    p.fullName=firstName(p).concat(sep()).concat(lastName(p)))
```

A further OCL query operation calculates a family member's full name by concatenating the family last name and the member's first name (independent from the member's role in a family). This operation is used in the main invariant which constraints the actual transformation from a source to a target model.

```
fullName(m:Member)=
  let fam=
    if m.famSon.isDefined() then m.famSon else
    if m.famDaughter.isDefined() then m.famDaughter else
    if m.famFather.isDefined() then m.famFather else m.famMother
    endif endif endif in
  m.firstName.concat(sep()).concat(fam.lastName)
```

Each person must have exactly one family member counterpart. The person's and respective member's full name are identical. Their age and gender must coincide. The number of children with respect to persons and members must be equal. Children must be sons or daughters, respectively, at the target side. Married persons imply members who are a mother or father, respectively.

```
inv SRC_TRG_forPersonOneMember:
  Female.allInstances->forAll(p| Member.allInstances->one(m|
    p.fullName=fullName(m) and p.age=m.age and m.gender=#female and
    (p.child->notEmpty() implies (let fam=m.famMother in
      p.child->size()=fam.daughter->union(fam.son)->size())) and
    (p.parent->notEmpty() implies m.famDaughter.isDefined()) and
    (p.oclAsType(Female).husband.isDefined() implies
      m.famMother.isDefined()) )) and
  Male.allInstances->forAll(p| Member.allInstances->one(m|
    p.fullName=fullName(m) and p.age=m.age and m.gender=#male and
    (p.child->notEmpty() implies (let fam=m.famFather in
      p.child->size()=fam.daughter->union(fam.son)->size())) and
    (p.parent->notEmpty() implies m.famSon.isDefined()) and
    (p.oclAsType(Male).wife.isDefined() implies
      m.famFather.isDefined()) ))
```

Validation Based on a Partial Snapshot. In order to check the main transformation properties, we provide a concrete source model, i.e., an instance of the source meta model, in form of a snapshot displayed in Fig. 3. To keep the diagram clear, we hide the role names, since they follow from the family tree structure. Instead of manually defining a fitting or non fitting family target

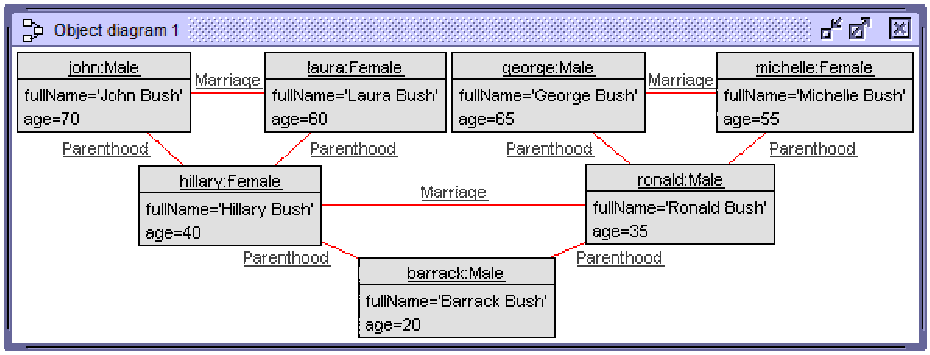

Fig. 3. Manually defined source model (partial snapshot)

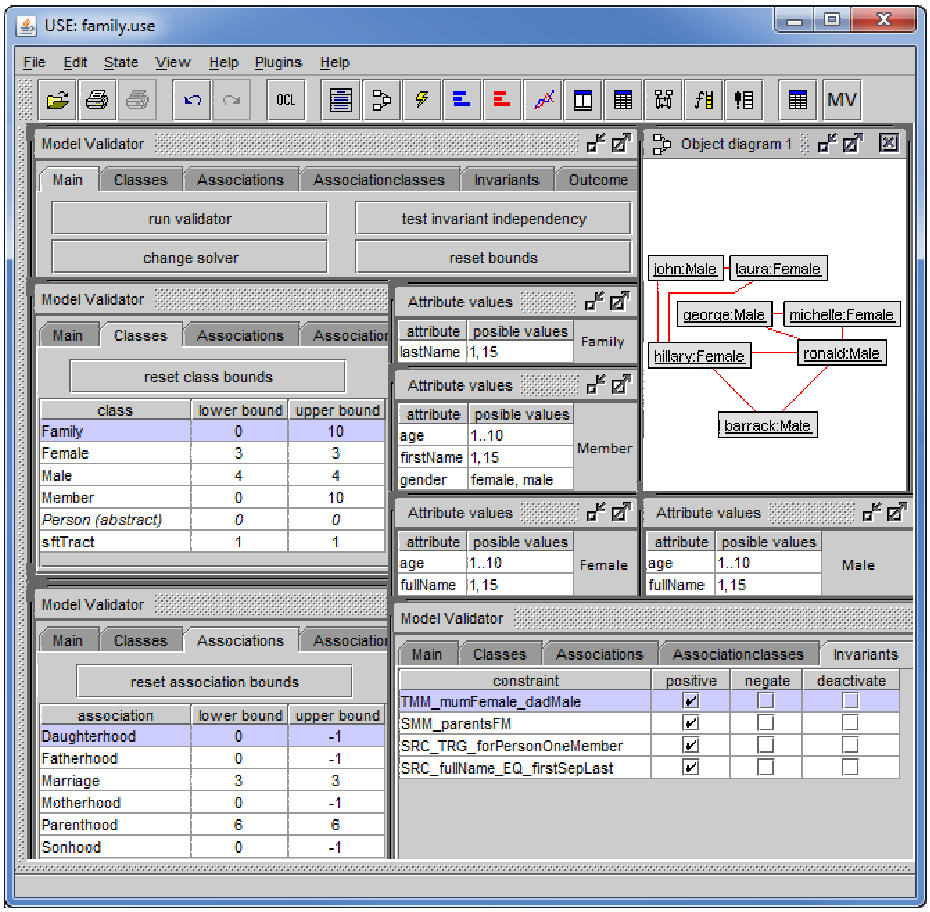

Fig. 4. Graphical user interface of the OCL2Kodkod plugin within USE

model, we order the model validator to automatically search a valid target side which we can in turn contrast with our expectations.

Figure 4 depicts a screenshot of the USE GUI which contains configuration windows provided through the model validator plugin. The screenshot shows all information necessary for the first invocation of the validator including the partial snapshot (shown in the USE screenshot with hidden attributes and roles). Since we do not want the source model to be extended by further male and female objects or links we equalize the corresponding classes' and associations' lower and upper bounds. String type attributes are bounded by determining the minimum and maximum string length. However, the settings for the attributes of class Female and Male are not relevant in the current validation context because they would only apply to newly generated Female and Male objects.

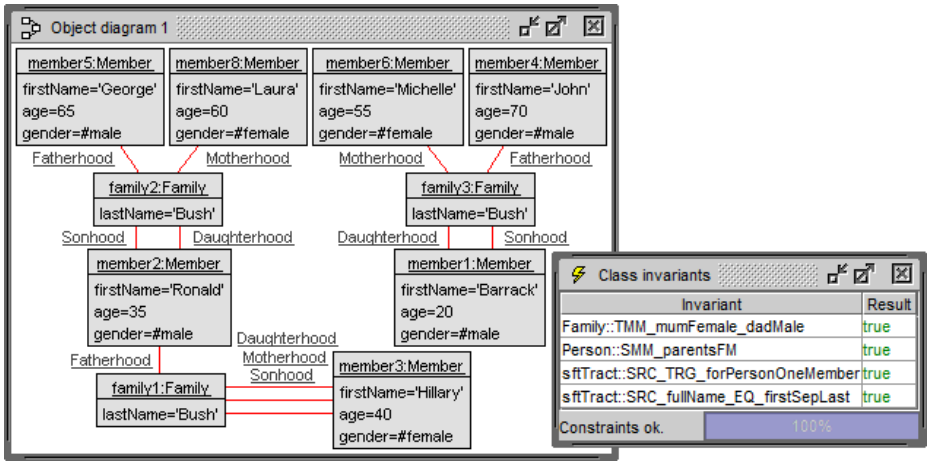

Fig. 5. Automatically generated target with unwanted properties

Finally, we activate all available invariants ordering the model validator to respect them in the solving process. About ten seconds after starting the search process (on an ordinary desktop pc), we obtain the target model shown in Fig. 5. We hide the source model because it remained unchanged as well as the generated sftTract object. The target model and with it the whole transformation represent a valid system state, since all UML and OCL constraints are fulfilled. But we immediately see several unwanted properties like members taking more than one role in one family indicating a too weak set of constraints. Thus, we extend the model by the following invariant which forbids more than one link from one member to a family object.

```
inv TMM_oneRoleInOneFamily:
  Family.allInstances()->forAll(fam|
    fam.mother<>fam.father and
    fam.daughter->union(fam.son)->excludesAll(Set{fam.mother,fam.father})
    and fam.daughter->intersection(fam.son)->isEmpty())
```

A second searching run with unchanged bounds yields a target which looks fine at first view (Fig. 6). However, in the eyes of the developer this result should

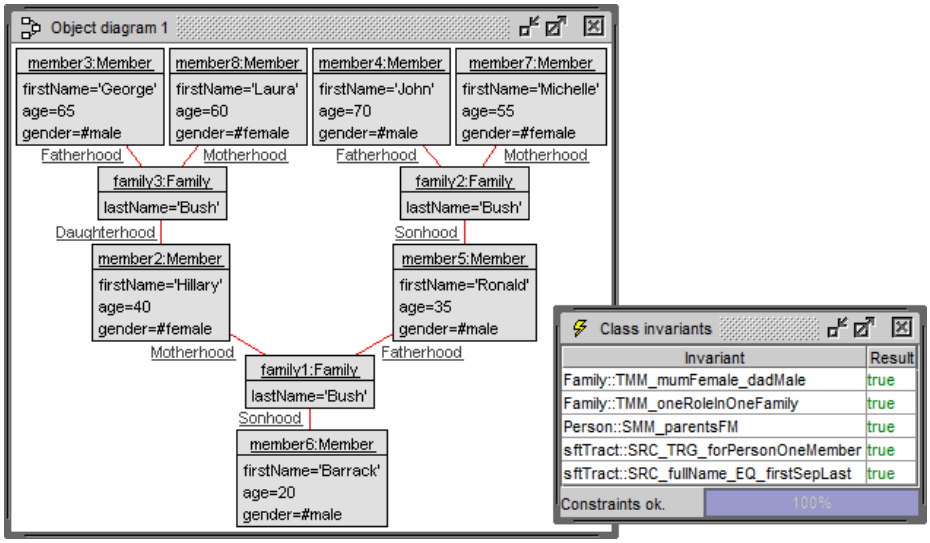

Fig. 6. Automatically generated target with switched husbands

be still invalid because Laura and Michelle switched their husbands during the transformation. As a consequence, George's child is Hillary instead of Ronald. The transformation constraint SRC_TRG_forPersonOneMember accepts this situation because it just enforces the same number of children of a person (source) and the corresponding member. We strengthen the invariant by adding a further restriction to the 'same size' requirement. The children of a person must have the same full name at both transformation sides. Generating a new target based on the adapted model yields the intended result, i. e., the same target as shown in Fig. 6 but without switched husbands.

```
p.child->forAll(c| fam.daughter->union(fam.son)->one(mc|
  fullName(mc)=c.fullName))
```

In the next step, it is reasonable to check consequences from the model's constraints. This way, the developers can check if their model implies the proper properties or not imply unwanted properties. We exemplarily consider person names, since names and their transformation present a main aspect in this model. We would like to check if our model allows family trees with different last names. At first, we change the full names of George and Michelle to 'George Obama' and 'Michelle Obama' in the predefined partial snapshot (Fig.3). These changes lead to the fact that the model validator does not find any valid target model in a search space with up to 10 members and 10 families, thus different last names are in this context not allowed.

The validation by complete snapshot generation can be performed the same way as described before except that no object diagram is given as a partial solution. In [12] our example model is explained in greater detail, including a discussion of validation of models and model transformations.

Table 2. Overview of the mapping from UML concepts to relations and formulas

UML	Relation	Formula
class	unary ($[[object_1], \ldots]$)	no formula
enumeration	unary ($[[literal_1], \ldots]$)	no formula
n-ary association	n-ary ($[[obj_{11}, \ldots, obj_{1n}], \ldots]$)	typing, multiplicity
n-ary association class	n+1-ary ($[[assoc_obj_1, obj_{11}, \ldots, obj_{1n}], \ldots]$)	typing, multiplicity
attribute (basic, object, enumeration and set type)	binary ($[[obj_1, val_1], \ldots]$)	typing, multiplicity
attribute (string type)	ternary ($[[obj_1, pos_1, char_1], \ldots]$)	typing, multiplicity
class inheritance ($A < B$)	unary (B), unary ($B - A$)	subset

5 Translation from UML and OCL into Kodkod

In this section we sketch the translation of UML concepts (outlined in Tab. 2) and OCL operations into relational structures and formulas. We illustrate selected translations on the basis of the concrete UML elements Person (class), age (attribute) and Parenthood (association).

Translated UML concepts result in relations whose tuples hold elements with a specific meaning. The assumption that, for example, class relations consist of objects, enumeration relations comprise enumeration literals, or integer type attribute relations yield pairs of objects and integer values must be made explicit, since the elements of a tuple are originally just plain atoms of the Kodkod universe. Bound declarations generally type the relations by assigning atoms to the corresponding relations, e. g., relating Person objects (e. g., ada, bob and cyd) to the Person relation, to the relation of the attribute age, and to the Parenthood association relation. A concrete Kodkod solution yields all elements of the lower bounds and a subset of the upper bounds. The upper bounds comprise all fitting atoms and all possible resulting tuples (i. e., the Cartesian product of the atoms) if no concrete tuples are defined in the lower bounds. Thus, the types and multiplicities of the association ends, as well as the attribute domains and values must be further constrained using formulas which apply to concrete relation instances in a solution.

Bounds: (relation name, lower bound, upper bound)

`Person`:	$[[ada]]$	$[[ada], [bob], [cyd]]$
`ints`:	$[\,]$	$[[1], [2]]$
`age`:	$[[ada, 1]]$	$[[ada, 1], [bob, \bot], [bob, 1], [bob, 2], [cyd, \bot], [cyd, 1], [cyd, 2]]$
`Parent hood`:	$[\,]$	$[[ada, ada], [ada, bob], [ada, cyd], [bob, ada], [bob, bob], [bob, cyd], [cyd, ada], [cyd, bob], [cyd, cyd]]$

Formulas:

`age`	(domain/type):	$(age.univ)$ *in* $Person$ & $(univ.age)$ *in* $(ints + \bot)$
	(multiplicity):	*all* $c : Person \mid$ *one* $(c.age)$
`Parent hood`:	(type)	$(Parenthood.univ)$ *in* $Person$ & $(univ.Parenthood)$ *in* $Person$
	(multiplicity):	*all* $c2 : Person \mid \#(Parenthood.c2) <= 2$

The given example bounds determine that Ada must be included in a valid solution, while the existence of Bob and Cyd is optional. There may be two integer atoms representing the numbers 1 and 2. Ada must be one year old. Bob's and Cyd's age is not constrained, it may be undefined ($\perp$). UML associations are translated into Kodkod relations in which each association end takes one position (cf. Tab. 2). The example bounds of the Parenthood relation allow all parenthood constellations. As a consequence, a solution may yield the Parenthood tuple $[ada, bob]$ (representing the UML link $(parent : ada, child : bob)$ and at the same time a Person relation which does not include the atom *bob*. This type inconsistency is prevented by the shown formulas which demand that (1) the tuples of the age relation yield Person objects at the first position and integer atoms or the undefined value at the second position (the expression *univ* represents the relation including all atoms existing in the universe), (2) the relation age connects each person object existing in a solution to exactly one (un)defined value, (3) both positions of Parenthood tuples hold Person objects, and (4) the Parenthood relation connects each Person object at the second position (child) to at most two Person objects (parents).

The translation of OCL operations profit from the large collection of operations provided by Kodkod. Thus, many OCL operations (e. g., integer and set operations) have a counterpart in the relational logic (e. g., the OCL expression `set->size<=2` can be expressed as $\#(set) <= 2$ in Kodkod). However, some OCL peculiarities like the undefined value which can occur in all OCL expressions often result in more complex translations. In the following we present the translation of an example navigation which is a central feature in OCL.

In OCL we navigate from a person p to her parents through the expression `p.parent`. Analogously, her children are calculated with the expression `p.child`. At the Kodkod side, we have no explicit role names. Thus, the association ends are determined through the positions of the tuple elements, i. e., the first end (parent) takes the first position, the second end (child) takes the second. Consequently, the navigation from a person to the parents yields the relational join $Parenthood.p$. The navigation to the children results in $p.Parenthood$. This straightforward translation is only applied in the case of binary associations. However, the respective Java method implementing the navigation in Kodkod must respect all kinds of associations (n-ary and association classes) and any possible navigation from association end x to end y including the navigation from and to association class objects.

6 Related Work

The relational language of Kodkod directly corresponds to the relational logic of Alloy [13]. The Alloy language is implemented in the Alloy Analyzer which currently bases upon the Kodkod tool. The UML2Alloy approach [1] translates UML and OCL concepts into the Alloy language which is in turn translated by the Alloy Analyzer into Kodkod structures. Our direct approach of translating UML and OCL into the Kodkod relations and formulas is not limited by modeling restrictions of the Alloy language. Thus, we are able to handle several concepts which are not realizable with Alloy like multiple inheritance or a proper

treatment of undefined values (e. g., OCL2Kodkod distinguishes between empty sets, undefined sets and sets including an undefined value). UML2Alloy further does not support concepts like n-ary associations and association classes, OCL sequences and bags, or string and integer operations.

Approaches like [27] (specification of enterprise architecture models based on ontologies) and [15] (definition of modeling languages and their formal semantics) directly specify and analyze with Alloy. Kodkod has also been successfully applied in different fields, e. g., for executing declarative specifications in case of runtime exceptions in Java programs [21], reasoning about memory models [24], or generating counterexamples for Isabelle/HOL a proof assistant for higher-order logic (Nitpick) [3]. Whereas Nitpick searches instances to disprove given lemmas (e. g., through the assignment of free variables to concrete values), the model validator is intended for finding snapshots which conform the UML and OCL constraints, as well as snapshots violating constraints. Both kinds of snapshots can help the developer to reveal specific model properties.

Other approaches do not employ Alloy or Kodkod for connecting SAT techniques with UML and OCL. A direct mapping of UML and OCL concepts into SAT has been addressed in [22]. However, while a direct translation is not limited with respect to the peculiarities of an intermediate language, it cannot benefit from existing translation mechanisms like the sophisticated symmetry detection and breaking scheme which enables an efficient handling of partial solutions, or the detection and exploitation of redundant structures in formulas which are implemented in Kodkod [23]. Several other (non-SAT-based) techniques support the analysis of UML and OCL models. A translation of specific UML and OCL features into constraint satisfaction problems (CSP) is done in [5, 25]. Furthermore, the constructive query containment (CQC) method [20], rewriting-based techniques [7] or answer set programming (ASP) [19] is applied for analyzing static and dynamic model aspects. In [2] the authors use description logic for reasoning about UML class diagrams. An approach to explicitly checking properties of metamodels is presented in [18] . UMLAnT [26] allows for analyzing model behavior by animating object and sequence diagrams. Beside model animation and simulation Dresden OCL [8] supports constraint verification by code generation and execution, but like the UML tools OCLE [6] and RoclET [14], it does not support automated snapshot generation producing model instances with user-defined properties. In the context of the Eclipse Modeling Framework (EMF) the Epsilon Language [16] provides for model validation based on constraints similar to OCL. The named approaches differ from more interactive approaches like [4] involving formal verification by theorem proving.

Our approach is unique with respect to the use of Kodkod as a conceptual intermediate language for translating the source languages UML and OCL into boolean formulas.

7 Conclusion and Future Work

With the method OCL2Kodkod and its realization as the USE model validator we presented an automatic SAT-based approach for efficiently finding snapshots in large search spaces, i. e., a lightweight way to validate UML and OCL

model properties resulting from complex constraints and their often intransparent interdependencies. By immediately translating found SAT solutions back into snapshots displayed in form of object diagrams the model developers receive a detailed feedback on the UML and OCL modeling layer.

Future work comprises the enhancement of supported UML and OCL features like qualified associations, subset relations between association ends as well as OCL operations on sequences, bags, ordered sets and tuples. We will also further optimize the translation into the relational logic as well as the model validator's GUI options allowing developers a more comfortable configuration. Larger case studies are planned, in particular for model transformations employing the completion abilities of the model validator (both for the transformation source or target). Beside analyzing static model properties, we aim to involve behavioral features (operations) through transformation of UML and OCL application models into so-called snapshot models.

The flexible plugin architecture of USE is well-suited for further open-source development in research institutes. USE plugins for various purposes are planned, for example, with respect to special user-interfaces for particular diagrams (e. g., object histories in object diagrams or OCL assertions on lifelines in sequence diagrams), or the realization of domain-specific languages like an RBAC DSL [17].

Acknowledgements
We would like to thank the referees for their constructive comments.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. SoSyM 9(1), 69–86 (2010)
2. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. Artif. Intell. 168(1-2), 70–118 (2005)
3. Blanchette, J.C., Nipkow, T.: Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In: Kaufmann, M., Paulson, L.C. (eds.) First International Conference on Interactive Theorem Proving, ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Berlin (2010)
4. Brucker, A.D., Wolff, B.: HOL-OCL: A Formal Proof Environment for UML/OCL. In: FASE. pp. 97–100 (2008)
5. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL Class Diagrams using Constraint Programming. In: Proc. 2008 IEEE Int. Conf. Software Testing Verification and Validation Workshop. pp. 73–80. IEEE Computer Society, Washington, DC, USA (2008)
6. Chiorean, D., Pasca, M., Cârcu, A., Botiza, C., Moldovan, S.: Ensuring UML Models Consistency Using the OCL Environment. Electr. Notes Theor. Comput. Sci. 102, 99–110 (2004)
7. Clavel, M., Egea, M.: ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Static Class Diagrams. In: AMAST. pp. 368–373 (2006)
8. Demuth, B., Wilke, C.: Model and Object Verification by Using Dresden OCL. In: Proc. Russian-German Workshop Innovation Information Technologies: Theory and Practice. p. 81. Technical University, Ufa, Russia (July 2009)
9. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. SoSyM 4(4), 386–398 (2005)

10. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. SCP 69, 27–34 (2007)
11. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In: Dubois, C. (ed.) Proc. 3rd Int. Conf. Test and Proof (TAP'2009). pp. 90–104. Springer, Berlin, LNCS 5668 (2009)
12. Gogolla, M., Vallecillo, A.: Tractable Model Transformation Testing. In: France, R., Küster, J.M., Bordbar, B., Paige, R.F. (eds.) Proc. 7th Int. Conf. Modelling Foundations and Applications (ECMFA'2011). Springer, Berlin (2011)
13. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. The MIT Press, Cambridge, Massachusetts (2006)
14. Jeanneret, C., Eyer, L., Markovic, S., Baar, T.: RoclET- Refactoring OCL Expressions by Transformations. In: Software & Systems Engineering and their Applications,19th International Conference, ICSSEA 2006 (2006)
15. Kelsen, P., 0002, Q.M.: A Lightweight Approach for Defining the Formal Semantics of a Modeling Language. In: Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M. (eds.) Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008. LNCS, vol. 5301, pp. 690–704. Springer, Berlin (2008)
16. Kolovos, D., Rose, L., Paige, R.: The epsilon Book. http://www.eclipse.org/gmt/epsilon/doc/book/
17. Kuhlmann, M., Sohr, K., Gogolla, M.: Comprehensive Two-level Analysis of Static and Dynamic RBAC Constraints with UML and OCL. In: Baik, J., Massacci, F., Zulkernine, M. (eds.) Fifth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2011. IEEE Computer Society (2011)
18. de Lara, J., Guerra, E.: Deep Meta-modelling with MetaDepth. In: TOOLS (48). pp. 1–20 (2010)
19. Ornaghi, M., Fiorentini, C., Momigliano, A., Pagano, F.: Applying ASP to UML Model Validation. In: LPNMR. pp. 457–463 (2009)
20. Queralt, A., Teniente, E.: Reasoning on UML Class Diagrams with OCL Constraints. In: ER. pp. 497–512 (2006)
21. Samimi, H., Aung, E.D., Millstein, T.: Falling back on executable specifications. In: Proceedings of the 24th European conference on Object-oriented programming. pp. 552–576. ECOOP'10, Springer-Verlag, Berlin, Heidelberg (2010), http://portal.acm.org/citation.cfm?id=1883978.1884015
22. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL Models Using Boolean Satisfiability. In: Müller, W. (ed.) Proc. Design, Automation and Test in Europe (DATE'2010). IEEE (2010)
23. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Berlin (2007)
24. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: checking axiomatic specifications of memory models. SIGPLAN Not. 45, 341–350 (June 2010), http://doi.acm.org/10.1145/1809028.1806635
25. Treharne, H., Schneider, S., Grant, N., Evans, N., Ifill, W.: A Step towards Merging xUML and CSP || B. In: Rigorous Methods for Software Construction and Analysis. pp. 130–146 (2009)
26. Trong, T.D., Ghosh, S., France, R.B., Hamilton, M., Wilkins, B.: UMLAnT: An Eclipse Plugin for Animating and Testing UML Designs. In: Proc. OOPSLA workshop Eclipse technology eXchange. pp. 120–124. ACM, New York, NY, USA (2005)
27. Wegmann, A., Le, L.S., Hussami, L., Beyer, D.: A Tool for Verified Design using Alloy for Specification and CrocoPat for Verification. In: Jackson, D., Zave, P. (eds.) Proc. First Alloy Workshop (2006)