Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool

Martin Gogolla[1], Frank Hilken[1]

Abstract: Modern systems and their architectures are getting more complex than ever. Development strategies, like model-driven engineering (MDE), help to abstract architectures and provide a promising way to deal with the complexity. Thus, the importance for the underlying models to be correct arises. Today's validation and verification tools should support the developer in generating test cases and provide good concepts for fault detection. In this contribution, we introduce and structure essential use cases for model exploration, validation and verification that help developers find faults in model descriptions. Along with the use cases, we demonstrate the model validator of the USE tool, a modern instance finder for UML and OCL models based on an implementation of relational logic and present the results and findings from the tool.

1 Introduction

Model-driven engineering (MDE) takes a view on system development focusing rather on models than on code. A model catches a system by abstracting its complexity through reduction of information, however preserving properties relative to a given set of concerns. Today, modeling languages, such as the UML (Unified Modeling Language) which comprises the OCL (Object Constraint Language), have found their way into mainstream software development. Models are the central artifacts in MDE because other software elements like code, documentation or tests can be derived from them using model transformations. Common model quality improvement techniques are model validation ("Are we building the right product?") and verification ("Are we building the product right?") [Bo89]. Among the different aspects of a system to be caught, structural aspects represented by class and object diagrams are of central concern.

The tool USE (UML-based Specification Environment) [GBR07] supports the development of UML models enhanced by OCL constraints. USE offers class, object, sequence, statechart, and communication diagrams. It facilitates class and state invariants as well as pre- and postconditions for operations and transitions formulated in OCL. It allows the modeler to validate models and to verify properties by building test scenarios. One USE component that is in charge for this task is the so-called model validator that transforms UML and OCL models as well as validation and verification tasks into the relational logic of Kodkod [TJ07], performs checks on the Kodkod level, and transforms the obtained results back in terms of the UML and OCL model. The modeler works on the UML and OCL level only without a need for expressing details on the relational logic level, i.e., on the Kodkod level.

[1] University of Bremen, Computer Science Department, E-Mail: {gogolla,fhilken}@informatik.uni-bremen.de

The starting point for our current approach is a structural UML model (class diagram) enriched by OCL invariants. With a small, but versatile running example, we discuss various use cases for model validation and verification: model consistency, property reachability, constraint implication, constraint independence, solution interval exploration, and partial solution completion. For example, *model consistency* means that classes and associations considered together with the OCL constraints can be instantiated in form of an object diagram, or *solution interval exploration* means that not only a single instantiation in form of objects and links is examined but all instantiations are taken into account and may be inspected for validation and verification.

The running example here is rather small, but we have already checked that some use cases work for larger models as well [Go15]. Additionally, all presented use cases do not only benefit the USE tool, but also other verification engines [CCR14, Wi08, Br10, QT06] can be used to perform these tasks. Usually, some modifications to the model constraints or additions of such constraints is enough to adopt the tasks. In comparison to the other approaches, however, our current method has the highest coverage of OCL features and offers the most high-level interactions for the use cases.

The rest of the paper is structured as follows. Section 2 gives background information and recaps application areas of UML and OCL models, introduces our running example, shows the basic functionality of USE, and sketches relational logic and Kodkod. The central Sect. 3 discusses the main validation and verification use cases that can be realized, among them how to check model consistency and consequences from stated constraints. Section 4 puts our approach into the context of known work. The paper ends with a conclusion and future work in Sect. 5.

2 Preliminaries

2.1 Application Areas for UML and OCL

UML together with OCL have been succesfully used for system modeling in numerous industrial and academic projects. Here, we refer to only three example projects trying to indicate the wide spectrum of application options. In our own early work [ZG03], we have specified safety properties of a train system in the context of the well-known BART case study (Bay Area Rapid Transit, San Fransisco). In [Al11], central aspects of an industrial video conferencing system developed by Cisco have been studied. In [OM13], UML and OCL are employed for the specification of the UML itself by introducing the so-called UML metamodel in which fundamental well-formedness rules of UML are expressed as OCL constraints.

2.2 Example UML and OCL Model in USE

The screenshot in Fig. 1 shows how the running example employed here is represented in USE. The class diagram in the top right contains one class `Person` having a *first name*, a

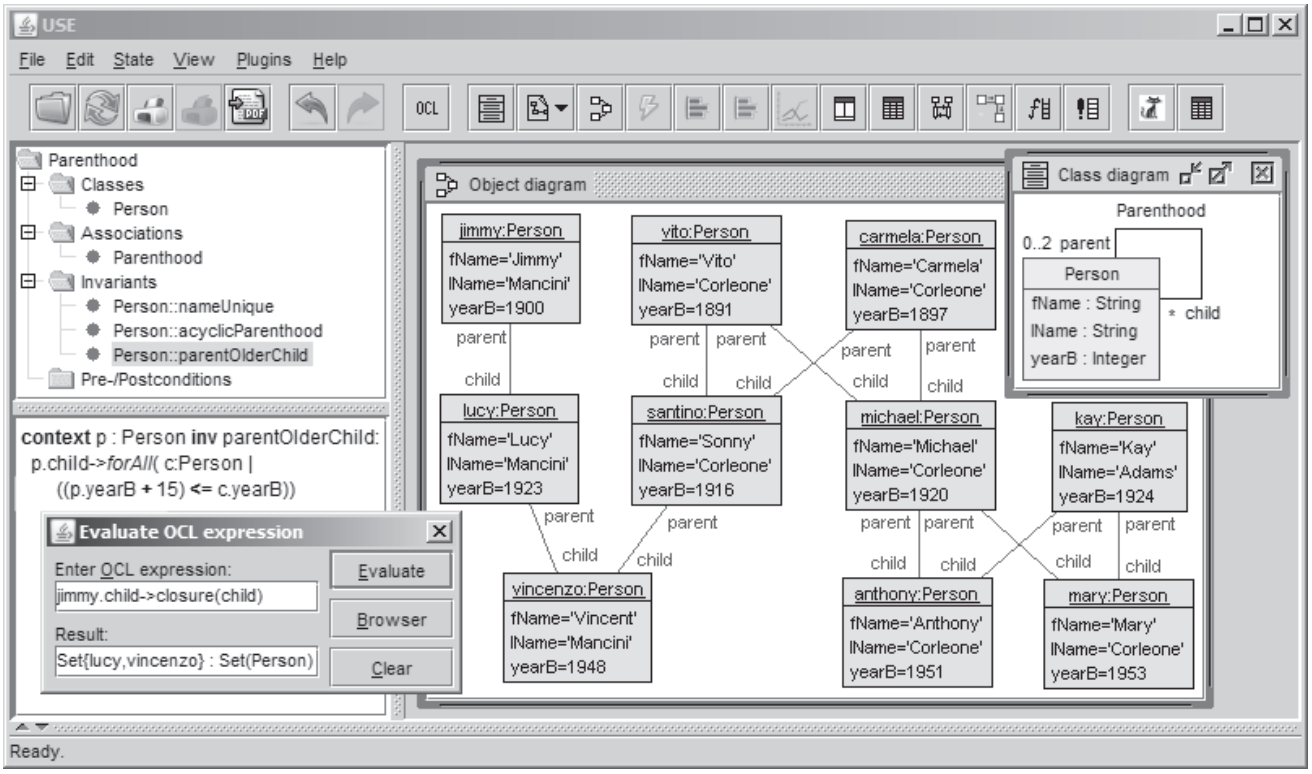

Fig. 1: Parenthood Example class and object diagram.

last name and the *year of birth* as attributes. The association `Parenthood` resembles the parent-child relationship with at most two parents and arbitrary many children per person.

The model may be employed to present family trees as UML object diagrams. To enforce rational family trees three simple invariants are employed, thereby restricting the set of allowed system states, that is defined by the class diagram. These OCL constraints ensure unique names for all persons, prevent cycles in the parenthood relationships and require parents to be at least 15 years older than their children.

```
context p1, p2 : Person inv nameUnique:
  p1<>p2 implies (p1.fName<>p2.fName or p1.lName<>p2.lName)
context p : Person inv acyclicParenthood:
  p.parent->closure(parent)->excludes(p)
context p : Person inv parentOlderChild:
  p.child->forAll(c | p.yearB+15<=c.yearB)
```

The object diagram in the center of Fig. 1, showing several `Person` objects and `Parenthood` links, illustrates a system state of the model showing the family tree as present in the movie *Godfather*. In the lower left, OCL is employed for an ad-hoc query that returns all offsprings of *jimmy*. Arbitrary OCL expressions can be used here to analyze the system state. The present system state satisfies all three invariants and model inherent constraints, such as multiplicity constraints.

2.3 Relational Logic and Kodkod

The verification engine used in this paper, the USE model validator, is based on the relational logic of Kodkod [TJ07]. Kodkod defines a problem to consist of a universe, i.e., a

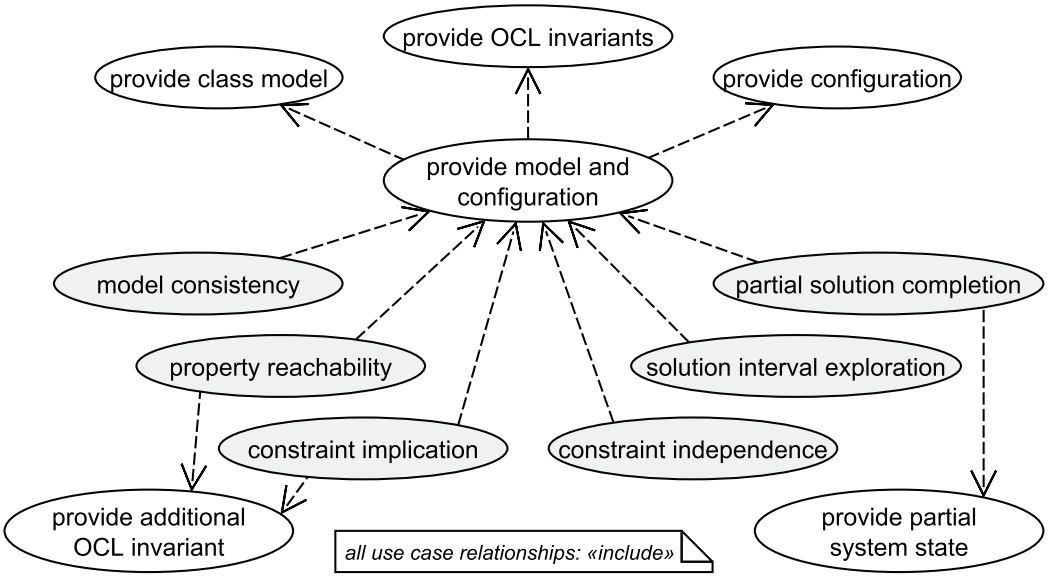

Fig. 2: Validation and verification use cases.

set of uninterpretable atoms, a set of relation declarations and a relational formula. The universe is defined by the underlying class diagram together with the configuration. The configuration specifies the number of objects available in a solution including primitive data types like `Integer` and `String`. For these types, a minimum and maximum number of instances is defined and specific values for class attributes can be specified. Finally, the relational formula is constructed from (1) UML structural constraints, (2) OCL class invariants and again (3) the configuration. The translation process is based on [KG12].

Given a class diagram and a configuration, the USE model validator generates all three aspects required for Kodkod to solve the problem instance with an off-the-shelf SAT solver. If a valid instantiation is found, the USE model validator generates the corresponding object diagram from the solution instance given by Kodkod. Otherwise, no instance exists for the given model together with the configuration and optionally[1], a counterproof is presented, hinting at possible problems in the model.

3 Validation and Verification Use Cases

The recent validation and verification options in USE will be demonstrated as shown in Fig. 2 by six use cases for central analysis tasks: model consistency, property reachability, constraint implication, constraint independence, solution interval exploration, and partial solution completion.

Figure 3 shows the uses cases from Fig. 2 and the primary input and output artifacts. The distinction between expected and unexpected output parts is as follows: for a use case one typically has in mind a particular expectation for the output of the main flow that is displayed in the upper half of the circle, whereas the output for an alternative flow is pictured in the lower half of the circle. For example, for the *constraint implication* use

[1] Whether a proof is generated depends on the employed SAT solver.

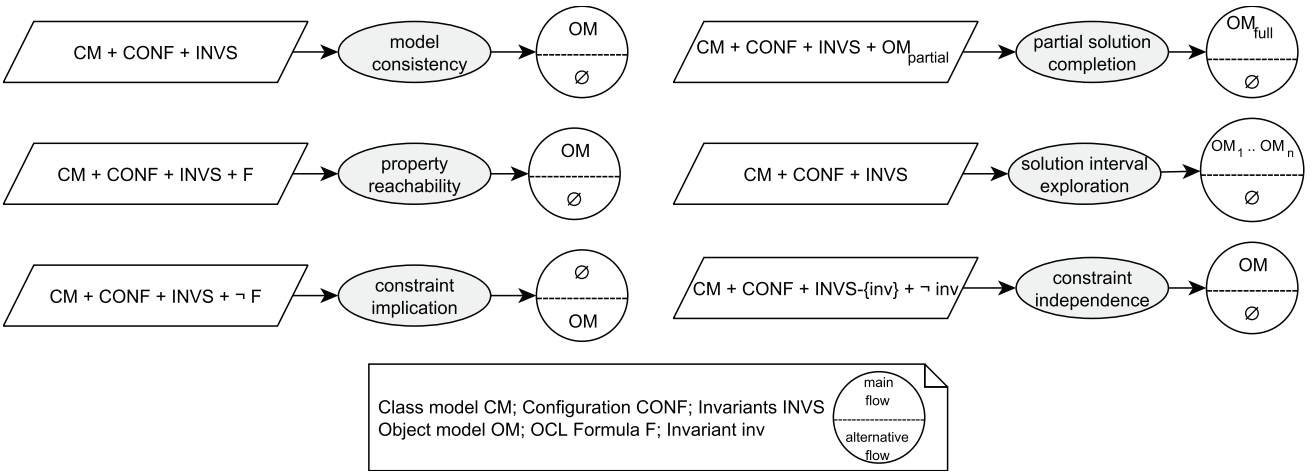

Fig. 3: Use case input and use case output for main and alternative flow.

case the expected output is that no object model is found, whereas a found object model represents a counter example for the suspected constraint consequence.

3.1 Model Consistency

Model consistency is a crucial property. In the context of a class diagram, it guarantees that the UML association multiplicities together with the explicit OCL invariants are not contradictory and that the class diagram can be instantiated with a system state, i.e., an object diagram. Within the context of a UML class diagram, sometimes this property is referred to as class liveness. Finding classes that are not live means that they cannot be instantiated and thus might be unusable in the model.

Model consistency can be proved by handing over to the model validator a configuration that describes the possible populations of classes, associations and attributes in terms of so-called bounds. In technical terms, model consistency is realized through the command `mv -validate <PropertyFile>`[2]. The model validator tries to construct within the specified bounds a valid system state (object diagram). If successful, the system state can be inspected, and if not, the model validator reports that the class diagram cannot be instantiated within the specified bounds. Such an analysis process realizes a verification task for a finite domain. The bitwidth used by the underlying solver for integer arithmetic can be configured in the model validator through the command `mv -config bitwidth:=<NumBits>`.

In the following configuration file used for the consistency use case, exactly 10 persons and 11 parenthood links together with attribute values from the stated enumerations are employed[3]. The object and link numbers and datatype values are exactly as in Fig. 1. The particular datatype values are not bound to particular objects, but the assignment is done by the model validator.

```
Person_min = 10; Person_max = 10;
```

[2] Commands, which are newly introduced, are displayed in a black-on-gray style.

[3] The syntax in the configuration files is partly slightly different. We have decided to use this self-explanatory notation.

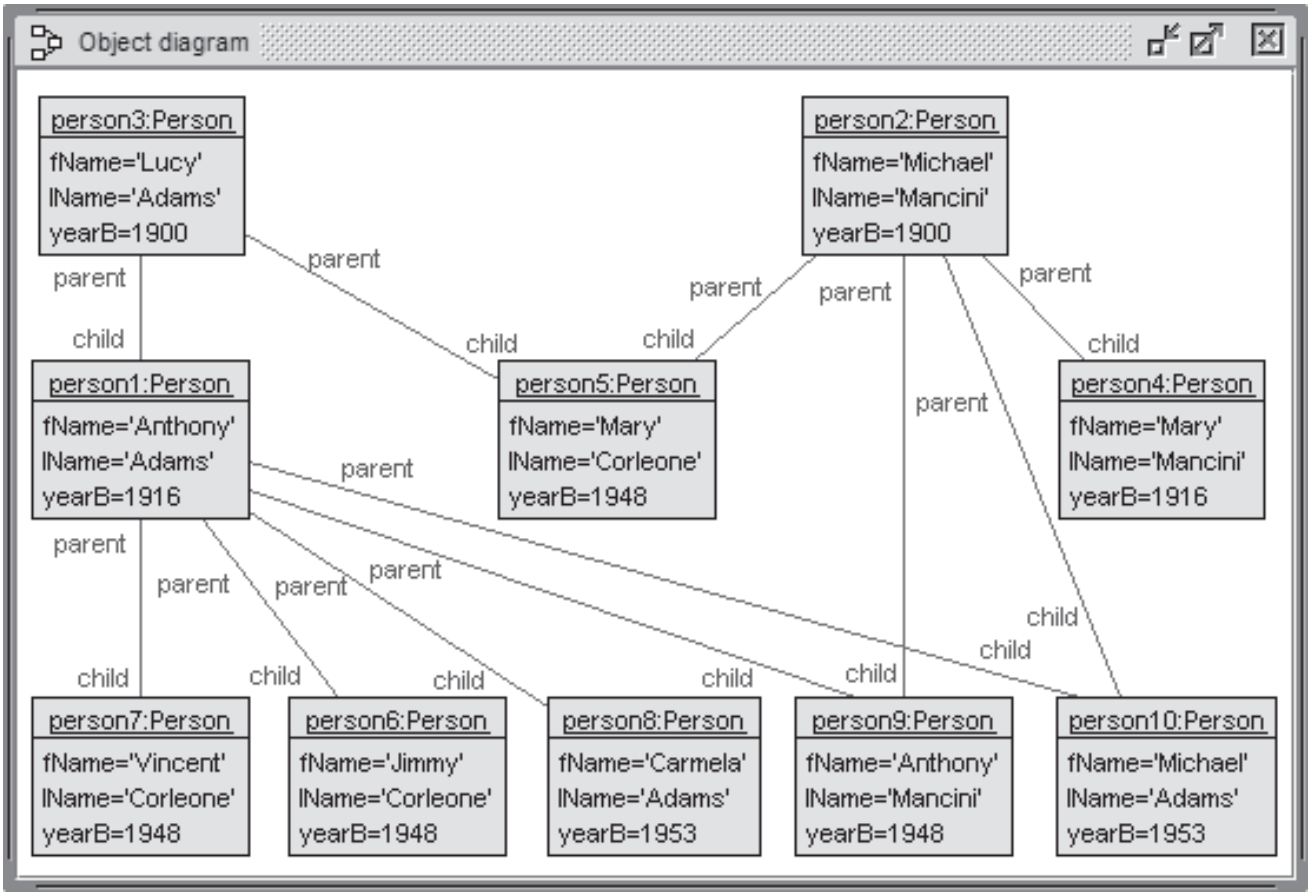

Fig. 4: Generated instantiation for model consistency.

```
Person_fName = Set{'Lucy','Jimmy','Vito','Carmela','Sonny',
    'Michael','Kay','Vincent','Anthony','Mary'};
Person_lName = Set{'Corleone','Adams','Mancini'};
Person_yearB = Set{1891,1897,1900,1916,1920,1924,1923,1948,1951,1953};
Parenthood_min = 11; Parenthood_max = 11;
```

The following protocol shows how USE is fed with the model. The bitwidth configuration in the model validator (`mv -config` command) is necessary due to the desirable realistic year numbers, however it slows down the underlying SAT solver. The validation process is kicked off with the `mv -validate` command, and the constructed object diagram is shown in Fig. 4, which is different from Fig. 1, because the model validator has chosen from the many possible solutions satisfying the specified bounds and datatype values *one* particular solution.

```
use> open parenthood.use
use> mv -config bitwidth:=12
     ModelValidatorConfiguration: Set bitwidth to 12
use> mv -validate corleone.properties
     ModelTransformator: Translation time: 234 ms
     ModelValidator: SATISFIABLE
         Translation time: 359 ms      Solving time: 5351 ms
```

The three time specifications refer to the time needed (a) to translate the class diagram including the invariants into the relational logic of Kodkod, (b) to translate the relational formula and configuration into SAT (this step is performed by Kodkod), and (c) to solve the translated relational formula by the underlying SAT solver. Setting the bitwidth is required

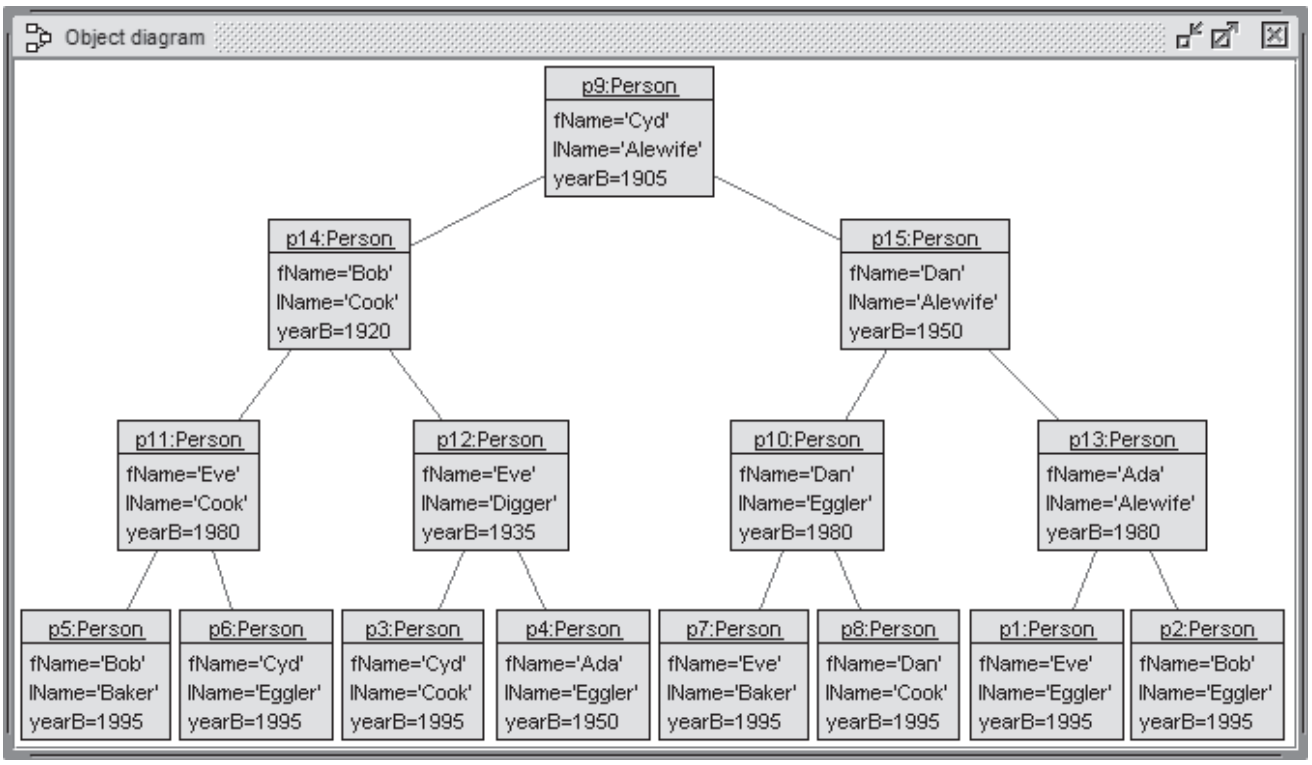

Fig. 5: Generated instantiation for property reachability.

due to the large integers for the years of birth and is necessary in the following scripts, as well. However, we will not repeat all commands below.

3.2 Property Reachability

Property reachability is another verification task that proves that a specific property can be established by object diagrams that are also valid with regard to the original model without having to modify it. Thus the object diagrams of the newly formulated property are a subset of the original object diagrams. The properties are arbitraray OCL expressions that must hold in the generated system state. Additionally, negative properties can be formulated to verify the absence of, e.g., dangerous or illegal system states, or simply unwanted constellations in the system.

In technical terms, property reachability is realized by adding another invariant to the model and by asking the model validator to instantiate the enriched model on the basis of a configuration. `constraints -load <constraintFile>` adds the constraint from the file to the current model. After starting the model validator with the original model enriched by the additional invariant employing the given configuration, the expected result should be a system state that satisfies the original model and the additional specific property.

The following invariant shows that the `Parenthood` model allows object diagrams that constitute perfectly balanced, binary trees. OCL allows to catch the essentials in condensed form.

```
context p:Person inv balancedBinaryTree:
```

```
  ---------------------------------------- balance, binary
  (p.child->size=0 or p.child->size=2) and
  ------------------------------------------------- root
  Person.allInstances->one(r | r.parent->size=0 and
    Person.allInstances->excluding(r)->forAll(p2 |
      p2.parent->size=1)) and
  ------------------------------------------------ balance
  p.child->forAll(c1,c2 | c1.child->closure(child)->size =
    c2.child->closure(child)->size)
```

In the following configuration, exactly 15 person objects are specified, whereas the number of `Parenthood` links is left open. The model validator finds out that exactly 14 `Parenthood` links are needed.

```
Person_min = 15; Person_max = 15;
Person_fName = Set{'Ada', 'Bob', 'Cyd', 'Dan', 'Eve'};
Person_lName =
  Set{'Alewife', 'Baker', 'Cook', 'Digger', 'Eggler'};
Person_yearB = Set{1905, 1920, 1935, 1950, 1965, 1980, 1995};
Parenthood_min = 0; Parenthood_max = *;
```

Specifying exact bounds for `Parenthood` (min 14, max 14) would dramatically speed up the solving process. The following protocol adds the above invariant to the model. The resulting object diagram is shown in Fig. 5. The generated system state confirms the claim, that the property does in fact hold in the running example.

```
use> constraints -load balancedBinaryTree.invs
     Added invariants: Person::balancedBinaryTree
use> mv -validate balancedBinaryTree.properties
     ModelTransformator: Translation time: 296 ms
     ModelValidator: SATISFIABLE
         Translation time: 1576 ms      Solving time: 16396 ms
```

3.3 Constraint Implication

Typically, the modeler specifies a bunch of central properties directly in terms of constraints. However, the modeler often has in mind that the constraint set guarantees that a more global property also holds, i.e., that the global property is an implication of the specified model. In order to formally check the intuitively present global property against the model, the global property is formulated as an invariant, and it is tested whether the suspected implication formally holds.

In technical terms, checking constraint implication is realized by adding the global property to the model. Then that property is logically negated with the command `constraints -flags <Invariant> +n`, and the model validator is asked on the basis of a configuration to instantiate the model. If the global property is an implication from the original model, then the model cannot be instantiated in this situation as the global, implied property has been added in logically negated form. Then the expected result is that the model is unsatisfiable. Otherwise, the model validator will construct a counter example that explains that the suspected invariant is not a model implication.

In the example, the invariant implication which we expect to hold is that a grandparent is at least 30 years older than the grandchild, formulated here as an additional OCL invariant.

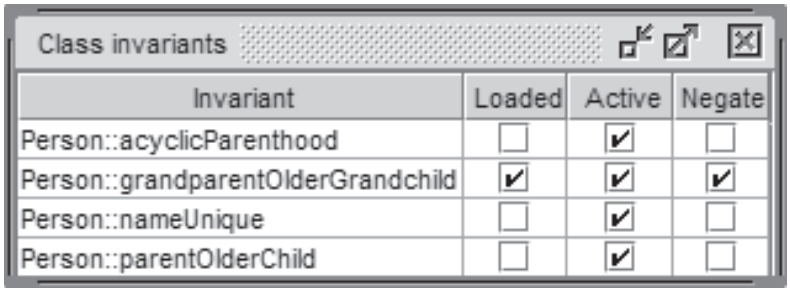

Fig. 6: Status of original and loaded invariants.

```
context gp : Person inv grandparentOlderGrandchild:
  gp.child.child->forAll( gc | gp.yearB+30 <= gc.yearB )
```

The following configuration binds the number of persons to at most 6. The `Parenthood` links are not restricted. Possible attribute values are as above.

```
Person_min = 0; Person_max = 6;
Person_fName = Set{'Ada', ..., 'Eve'};
Person_lName = Set{'Alewife', ..., 'Eggler'};
Person_yearB = Set{1905, ..., 1995};
Parenthood_min = 0; Parenthood_max = *;
```

The protocol to follow adds the previously defined invariant `grandparentOlder-Grandchild` to the model and logically negates it. The status of the invariants can be checked either on the command shell or in the USE GUI as shown in Fig. 6. The model validator reports that under the stated configuration the model including the additional negated invariant is unsatisfiable. One could increase the number of possible objects in class `Person` (`Person_max=7, 8, 9, ...`), however this will not change the resulting report. Being convinced that we have performed enough checks, we assume now that the suspected invariant is indeed an implication from the stated model.

```
use> constraints -load grandparentOlderGrandchild.invs
     Added invariants: Person::grandparentOlderGrandchild
use> constraints -flags Person::grandparentOlderGrandchild +n
use> constraints -flags
     -- active class invariants:
     Person::acyclicParenthood
     Person::grandparentOlderGrandchild (negated)
     Person::nameUnique
     Person::parentOlderChild
use> mv -validate grandparentOlderGrandchild.properties
     ModelTransformator: Translation time: 296 ms
     ModelValidator: UNSATISFIABLE
         Translation time: 171 ms      Solving time: 2590 ms
```

3.4 Constraint Independence

Constraint independence is a property of the complete set of constraints. Its goal is to check whether the constraints are independent from each other, i.e., no single constraint is an implication from the other constraints. This property may also be regarded as a kind of minimality property for the constraint set: in this case no single invariant can be removed without changing the set of object diagrams for the class diagram. Independence may or may not hold for the stated constraints. In any case it is interesting to know whether this

property holds, for example, in the context of model slicing it will be crucial to reduce the model complexity by identifying a minimal set of needed invariants.

With regard to technical realization, the model validator is started with the option `mv -invIndep <PropertyFile> all`. The result will be a statement for each individual invariant whether it is independent from the other invariants or not. Internally the model validator is started as many times as there are invariants in the model, and in each model validator run exactly one invariant is passed in logically negated form. As a variation of the already discussed `invIndep` option, `mv -invIndep <PropertyFile> <singleInvariant>` (without the keyword `all`) is available in order to construct the example for independence of the single invariant. If an invariant cannot be shown to be independent, further analysis can be performed by deactivating invariants with `constraints -flags <singleInvariant> +d`[4].

Concerning the example, the property file for the independence use case is the same as for the constraint implication use case. Below you see the protocol for calling the model validator with the independence option. You see that the invariants are indeed *not* independent. As detailed in the protocol and shown in Fig. 7, a further analysis with two checks, which deactivate one invariant, reveal that the invariant `parentOlderChild` is implying `acyclicParenthood`.

```
use> mv -invIndep invIndep.properties all
     InvIndepCheck:
       Person::acyclicParenthood: Not Independent
       Person::nameUnique: Independent
       Person::parentOlderChild: Independent
-----------------------------------------------
-- nameUnique => acyclicParenthood ?
-- parentOlderChild => acyclicParenthood ?
-----------------------------------------------
use> reset
use> constraints -flags Person::acyclicParenthood -d +n
                        Person::nameUnique        -d -n
                        Person::parentOlderChild  +d -n
use> mv -validate invIndep.properties
     ModelValidator: SATISFIABLE
-----------------------------------------------
use> reset
use> constraints -flags Person::acyclicParenthood -d +n
                        Person::nameUnique        +d -n
                        Person::parentOlderChild  -d -n
use> mv -validate invIndep.properties
     ModelValidator: UNSATISFIABLE
```

3.5 Solution Interval Exploration

There may be circumstances during validation in which the modeler is not only interested in a single solution in terms of a system state, but the modeler wants to obtain an overview on all solutions. Naturally this will be feasible only if the solution interval is relatively small. By choosing reasonable small bounds for classes and association and by restricting attribute values, interesting results can be achieved: “Even a small scope defines a huge space, and thus often suffices to find subtle bugs.” [Ja06, p. 16].

[4] On the USE command shell, deactivating and negating invariants can be combined.

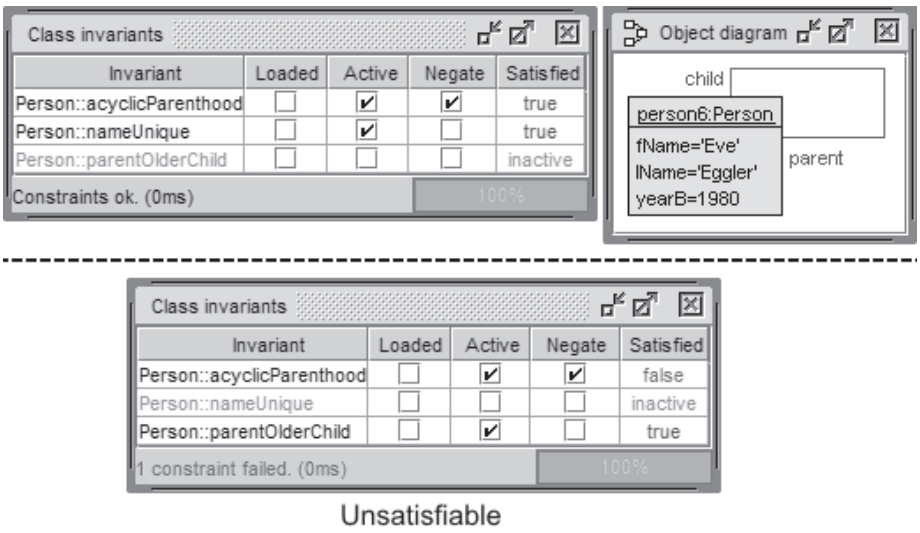

Fig. 7: Invariant status for independence and generated counterexample.

The technical option for the exploration of a solution interval is accessible in the model validator with the command `mv -scrollingAll <PropertyFile>` in combination with the additional succeeding commands `mv -scrollingAll [prev|next|show(<N>)]`. The first command computes all solutions with regard to the property file. The following commands allow to scroll through the solution interval or to access a solution with the respective solution number (referring to the order in which the solutions have been found).

In the example, the configuration file restricts the number of possible `Person` objects and names to three and the number of age values and parenthood links to two.

```
Person_min = 3; Person_max = 3;
Person_fName = Set{'Ada', 'Bob', 'Cyd'};
Person_lName = Set{'Alewife'};
Person_yearB = Set{1950, 1965};
Parenthood_min = 2; Parenthood_max = 2;
```

The protocol shows that the model validator finds six solutions which are displayed in Fig. 8. These object diagrams represent the complete search space, i.e., all allowed object diagrams of the running example, for the (admittedly and purposely) small configuration.

```
use> mv -scrollingAll scrollingAll.properties
     ModelTransformator: Translation time: 234 ms
     ModelValidator: SATISFIABLE
         Translation time: 1872 ms      Solving time: 187 ms
     ...
     ModelValidator: UNSATISFIABLE
         Translation time: 1622 ms      Solving time: 328 ms
     ModelValidator: Found 6 solutions
use> mv -scrollingAll show(1)   -- show(2) ...
     ModelValidator: Show solution 1
```

We repeat our warning remark with respect to large solution intervals described in the property file when employing the `scrollingAll` option: there may be many solutions; in the example, if the configuration offers one more year (e.g., in total the years `1950`, `1965`, `1980`), then the number of solutions grows from 6 to 36.

If it is too complex to explicitly construct the complete solution interval, one can approximate the interval by computing the next solution in a stepwise manner. The command

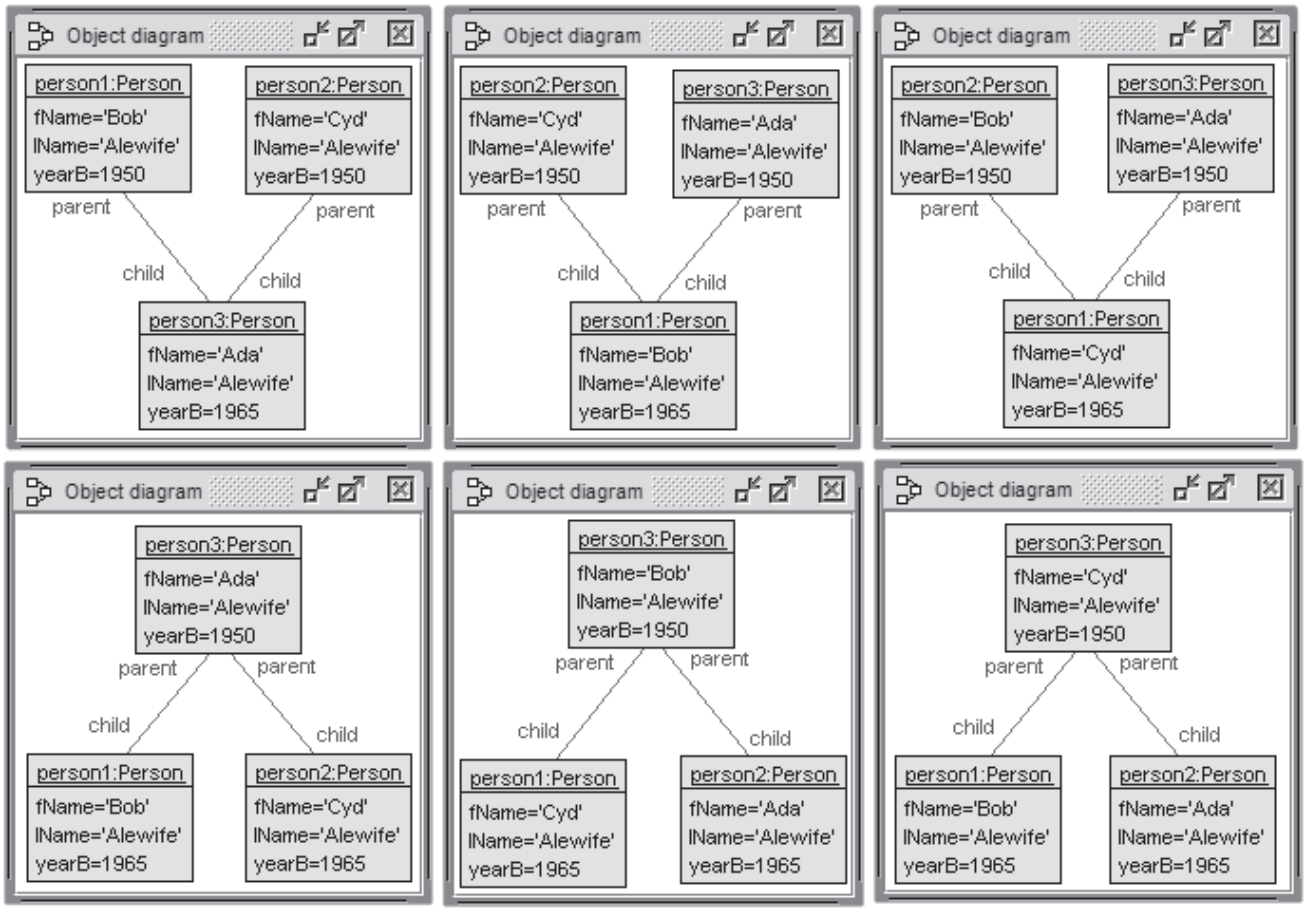

Fig. 8: Solution interval with 6 object diagrams.

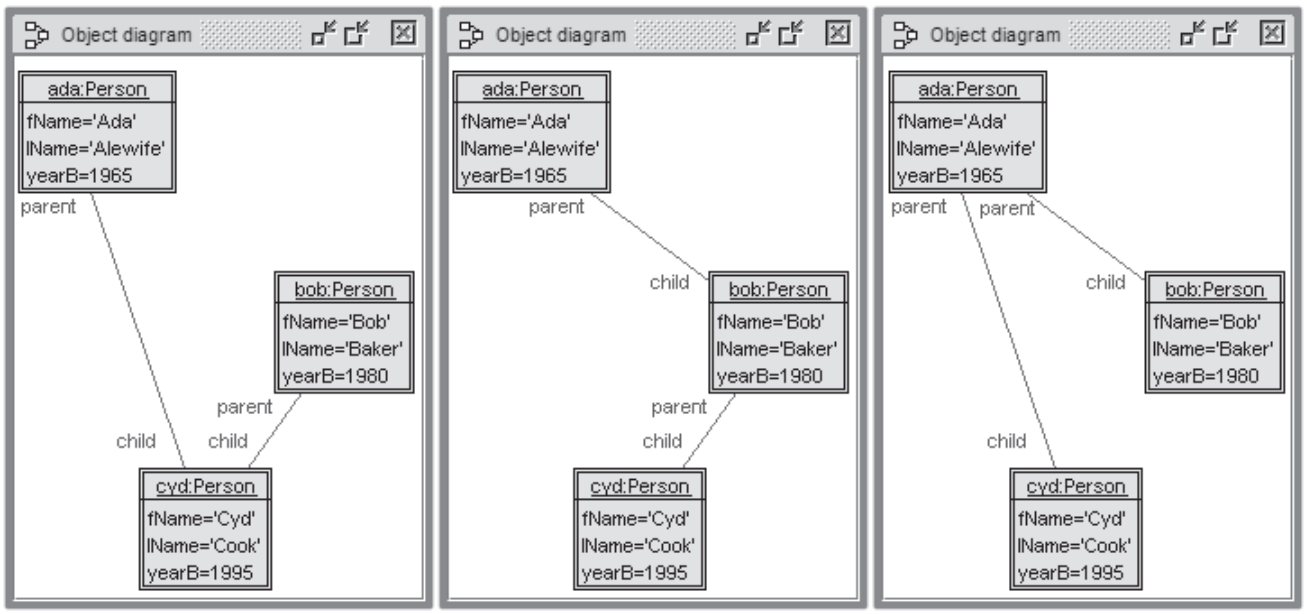

Fig. 9: Completions of partially specified solutions.

`mv -scrolling <PropertyFile>` finds a first solution, and following solutions can be obtained by `mv -scrolling next`.

3.6 Partial Solution Completion

The last option for a validation and verification task is the completion of a partially specified solution. When one has already constructed objects, attribute values and links (which taken together do not necessarily have to yield a valid system state), one may ask the model validator to complete such a partial system state to a valid solution. If a valid completion with regard to the configuration can be found, a valid system state containing the partially specified system state is constructed. If no valid completion can be found, this is reported to the modeler. An example is shown in Fig. 9.

In terms of the technical realization, the model validator must be explicitly directed to consider the already existing objects and links through specifying `mv -config objExtraction:=on` before the partial system state is asked to be completed. This option may be turned off later, if not needed any more.

The property file fixes the number of `Person` objects to three and the number of `Parenthood` links to two.

```
Person_min = 3; Person_max = 3;
Person_fName = Set{'Ada', 'Bob', 'Cyd', 'Dan', 'Eve'};
Person_lName =
  Set{'Alewife', 'Baker', 'Cook', 'Digger', 'Eggler'};
Person_yearB = Set{1905, 1920, 1935, 1950, 1965, 1980, 1995};
Parenthood_min = 2; Parenthood_max = 2;
```

The protocol file shows the construction of the three objects and fixes their attributes. The links however are not explicitly fixed, but are left as the central construction task for the model validator. The extraction of already existing objects together with their attributes is combined here with the `scrolling` option.

```
use> mv -config objExtraction:=on
     ModelValidatorConfiguration: Enable object extraction
use> !new Person('ada')
use> !set ada.fName := 'Ada'
use> !set ada.lName := 'Alewife'
use> !set ada.yearB := 1965
use> !new Person('bob')
use> ...                              -- bob, cyd analogously
use> mv -scrollingAll completion.properties
     ModelTransformator: Translation time: 202 ms
     ObjectDiagramModelEnricher: Extraction successful
     ModelValidator: SATISFIABLE
         Translation time: 62 ms      Solving time: 16 ms
     ...
     ModelValidator: UNSATISFIABLE
         Translation time: 16 ms      Solving time: 0 ms
     ModelValidator: Found 3 solutions
use> mv -scrollingAll show(1)      -- show(2) ...
```

Figure 9 reveals that three structurally different solutions are found by the model validator. In all three solutions the objects and their attribute values coincide.

Our techniques can also be employed for fault detection. If, for example, a new requirement would be that a person has to have two parents or no parents at all, then adding the constraint `Person.allInstances->exists(parent->size<>2 and parent->size<>0)` and employing the property reachability use case, would lead to a counterexample disproving the faulty assumption that the new requirement is already granted by the current model.

4 Related Work

The transformation of UML and OCL into formal specifications for validation and verification is a widely considered topic. The approach in [BKS02] presents a translation of UML and OCL into first-order predicate logic to reason about models utilizing theorem provers. Another, similar tool is *UML-RSDS* [LKR10], which allows for the validation of UML class diagrams.

Verification tools use such transformations to reason about models and verify test objectives. *UMLtoCSP* [CCR14] is able to automatically check correctness properties for UML class diagrams enhanced with OCL constraints based on Constraint Logic Programming. The approach operates on a bounded search space similar to the model validator. In [An10], *UML2Alloy* is presented. A transformation of UML and OCL into Alloy [Ja06] is used to be able to automatically test models for consistency with the help of the *Alloy Analizer*. Another approach based on Alloy is presented in [MJOR11]. In particular, limitations of the previous transformation are eliminated by introducing new Alloy constructs to allow for a transformation of more UML features, e.g., multiple inheritance. In [Wi08], OCL expressions are transformed into graph constraints and instance validation is performed by checking models against the graph constraints. Additionally, in [Ca10], a transformation of OCL pre- and postconditions is presented for graph transformations.

The work in [Br10] describes an approach for test generation based on a transformation of UML and OCL into higher-order logic (HOL). With the *HOL-TestGen* tool, test cases(model instances) are generated and validated. In [QT06], a transformation of UML and OCL into first-order logic is described and test methods for models are shown, e.g., *class liveliness* (consistency) and *integrity of invariants* (constraint independence). A different approach is presented in [CGR15]. The authors suggest to use Alloy for the early modeling phase of development due to its better suitability for validation and verification.

Finally, the USE model validator is to a certain degree the successor of the *ASSL* (A Snapshot Specification Language) [GBR05]. ASSL allows the specification of generation procedures for objects and links of each class and association. ASSL searches for a valid system state by iterating through all combinations defined by the procedures. In comparison, the USE model validator translates all model constraints into a SAT formula, which allows for a more efficient generation of a system state, due to detecting bad combinations earlier. Some of the use cases proposed here have been discussed employing ASSL in earlier work [GKH09]. However, the explicit options for formulating the use cases are new, and we employ a new underlying validation engine (Kodkod). In [GBR05] the use case

functionalities had to be explicitly formulated in the (programming-like language) ASSL. Now the use cases are basically formulated in terms of (descriptive) configurations.

The approaches mentioned above either already support a subset of the concepts as in the USE model validator or can be used to manually achieve results like *constraint independence* or *scrolling*. However, the degree of automation in the current approach is much higher. Without such a high level of automation, validation and verification is a cumbersome task: constraints have to be formulated manually, e.g., in the case of the scrolling use case, one constraint has to be added for every system state found to make sure a different state is generated next. Furthermore, the degree of UML and OCL concept coverage is typically lower in the mentioned approaches.

5 Conclusion and Future Work

In this paper, we have presented techniques to utilize a modern instance finder for a wide range of model validation and verification as well as fault detection methods in UML and OCL models. Examples are shown with the USE model validator using the six use cases: model consistency, property reachability, constraint implication, constraint independence, solution interval exploration, and partial solution completion. The techniques are useful from early development phases to explore models up to testing phases where model properties are verified. For example, the solution interval exploration has proven useful to present example instantiations of a model.

Future work should also concentrate on optimizing the verification tasks by providing help with determining bounds specifically for the presented techniques. Optimizations of the USE model validator itself includes support for more UML features and a more sophisticated handling of strings and large integers. Additionally, not all use cases have a high-level interface for the modeler to use. To make the use cases readily available for everyone, including non-experts, such high-level functions, like `mv -invIndep` for invariant independance, is desirable for all use cases. Finally, larger case studies have to further evaluate the individual methods presented.

References

[Al11] Ali, S.; Iqbal, M. Zohaib Z.; Arcuri, A.; Briand, L.: A Search-Based OCL Constraint Solver for Model-Based Test Data Generation. In (Núñez, M.; Hierons, R. M.; Merayo, M. G., eds): Proc. 11th Int. Conf. Quality Software QSIC. IEEE, pp. 41–50, 2011.

[An10] Anastasakis, K.; Bordbar, B.; Georg, G.; Ray, I.: On challenges of model transformation from UML to Alloy. Software and System Modeling, 9(1):69–86, 2010.

[BKS02] Beckert, B.; Keller, U.; Schmitt, P.: Translating the Object Constraint Language into first-order predicate logic. In: Proc. 2nd Verification WS: VERIFY. pp. 2–7, 2002.

[Bo89] Boehm, B.: Software Risk Management. In (Ghezzi, Carlo; McDermid, John A., eds): Proc. 2nd European Software Engineering Conf. (ESEC 1989). Springer, LNCS 387, pp. 1–19, 1989.

[Br10] Brucker, A.; Krieger, M.; Longuet, D.; Wolff, B.: A Specification-Based Test Case Generation Method for UML/OCL. In (Dingel, J.; Solberg, A., eds): Models in Software Engineering. Springer, LNCS 6627, pp. 334–348, 2010.

[Ca10] Cabot, J.; Clarisó, R.; Guerra, E.; de Lara, J.: Synthesis of OCL Pre-conditions for Graph Transformation Rules. In (Tratt, L.; Gogolla, M., eds): Int. Conf. Theory and Practice of Model Transformations. Springer, LNCS 6142, pp. 45–60, 2010.

[CCR14] Cabot, J.; Clarisó, R.; Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. Journal of Systems and Software, 93:1–23, 2014.

[CGR15] Cunha, A.; Garis, A. Gabriela; Riesco, D.: Translating between Alloy specifications and UML class diagrams annotated with OCL. SoSyM, 14(1):5–25, 2015.

[GBR05] Gogolla, M.; Bohling, J.; Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Software and System Modeling, 4(4):386–398, 2005.

[GBR07] Gogolla, M.; Büttner, F.; Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program., 69(1-3):27–34, 2007.

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

[Go15] Gogolla, M.; Hamann, L.; Hilken, F.; Sedlmeier, M.: Checking UML and OCL Model Consistency: An Experience Report on a Middle-Sized Case Study. In (Blanchette, J. et al., eds.): Tests and Proofs, TAP 2015. Springer, LNCS 9154, pp. 129–136, 2015.

[Ja06] Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press, 2006.

[KG12] Kuhlmann, M.; Gogolla, M.: From UML and OCL to Relational Logic and Back. In (France, R. B.; Kazmeier, J.; Breu, R.; Atkinson, C., eds): Model Driven Engineering Languages and Systems, MODELS 2012. Springer, LNCS 7590, pp. 415–431, 2012.

[LKR10] Lano, K.; Kolahdouz-Rahimi, S.: Specification and Verification of Model Transformations Using UML-RSDS. In (Méry, D.; Merz, S., eds): Integrated Formal Methods, IFM 2010. Springer, LNCS 6396, pp. 199–214, 2010.

[MJOR11] Maoz, S.; J.-O.Ringert; Rumpe, B.: CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In (Whittle, J.; Clark, T.; Kühne, T., eds): Model Driven Engineering Languages and Systems, MODELS 2011. Springer, LNCS 6981, pp. 592–607, 2011.

[OM13] OMG: Unified Modeling Language 2.5. Object Management Group (OMG), http://www.omg.com/uml/, 2013.

[QT06] Queralt, A.; Teniente, E.: Reasoning on UML Class Diagrams with OCL Constraints. In (Embley, D. W.; Olivé, A.; Ram, S., eds): Conceptual Modeling - ER 2006. Springer, LNCS 4215, pp. 497–512, 2006.

[TJ07] 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. Springer, LNCS 4424, pp. 632–647, 2007.

[Wi08] Winkelmann, J.; Taentzer, G.; Ehrig, K.; Küster, J. Malte: Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. ENTCS, 211:159–170, 2008.

[ZG03] Ziemann, P.; Gogolla, M.: Validating OCL Specifications with the USE Tool: An Example Based on the BART Case Study. ENTCS, 80:157–169, 2003.