Chapter 8
Quality Improvement of Conceptual UML and OCL Schemata through Model Validation and Verification

Martin Gogolla and Khanh-Hoang Doan

8.1 Introduction

Model-driven engineering (MDE) is a software development approach that puts emphasis on models and not on code. The main purpose of a model is abstraction. By abstracting system complexity through reduction of information, a model can catch the essentials of a system preserving properties relative to a given set of concerns [Selic(2006)]. MDE techniques are able to disregard details of different implementation dependent platforms, thereby allowing to concentrate on essentials characteristics that are valid for many platforms.

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. Finding correct and expressive models is important. Common model quality improvement techniques are model validation ("Are we building the right product?") and verification ("Are we building the product right?") [Boehm(1989)]. Among the different aspects of a system to be caught, structural aspects represented by class and object diagrams are of central concern.

The context of our work is the tool USE (UML-based Specification Environment), see [Gogolla et al(2007)] and [Gogolla and Hilken(2016)] that 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

Martin Gogolla
Database Systems Group, University of Bremen, e-mail: gogolla@cs.uni-bremen.de

Khanh-Hoang Doan
Database Systems Group, University of Bremen, e-mail: doankh@cs.uni-bremen.de

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 [Torlak and Jackson(2007)], 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.

In this paper, we discuss how to apply the tool USE in a larger example and demonstrate the advantages of our approach for relational database design. We start with a conceptual UML model in form of a UML class diagram and transform this conceptual schema into a relational database schema that is again represented as a UML class model. The typical constraints in the relational database model as primary and foreign key constraints are formulated as OCL invariants. We check properties of the resulting model with our tool USE.

The rest of the paper is structured as follows: Section 8.2 sketches our validation and verification use cases. Section 8.3 shows how the use cases can be applied in the context of a relational database schema and typical relational database constraints. Related work is discussed in Sect. 8.4. The paper ends with concluding remarks and future work in Sect. 8.5.

8.2 Validation and Verification Use Cases

Following [Gogolla and Hilken(2016)], Fig. 8.1 gives an overview on the options of our approach in form of a UML use case diagram. The central functionalities are shown as eight main use cases that are pictured in light gray whereas the remaining ones in white are subordinate use cases. All main use cases rely on a class model including accompanying OCL invariants and a configuration that fixes a finite search space for the population of classes, associations, attributes and datatypes. Let us go through the eight main use cases one by one and explain shortly their characteristics.

1. The use case ‘model consistency’ checks whether the model can be instantiated by at least one object diagram under the stated finite search space from the configuration. If this is possible, the consistency of the model has been shown.
2. The use case ‘property satisfiability’ tests whether a given additional OCL invariant, which can describe a more particular requirement on the model and which is added, can be satisfied with an object diagram as well.
3. In a similar way the use case ‘constraint implication’ is designed for determining whether an additional invariant is a logical consequence of the model. For achieving this, the additional invariant is loaded and then logically negated. If within the finite search space of the configuration an object digram is found, the logical consequence is not valid; if no object diagram is found, the logical consequence is valid in the finite search space.
4. The use case ‘constraint independence’ tests whether the stated OCL invariants are independent from each other, i.e., it will be checked whether each single

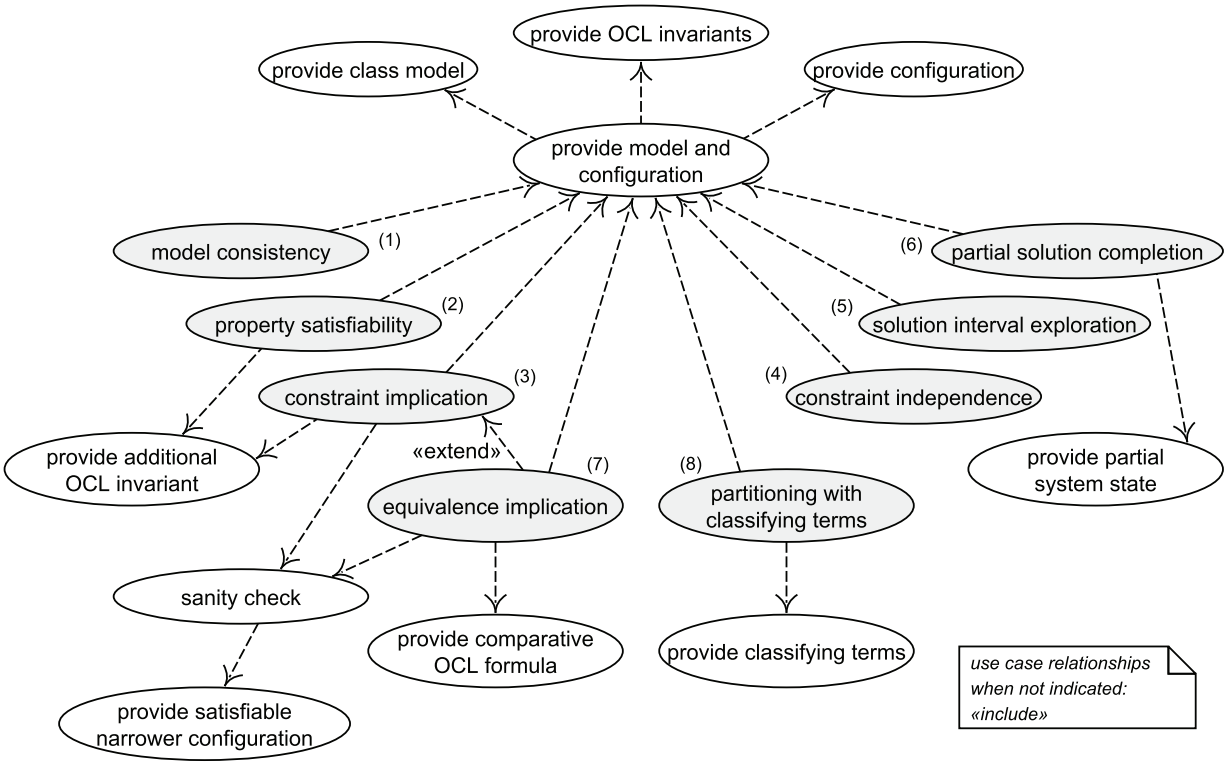

Fig. 8.1 Validation and verification use cases.

invariant is not a logical consequence from the remaining invariants. It is assured that no invariant can be removed without changing the model's induced set of object diagrams.

5. The use case 'solution interval exploration' is intended to be applied in situations where not only one single object diagram of the finite search space is of interest, but all solutions in form of object diagrams should be found. Even for smaller search spaces a comparison of different solutions can give interesting feedback.
6. The use case 'partial solution completion' assumes a partially described object diagram is present that might not yet satisfy model-inherent or explicit constraints; the task is then to find a completion in terms of objects, links and attribute values such that a valid object diagram satisfying all constraints is presented.
7. The use case 'equivalence implication' verifies for two OCL formulas `A` and `B` whether they are equivalent; the use case adds the logically negated invariant `(A implies B) and (B implies A)` and inspects whether that formula holds as a consequence, i.e., it checks that no object diagram exists in the search space and under the negated formula.
8. The last use case 'partitioning with classifying terms' allows to construct object diagram equivalence classes that are characterized by closed OCL query terms; in each equivalence class all OCL query terms evaluate to the same result; for each equivalence class a canonical representative in form of an object diagram is chosen; only a finite number of equivalence classes can be constructed.

Figure 8.2 shows the uses cases from Fig. 8.1 and the primary input and output artifacts. The input is in all use cases the class model, a configuration, (optionally a

variation of) the invariants and depending on the use case further input. The output is a single object model or a collection of object models.

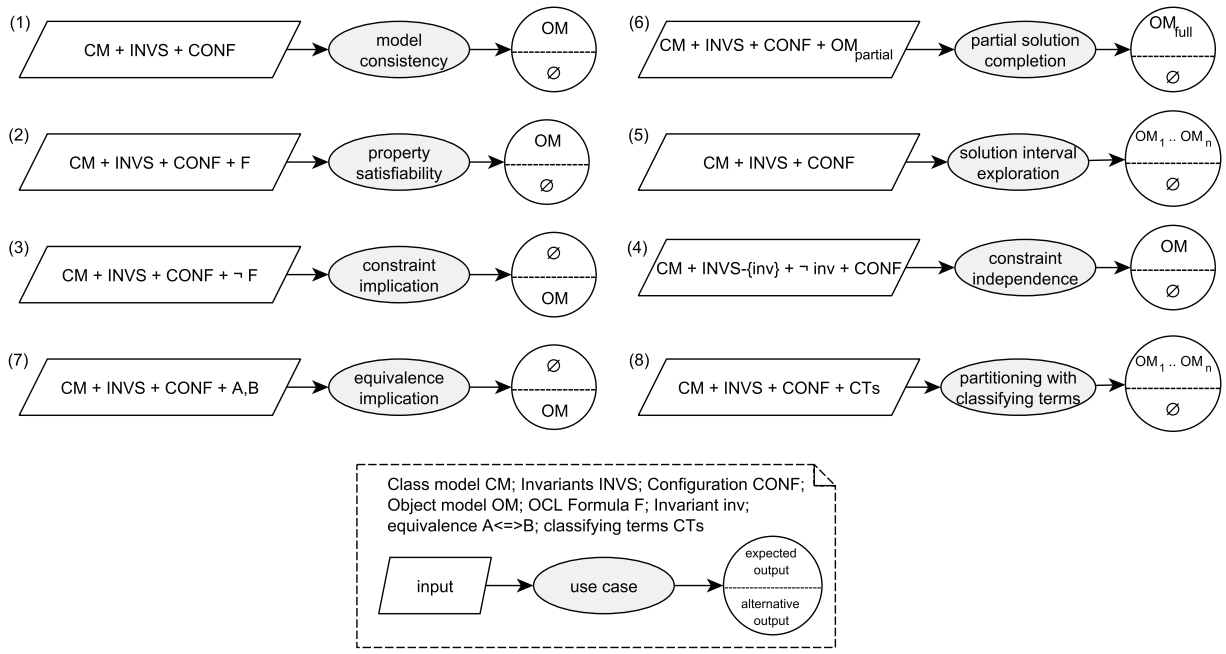

Fig. 8.2 Use case input and use case output for main and alternative flow.

8.3 Use Cases Applied in a Conceptual Modeling Example

This section explores four from the above eight model validation and verification use cases in a conceptual modeling example. The running example in this section discusses a relational database schema where a single table (relation) is modeled as a single UML class, primary and foreign key constraints are described as OCL constraints, and derived associations representing foreign keys from the relational database schema visualize the connection between the referencing tuple and the referenced tuple. Tuples from the relational database are represented as objects from the UML class diagram representing the relational database schema.

The UML class diagram in Fig. 8.3 is an example schema extracted from the book by Antoni Olivé on conceptual modeling [Olivé(2007)]. The example is an Order-OrderLine-Product world: an order line belongs to an order and refers to a product; products possess manufacturers and tax classes; products can have reviews written by customers who also trigger orders; products are classified by categories on which an ontology with parent categories and subcategories is provided.

The example uses associations of various kinds: many-to-many, functional (partial 0..1 and total 1..1), reflexive (binary association defined on a single class) and part-whole association. We use the term functional association to denote a many-to-one association as a source instance is functionally mapped to at most one or exactly one target instance.

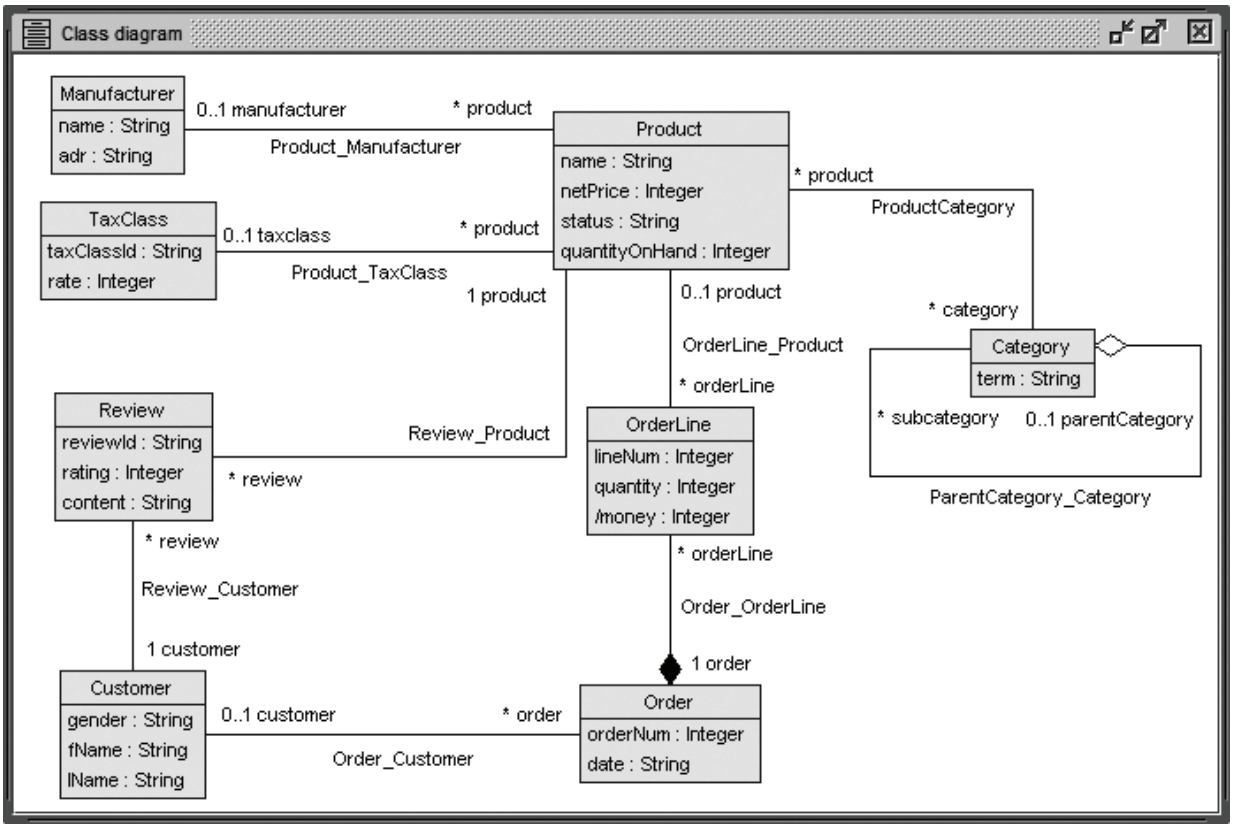

Fig. 8.3 UML example schema from Olive's original work.

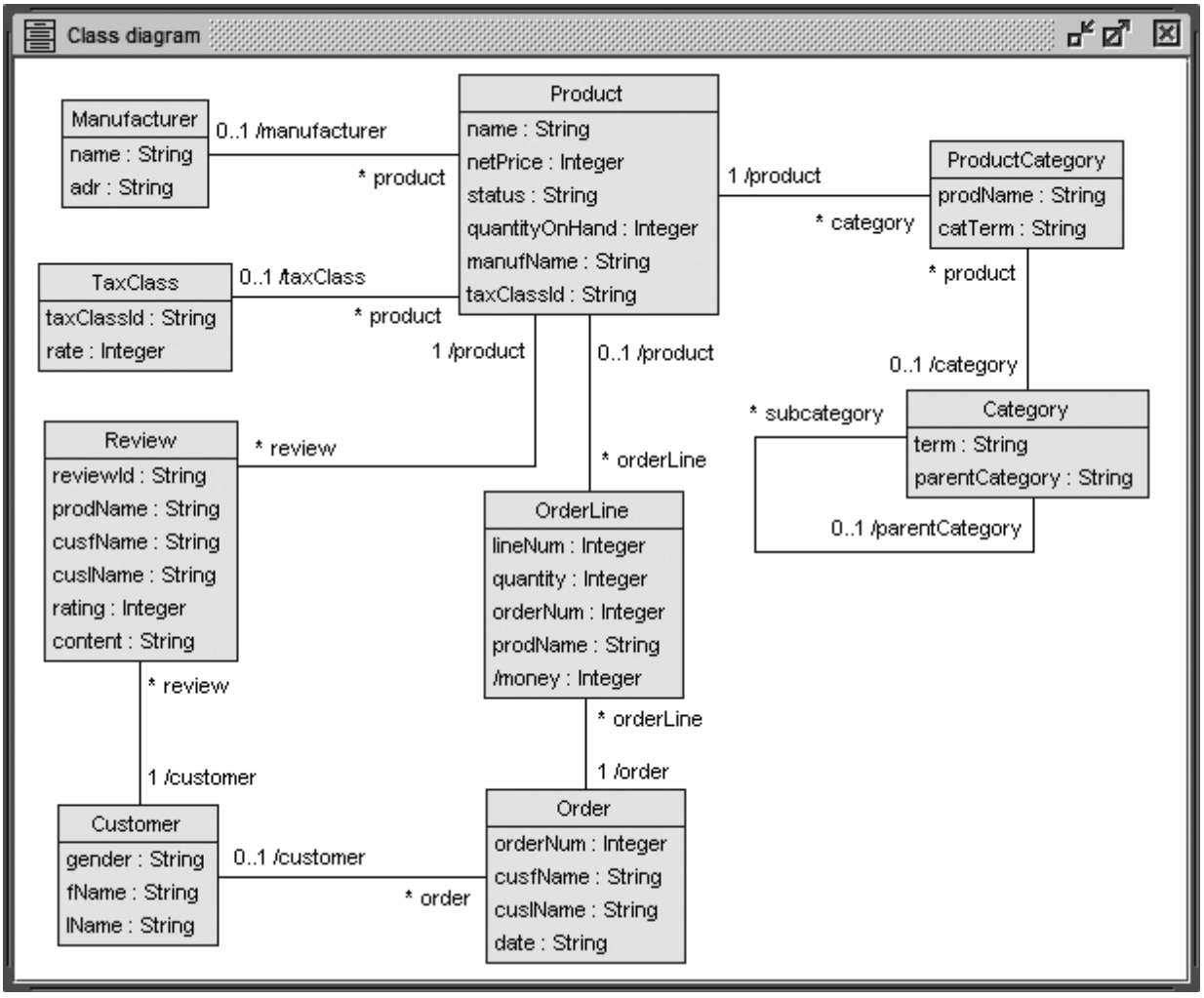

Fig. 8.4 Olive's example as a UML class diagram with classes for relations.

```
Manufacturer::namePK
Product::namePK
ProductCategory::productCategoryPK
TaxClass::taxClassIdPK
Review::reviewIdPK
Category::termPK
OrderLine::lineNumOrderNumPK
Customer::lNameFNamePK
Order::orderNumPK

Product::manufName_FK_Manufacturer
Product::taxClassId_FK_TaxClass
ProductCategory::catTerm_FK_Category
ProductCategory::prodName_FK_Product
Review::cuslName_cusfName_FK_Customer
Review::prodName_FK_Product
Category::parentCategory_FK_Category
OrderLine::orderNum_FK_Order
OrderLine::prodName_FK_Product
Order::cuslName_cusfName_FK_Customer

Product::statusValues
Category::acyclicSub
Category::subcategoryExists
Review::ratingValues
OrderLine::lineNum1exists
OrderLine::lineNumGE1
OrderLine::lineNumNoGaps
OrderLine::lineNumUniqueWithinOrder
Customer::genderValues
```

Fig. 8.5 Constraints defined in Olive's example.

In Fig. 8.4, we see how the UML conceptual schema is represented as a relational database schema in form of another UML class diagram. As a forward reference, one may look at Fig. 8.6 to see a simple object diagram illustrating the representation of a relational database state with tuples. The core of the transformation from the conceptual UML schema to the relational database schema can be characterized as follows: an entity is mapped to a relation that is represented as a class; a functional association is mapped to (a) an attribute (or many attributes) in the relation resp. the class corresponding to the source entity of the functional association and (b) a derived association for the foreign key; a general association (many-to-many) is mapped to (a) one relation represented as a class and (b) derived associations for the foreign keys.

The UML class diagram in Fig. 8.4 shows eight classes originating from entities: Manufacturer, Product, TaxClass, Review, OrderLine, Category, Customer, and Order; and the class diagram displays one class originating from an association: ProductCategory.

Fig. 8.5 shows the OCL constraints: each of the nine classes has a primary key constraint with a name ending in 'PK'. For each functional association (eight associations) and for each 'arm' of the other association (two 'arms') there is a foreign key constraint with a name containing 'FK' (ten foreign key constraints); the ten derived role names (indicated with the leading dash '/' in the name) are shown in the class diagram; there are nine other constraints, among them 'Category::acyclicSub' which requires the part-whole Category connections to form a directed, acyclic graph. It is a constraint involving the transitive closure. Standard SQL does not support to express this, but OCL due to the presence of the closure operation allows to describe the transitive closure.

A foreign key constraint establishes a connection between two relations. Because a table is represented in our UML model as a class, a corresponding OCL foreign key constraint must connect two classes. If the key of the referenced table consists of one attribute, there is one referencing attribute in the referencing table that points to one tuple in the referenced table. This is formally established with the OCL collection operation one(). We exemplarily show the requirement for the foreign key from OrderLine to Product.

```
context OrderLine inv prodName_FK_Product:
          Product.allInstances()→one(p|p.name=prodName)
```

Taking together the primary and foreign key constraints, all restrictions on the system states for the relational database have been expressed, and all necessary constraints are stated. In particular, the foreign key connection between the referencing tuple (represented as an object) and the referenced tuple (represented as an object) are manifested through the respective attribute values. Nothing more is needed. Thus only the objects with its values describe a database state. However, as UML and USE support derived associations, we can additionally visualize these connections also in formal terms through derived links. Each derived association is constructed by using a corresponding foreign key derivation term. The following definition shows exemplarily the foreign key derived association between the classes OrderLine and Product. The other derived associations and their roles are formulated analogously.

```
association FK_OrderLine_Product between
  OrderLine [0..*] role orderLine
  Product   [0..1] role product
    derived = Product.allInstances()→any(p|p.name=self.prodName)
end
```

If we compare the original UML schema in Fig. 8.3 and the corresponding relational database schema formulated as a UML class diagram with derived associations in Fig. 8.4, we see that the graph structures of both diagrams are nearly identical. An eye-catching difference is probably that the functional associations are not represented by an independent class, but these associations are integrated into the relation representing the source entity of the functional original association. These associations are present in the UML class diagram for the relational database schema through the referencing foreign key attributes and the derived role names.

The representation of foreign keys as derived associations seems to offer an intuitive way to represent the connections between tuples on the modeling level within a database state.

8.3.1 Model Consistency

As explained in Sec. 8.2, the purpose of the model consistency use case is to ensure that a valid system state (object diagram) can be instantiated, which ideally includes objects from all classes and links from all associations. To achieve this, we use the following configuration. Because we want to keep the generated object diagram in a reasonable size, we here use quite small numbers for the objects in the respective classes. The configuration will also provide finite, concrete sets for attributes values, e.g., `Product` names like `'Apple'` or `'Banana'`.

```
Manufacturer_min = 1        Manufacturer_max = 1
Review_min = 1              Review_max = 1
TaxClass_min = 1            TaxClass_max = 1
Category_min = 2            Category_max = 2
Product_min = 2             Product_max = 2
ProductCategory_min = 1     ProductCategory_max = 1
Order_min = 1               Order_max = 1
OrderLine_min = 4           OrderLine_max = 4
Customer_min = 1            Customer_max = 1
```

Fig. 8.6 shows the generated object diagram when we execute the model validator with the above configuration. As can be seen, the object diagram shows objects being instantiated from all nine classes and links originating from all ten foreign key derived associations. We emphasize the fact that, during the construction process, the model validator must take into account the nine classes and the 28 non-trivial invariants defined in the model.

8.3.2 Property Satisfiability

Basically, checking property satisfiability is finding the answer to the question whether a scenario, which is defined by an additional OCL formula, exists or does not exist. If we provide a sufficient finite search space (via a configuration), the model validator will give the answer (1) as a object diagram, in which the given property is satisfied, or (2) by answering that a valid scenario cannot be constructed within the given finite search space.

In this example, we want to check the property 'Is it possible to build a scenario where the Category objects build a tree?'. The property is formulated as the following invariant.

```
context Category inv categroryTree:
  Category.allInstances→one(c | c.parentCategory='') and
```

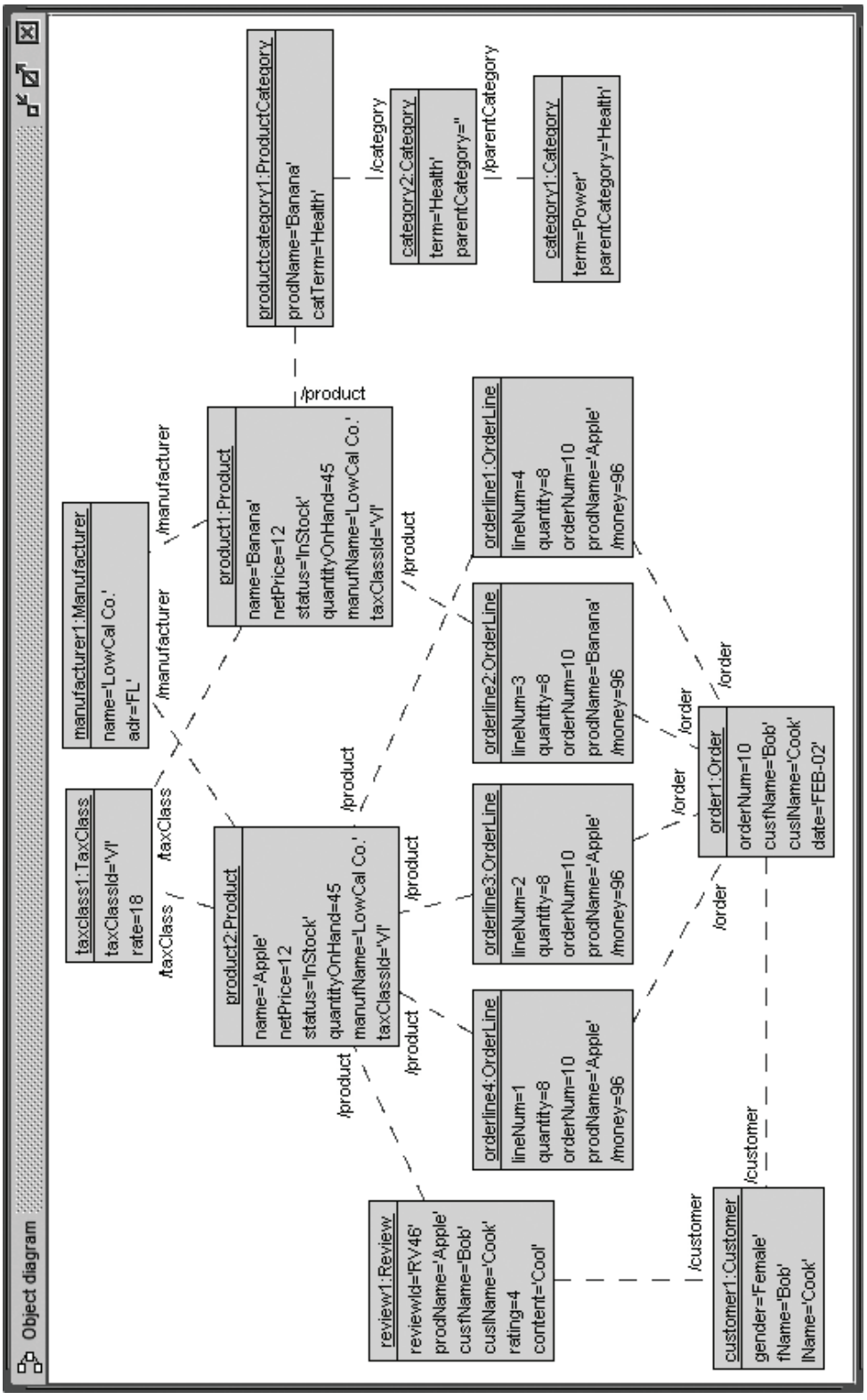

Fig. 8.6 Valid object diagram for model consistency use case.

```
Category.allInstances→exists(c | c.subcategory→size>=2)
```

Executing the model validator after the model has been enriched with the property invariant, we receive a satisfying scenario as shown in Fig. 8.7. We here use a slightly modified configuration as in the model consistency use case.

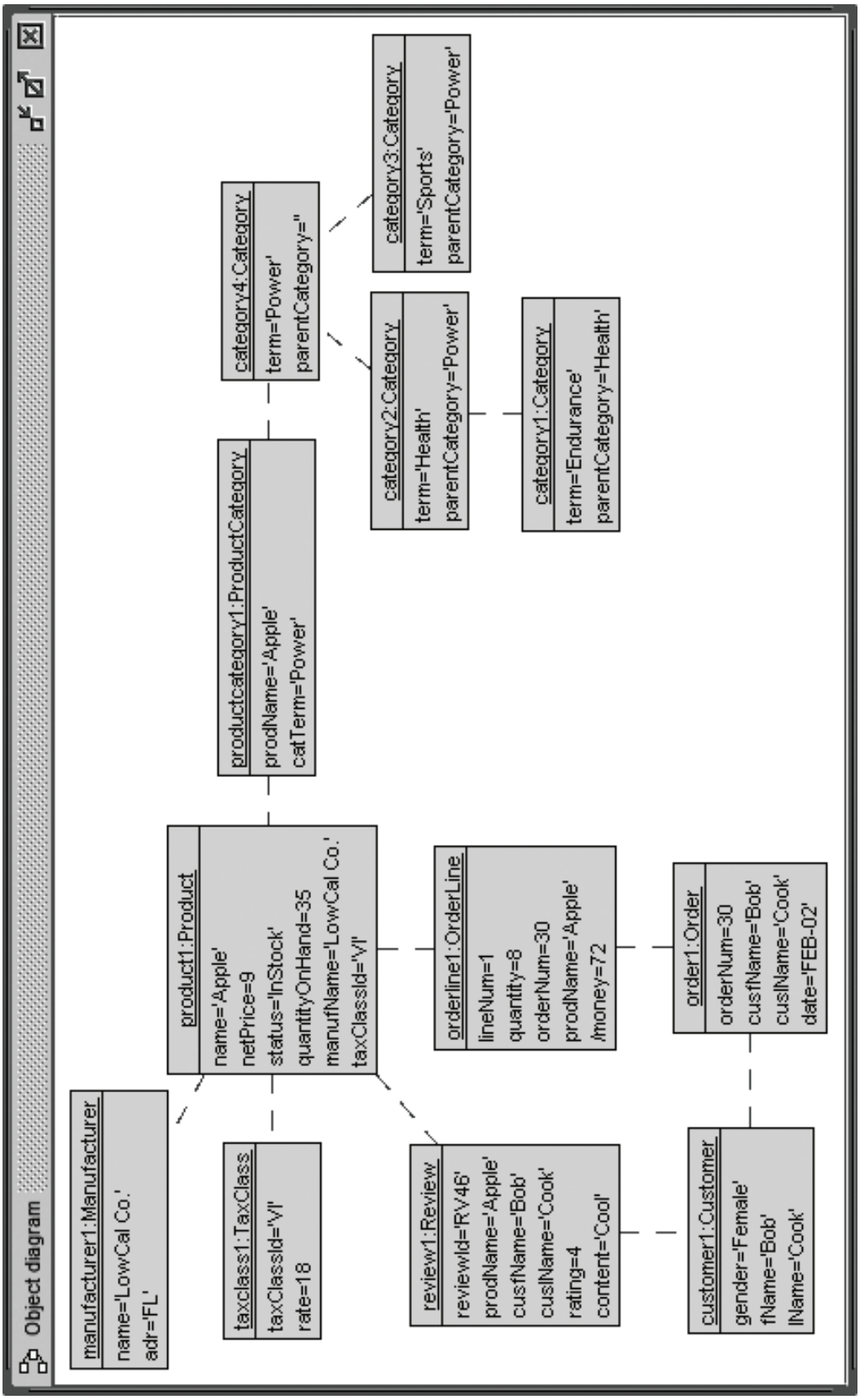

Fig. 8.7 Generated object diagram for property satisfiability use case.

8.3.3 Constraint Independence

As mentioned before, the constraint independence use case realizes a process that checks whether an invariant is independent from the other invariants. We prototyp-

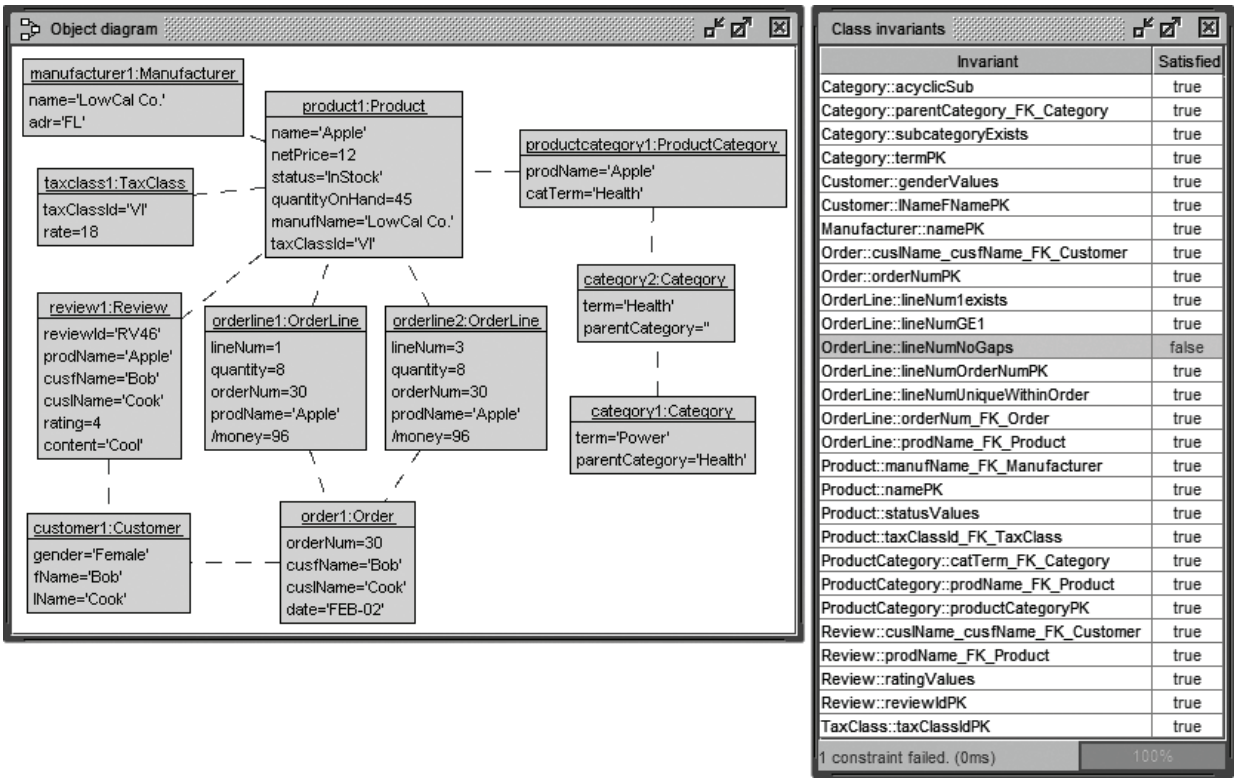

Fig. 8.8 Generated object diagram constraint independence use case.

ically select the following invariant that asserts that within an order the order lines are consecutively numbered.

```
context OrderLine inv lineNumNoGaps: lineNum>1 implies
  order.orderLine→exists(ol | ol.lineNum=lineNum-1)
```

In order to check constraint independence, we only do minor changes in the previous configuration. But we have to negate the selected invariant. As shown in Fig. 8.8, the model validator will construct an object diagram where two order lines are present that do not have consecutive numbers. Except of the selected invariant, which is evaluated to false, all other invariants evaluate to true. Thus the selected invariant is indeed independent. It is essentially needed in the model, otherwise unwanted system states like the one in Fig. 8.8 might occur.

8.3.4 Partial Solution Completion

In Fig. 8.9 you see an example for applying the use case 'partial solution completion' in our running relational database schema model.

The upper object diagram and class invariant evaluation picture the starting situation with a partial object diagram and six failing OCL invariants. Among the failing invariants are the Review primary key constraint and the foreign key constraint from Review to Customer. In this use case, the model validator modifies undefined attributes to defined ones, and through this, links for the derived foreign key association between Review and Customer can be established. The lower part of the figure

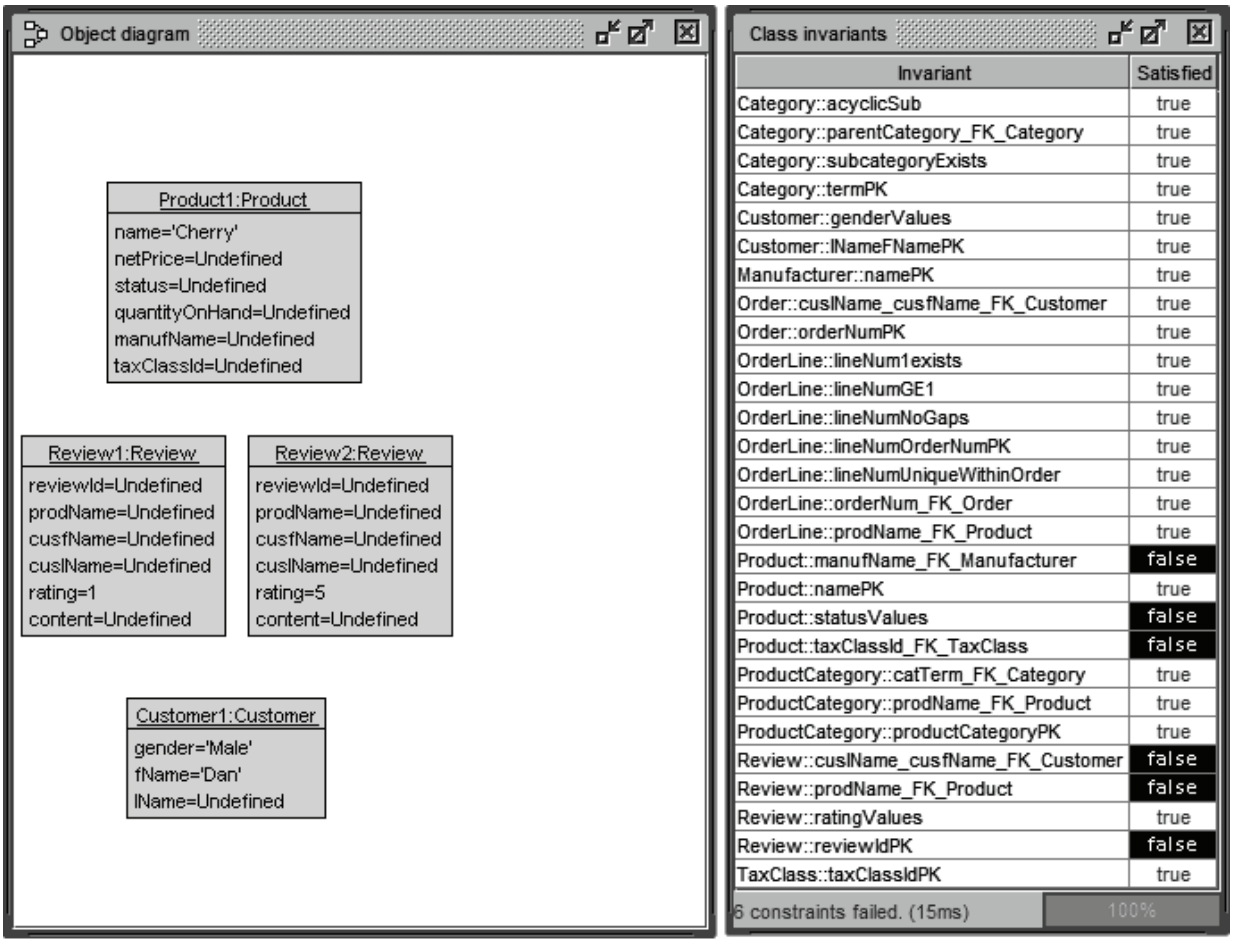

Fig. 8.9 Completion of partial object diagram by the USE model validator.

shows the enriched object diagram and the invariant evaluation which proves that after the completion all invariants are valid.

The example nicely demonstrates that proper attribute values have to be available in the configuration. Although the object diagram is quite small, the example illustrates well the use case 'partial solution completion' that is employed here in order to adjust an invalid system state to a correct one.

8.4 Related Work

In recent years, a number of interesting verification and validation approaches of conceptual schemata have been introduced. In [Queralt and Teniente(2012)], [Queralt et al(2012)], [Farré et al(2013)] a list of major properties, e.g., satisfiability, class liveliness, nonredundancy, consistency, can be verified using different techniques. [Queralt et al(2012)] introduces a fragment of OCL, the so-called OCL-Lite, encoding it in description logic. [Farré et al(2013)] focusses on the automated reasoning on UML schemata containing arbitrary constraints, derived roles, derived attributes and queries after translating the UML/OCL schema into a first order logic formalisation. The above approaches all translate UML and OCL into logic before reasoning about the conceptual model. The transformation of UML and OCL into formal specifications for validation and verification purposes is a widely considered topic, as it is presented in the following papers. In [Snook et al(2010)], a translation from UML to UML-B is presented und used for the validation and verification of models, focusing on consistency and checking safety properties. The approach in [Beckert et al(2002)] presents a translation of UML and OCL into first-order predicate logic to reason about models utilizing theorem provers.

Furthermore, incremental checking of OCL constraints is a popular technique for ensuring the quality of a UML schema. A runtime checking approach for the satisfiability of all constraints after changes in the system state is presented in [Cabot and Teniente(2009)]. Similarly, [Oriol and Teniente(2015)] introduces a method to verify OCL constraint violations by checking the emptiness of SQL queries, which are automatically obtained from the OCL constraint.

After checking the problems in a UML schema through verification and validation techniques, automatic fixing and repairing will help to improve the conceptual schema. The approach in [Oriol et al(2014)] can detect non-executable operations in a UML/OCL conceptual schema and automatically correct the missing effects in the postconditions of the operations. Another approach, which is presented in [Oriol et al(2015)], automatically computes the additional changes needed to keep the UML/OCL schema consistent, i.e., all constraints are satisfied, when a set of update events is applied to the system state of the schema.

Testing a conceptual schema is also a research direction for quality improvement. [Granda(2013)] introduces a solution for automatically generating test cases from a UML/OCL model. This approach integrates several existing languages and tools, i.e., the USE tool and the CSTL language. The CSTL language is used as a language

for writing automated tests in the approaches presented in [Tort and Olivé(2010)] and [Tort et al(2012)].

8.5 Conclusion and Future Work

In this paper, we have presented techniques to utilize an up-to-date modeling tool for a wide range of model validation and verification tasks. Examples are shown with the USE model validator using four use cases: model consistency, property reachability, constraint independence, and partial solution completion.

Future work could 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. In order to offer support for relational database design, we plan to import SQL database schemata, represent them as UML and OCL models and generate (positive and negative) test database states with the model validator (exported then again as SQL scripts). Finally, larger verification and validation case studies have to further evaluate the individual methods presented.

References

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

[Boehm(1989)] Boehm B (1989) Software risk management. In: Ghezzi C, McDermid JA (eds) Proc. 2nd European Software Engineering Conf. (ESEC 1989), Springer, LNCS, Vol 387, pp 1–19

[Cabot and Teniente(2009)] Cabot J, Teniente E (2009) Incremental integrity checking of UML/OCL conceptual schemas. Journal of Systems and Software 82(9):1459–1478

[Farré et al(2013)] Farré C, Queralt A, Rull G, Teniente E, Urpí T (2013) Automated reasoning on UML conceptual schemas with derived information and queries. Information & Software Technology 55(9):1529–1550

[Gogolla and Hilken(2016)] Gogolla M, Hilken F (2016) Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool. In: Oberweis A, Reussner R (eds) Proc. Modellierung (MODELLIERUNG'2016), GI, LNI 254, pp 203–218

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

[Granda(2013)] Granda MF (2013) Testing-based conceptual schema validation in a model-driven environment. In: Proc. Doctoral Consortium 25th Int. Conf. Advanced Information Systems Engineering (CAiSE 2013)

[Olivé(2007)] Olivé A (2007) Conceptual modeling of information systems. Springer

[Oriol and Teniente(2015)] Oriol X, Teniente E (2015) Incremental checking of OCL constraints with aggregates through SQL. In: Conceptual Modeling - 34th Int. Conf., ER 2015, Proceedings, pp 199–213

[Oriol et al(2014)] Oriol X, Teniente E, Tort A (2014) Fixing up non-executable operations in UML/OCL conceptual schemas. In: Conceptual Modeling - 33rd Int. Conf., ER 2014. Proceedings, pp 232–245

[Oriol et al(2015)] Oriol X, Teniente E, Tort A (2015) Computing repairs for constraint violations in UML/OCL conceptual schemas. Data Knowl Eng 99:39–58

[Queralt and Teniente(2012)] Queralt A, Teniente E (2012) Verification and validation of UML conceptual schemas with OCL constraints. ACM Trans Softw Eng Methodol 21(2):13:1–13:41

[Queralt et al(2012)] Queralt A, Artale A, Calvanese D, Teniente E (2012) OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. Data Knowl Eng 73:1–22

[Selic(2006)] Selic B (2006) UML2: A Model-Driven Development Tool. IBM Systems Journal 45(3):607–620

[Snook et al(2010)] Snook C, Savicks V, Butler M (2010) Verification of UML Models by Translation to UML-B. In: Aichernig B, de Boer F, Bonsangue M (eds) Formal Methods for Components and Objects, FMCO 2010, Springer, LNCS, Vol 6957, pp 251–266

[Torlak and Jackson(2007)] Torlak E, Jackson D (2007) 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, Vol 4424, pp 632–647

[Tort and Olivé(2010)] Tort A, Olivé A (2010) An approach to testing conceptual schemas. Data Knowl Eng 69(6):598–618

[Tort et al(2012)] Tort A, Olivé A, Sancho M (2012) On checking executable conceptual schema validity by testing. In: Database and Expert Systems Applications - 23rd Int. Conf., DEXA, 2012. Proceedings, Part I, pp 249–264