Achieving Model Quality through Model Validation, Verification and Exploration

Martin Gogolla*, Frank Hilken, Khanh-Hoang Doan

Database Systems Group, University of Bremen, Bremen, Germany

Abstract

System development strategies, like model-driven engineering (MDE), help to abstract architectures and provide a promising way to deal with architecture complexity and design quality. 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. We here introduce and structure essential use cases for model validation, verification and exploration that help developers find faults in model descriptions and thus enhance model quality. Along with the use cases, we demonstrate 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.

Keywords: UML and OCL model, constraint, invariant, model validation, model verification, model quality.

1. Introduction

Model-driven engineering (MDE) is a software development paradigm that in the first place focuses on models and not on code. One purpose of a model is abstraction. A model can catch a system by abstracting its complexity through reduction of information, however preserving properties relative to a given set of concerns [1]. One advantage of MDE may be seen in the fact that a model is able to disregard details of different implementation dependent platforms, thereby allowing to concentrate on essentials characteristics that are valid for many platforms.

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. Finding correct and expressive models is important. Common model quality improvement techniques are model validation ("Are we building the right product?") and model verification ("Are we building the product right?") [2]. Among

*Corresponding author

Email addresses: gogolla@cs.uni-bremen.de (Martin Gogolla), fhilken@cs.uni-bremen.de (Frank Hilken), doankh@cs.uni-bremen.de (Khanh-Hoang Doan)

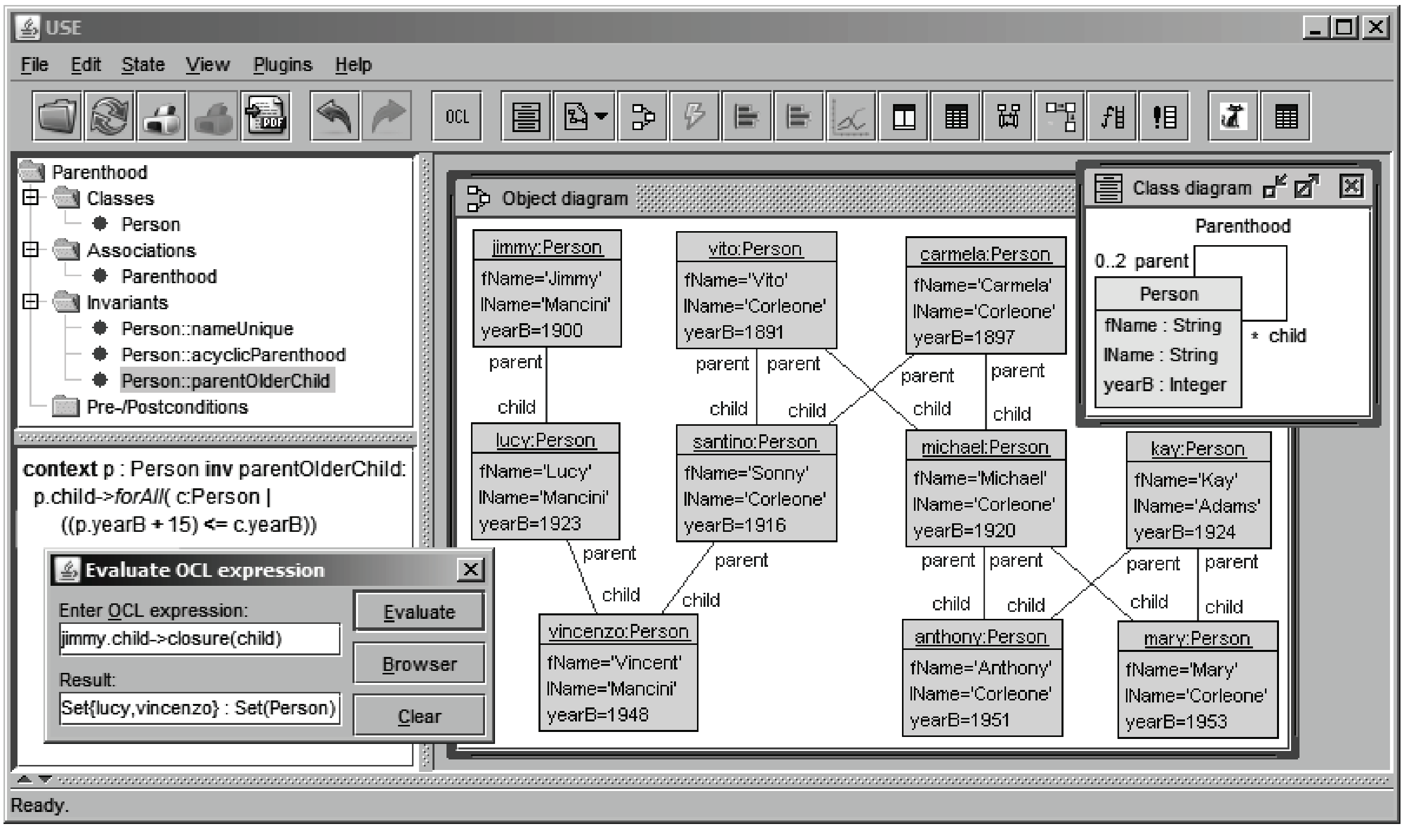
the different aspects of a system to be caught, structural aspects represented by class and object diagrams are of central concern.

Over the last years, a number of modeling approaches and tools enhancing model quality have been put forward, among them the proposals in [3, 4, 5, 6]. More details will be discussed in the related work section. The context of our work is the tool USE (UML-based Specification Environment) [7] that supports the development of UML models enhanced by OCL constraints. USE can work with 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 [8], 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.

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 satisfiability, constraint implication, constraint independence, solution interval exploration, partial solution completion, equivalence implication and partitioning with classifying terms. 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 *partitioning with classifying terms* means that not only a single object diagram in form of objects and links is examined but all object diagrams are taken into account and may be inspected for validation and verification, on the basis of OCL query terms (classifying terms) that determine the properties of the object diagrams [9]. The work in this paper extends the results from the work in [10] by additional use cases, presentation of further details for the already proposed use cases, and an application of our approach within a large example.

The running example in the first part of the paper is rather small, but it allows us to demonstrate the essentials of our approach in a condensed way. We have already checked that our use cases work for larger models as well [11]. In the second part of the paper, a large example demonstrates the advantages of our approach for relational database design. All presented use cases do not only benefit the USE tool, but also other verification engines [3, 4, 5, 6] 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 introduces the running example for the first part of the paper and sketches basic notions and methods. Section 3 discusses in detail our validation and verification use cases by means of a small example. Section 4 shows that the use cases can be applied in the context of a larger example describing relational database schemas and typical relational database constraints. Related work is discussed in Sect. 5. The paper ends with concluding remarks and future work in Sect. 6.



```
context p : Person inv parentOlderChild:
  p.child->forAll( c:Person |
    ((p.yearB + 15) <= c.yearB))
```

Enter OCL expression: jimmy.child->closure(child)

Result: Set{lucy,vincenzo} : Set(Person)

Figure 1: Parenthood Example class and object diagram.

2. Preliminaries

2.1. 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 (or *class model*) in the top right contains one class `Person` having a *first name*, a *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 (here, children and children of children) 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.

The USE tool allows developers to simulate given class diagrams by building or generating system states, evaluating expressions and change the behavior of OCL invariants on-the-fly. Using the USE specific language SOIL (Simple OCL-based Imperative Language), it is possible to manipulate system states using scripts in an imperative way. Furthermore, the tool implements features to dynamically load additional invariants to use for verification tasks. Finally, all invariants can be disabled to be ignored temporarily or set to be negated in order to achieve an inverse effect of the constraint, using the so-called *invariant flags*. USE is an interactive tool which allows developers to operate it using its GUI and shell. Most actions are possible with either interface, however the execution of the use cases in this paper will mostly be presented in the form of their respective shell commands.

2.2. Relational Logic and Kodkod

The verification engine used in this paper, the USE model validator, is based on the relational logic of Kodkod [8]. Kodkod defines a problem to consist of a universe, i.e., a set of uninterpretable atoms, a set of relation declarations and a relational formula. The universe is defined by the underlying class diagram together with a 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. For classes there has to be a mandatory upper bound, limiting the overall number of objects to a finite population. The lower bound is zero by default, but can be increased to specify a certain minimum number of objects required. Analogously, upper and lower bounds for links from an association may be stated. However, the upper bound may be left open in which case it is calculated from the possible connections between the limited number of objects. These configurations are also referred to as *bounds* as in the boundaries of the model search space, i.e., all possible instantiations of the model with the specified number of objects and links. 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 [12].

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. We call this *SAT*, which is short for the fact that the solver deemed the logical formula satisfiable by finding a valid assignment for all variables. Otherwise, no instance exists for the given model within the search space defined by the configuration. This situation is called *UNSAT*, which is short for unsatisfiable. In the case of an UNSAT result, it is very difficult to decide, why no instance could be found. As mentioned before, the formula given to the SAT solver is built from many elements translated from the model. Thus the problem can be, for example, an inconsistent model or a bad configuration.

Therefore, if the developer wants to draw conclusions from an UNSAT result, which will be required for certain use cases, the reason for the UNSAT must always be explicitly investigated, and the responsible, 'guilty' element (like the class model with model-inherent constraints, explicit invariant, loaded invariant, invariant flag, configuration) must be identified.

There exist special SAT solvers – so called provers – that give additional information in case of an UNSAT. Namely, the result is enriched with a set of constraints that lead to a contradiction in the formula. This can greatly help to analyze the reason for an UNSAT. For example, when the set only contains two OCL invariants, one can conclude that these invariants lead to a model inconsistency. However, the provers also have considerable downsides. Tracing the information necessary for the feedback requires more resources and impacts the performance, which leads to longer solving times. In addition, the feedback from Kodkod is in the form of relational logic formulas and has to be further translated back into UML and OCL constraints to make them be understood by a wider audience, which has not been done yet. With these severe drawbacks, this method is not well established, and we will be presenting another means to get more confidence from UNSAT solver results.

2.3. Broad Versus Narrow Bound Configurations

When working with bounded model checkers, such as the USE model validator, it is very important to choose sensitive bounds for each verification task. The bound configurations are a big factor in determining the performance of the verification process, i.e., the less possible variable assignments exist, the faster it is to try all of them. In [13] the notion of *bound tightening* is introduced, a procedure to programmatically reduce the intervals of bounds without violating model consistency using CSP. The process takes into account the factors of the model that put relationships between the number of model elements, e.g., multiplicities, attribute access, OCL `allInstances` and existential quantifiers, to name a few. For example, instead of defining lower and upper bounds of a class as $\langle 0, 8 \rangle$ allowing between 0 and 8 objects of this class, the lower and upper bounds $\langle 2, 6 \rangle$ would result in a smaller search space, due to fewer possibilities in total. We refer to these bounds as the *broad* configuration having larger intervals and the *narrow* configuration with tighter bound intervals. In most cases, the narrow bound configurations are preferred due to the positive impact on performance. This includes both outcomes of the solver: if there is no solution (UNSAT), the complete search space must be processed for all combinations of assignments, where a reduced search space has a great impact; if there exists a solution (SAT), the bound tightening has eliminated unreasonable situations that are not needed to be tried by the SAT solver anymore. However, considering bounded model checking, in the case of an UNSAT result, one has only proved that a certain property does not exist within the stated bounds. Therefore, in some cases, where one aims to show the absence of a property, a broader bound configuration can be of advantage.

If one considers a broad and a narrow configuration, i.e., in the narrow configuration all bounds are strictly more restricted or equal to the bounds in the broad configuration, and one shows SAT for the narrow configuration, then this means that also the broad configuration leads to SAT. The advantage of a narrow configuration might be that it shows a more interesting solution, for example, a solution with links whereas the broad configuration did not make any restriction on the links and could allow solutions without any links. Compare, for example, the two (partial) configurations in Table 1. The broad configuration allows for arbitrary many `Parenthood` links (-1 is equivalent to the unlimited upper bound `*`), whereas the narrow configuration requires at least one link to be present, making solutions that do not have links of the association invalid with this configuration.

Table 1: Simple example for broad and narrow configurations.

Configurations	broad	narrow
Person_min	0	2
Person_max	3	3
Parenthood_min	0	1
Parenthood_max	-1	4

Multiple configurations can be stored in one file, using names to distinguish which to choose. In the minimal example in Table 1, there are two configurations: *broad* in the middle column and *narrow* in the right column.

3. Validation and Verification Use Cases

Figure 2 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.

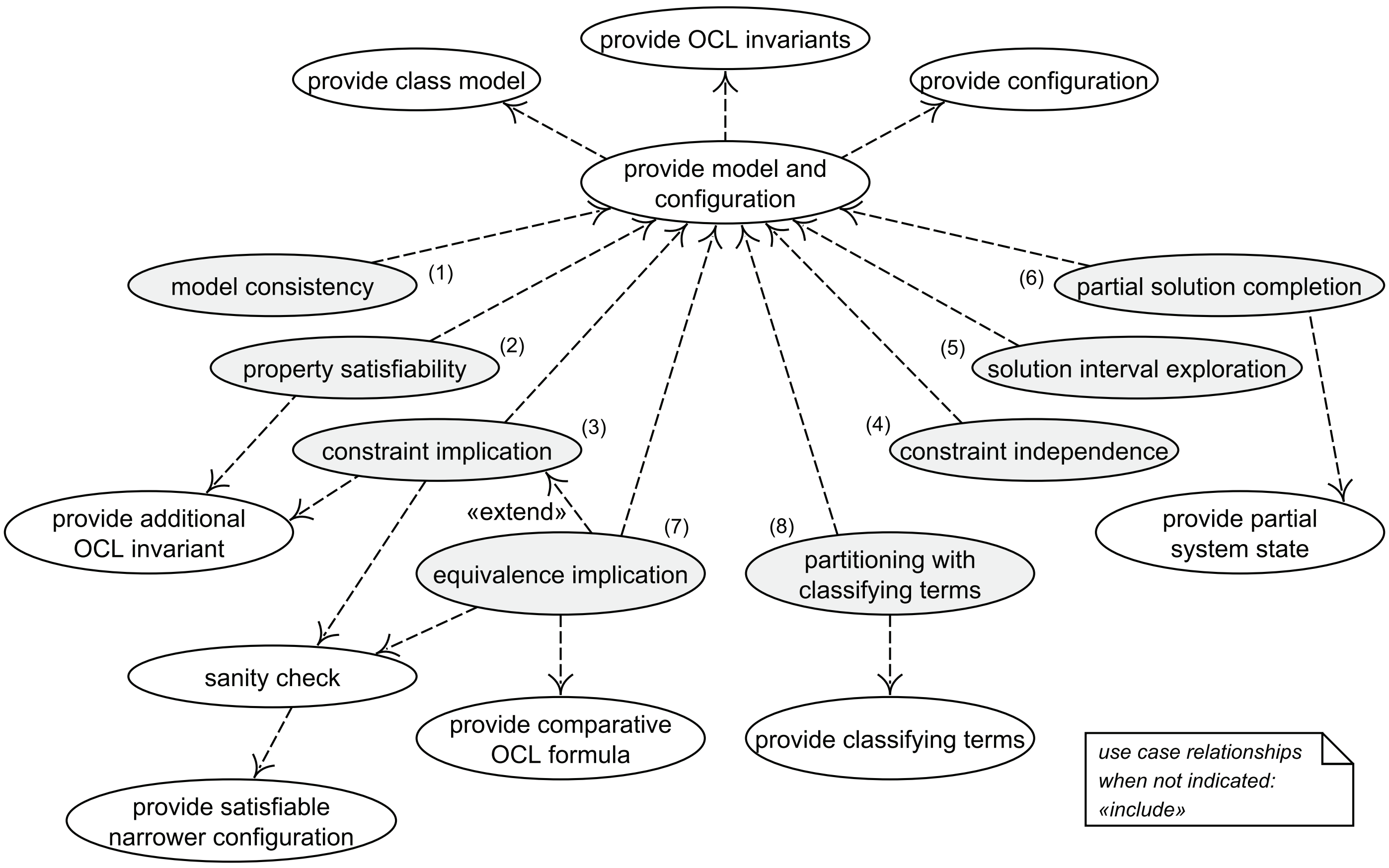

provide OCL invariants
provide class model
provide configuration
provide model and configuration
model consistency
(1)
(6)
partial solution completion
property satisfiability
(2)
(5)
solution interval exploration
constraint implication
(3)
(4)
constraint independence
provide additional OCL invariant
«extend»
(7)
(8)
provide partial system state
equivalence implication
partitioning with classifying terms
sanity check
provide comparative OCL formula
provide classifying terms
use case relationships when not indicated: «include»
provide satisfiable narrower configuration

Figure 2: Validation and verification use cases.

(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, i.e., the class model including the model-inherent multiplicity and whole-part constraints and the explicit OCL invariants are not contradictory.

(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, because the found object diagram constitutes a counter example; if no object diagram is found, the logical consequence is valid in the finite search space and

our approach assumes that the logical consequence is generally valid. As the answer relies on the fact that no object diagram is found, i.e., on an unsatisfiability answer, the subordinate use case 'sanity check' asserts that the reason for the unsatisfiability is the logical negation and not the original model and not the finite search space from the configuration. The following two use cases (7) and (8) in Fig. 2 will be discussed a bit later.

(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 invariant is not a logical consequence from the remaining invariants. In this use case a minimality property of the invariant set is assured, in the sense that in the successful case 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 remaining last two use cases are new with respect to [10]. 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; as the result again relies on unsatisfiability, the subordinate use case 'sanity check' assures that only the logical negation is responsible for the result, not the model and not the search space.

(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 [9].

All uses cases will be explained in detail with illustrating examples below.

Figure 3 shows the uses cases from Fig. 2 and the primary input and output artifacts. In all use cases the input is: 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. The distinction between expected and alternative 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 case the expected output is that no object model is found, whereas a found object model represents a counter example for the suspected constraint implication.

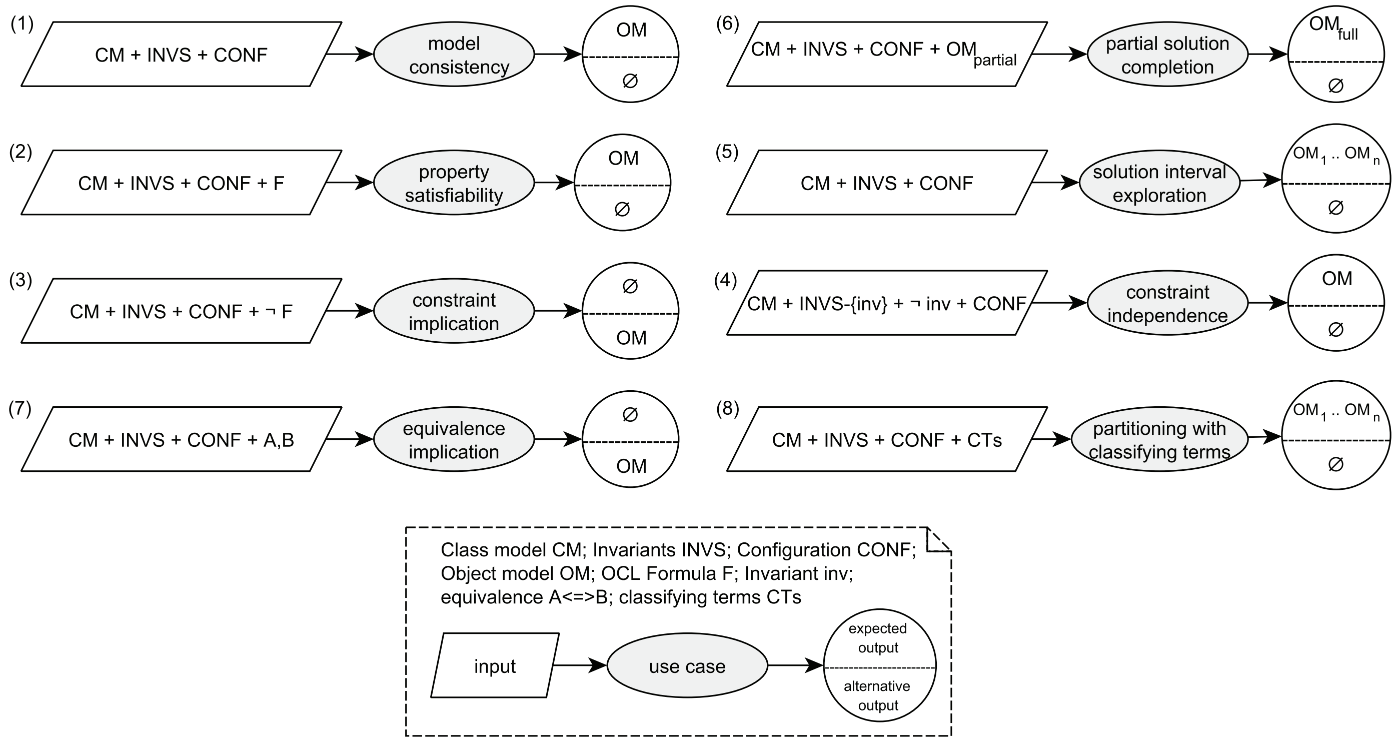


Figure 3: Use case input and use case output for main and alternative flow.

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>`[1]. 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, which is used for the consistency use case, exactly 10 persons and 11 parenthood links together with attribute values from the stated enumerations are employed. 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
```

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

```
Person_max = 10
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
```

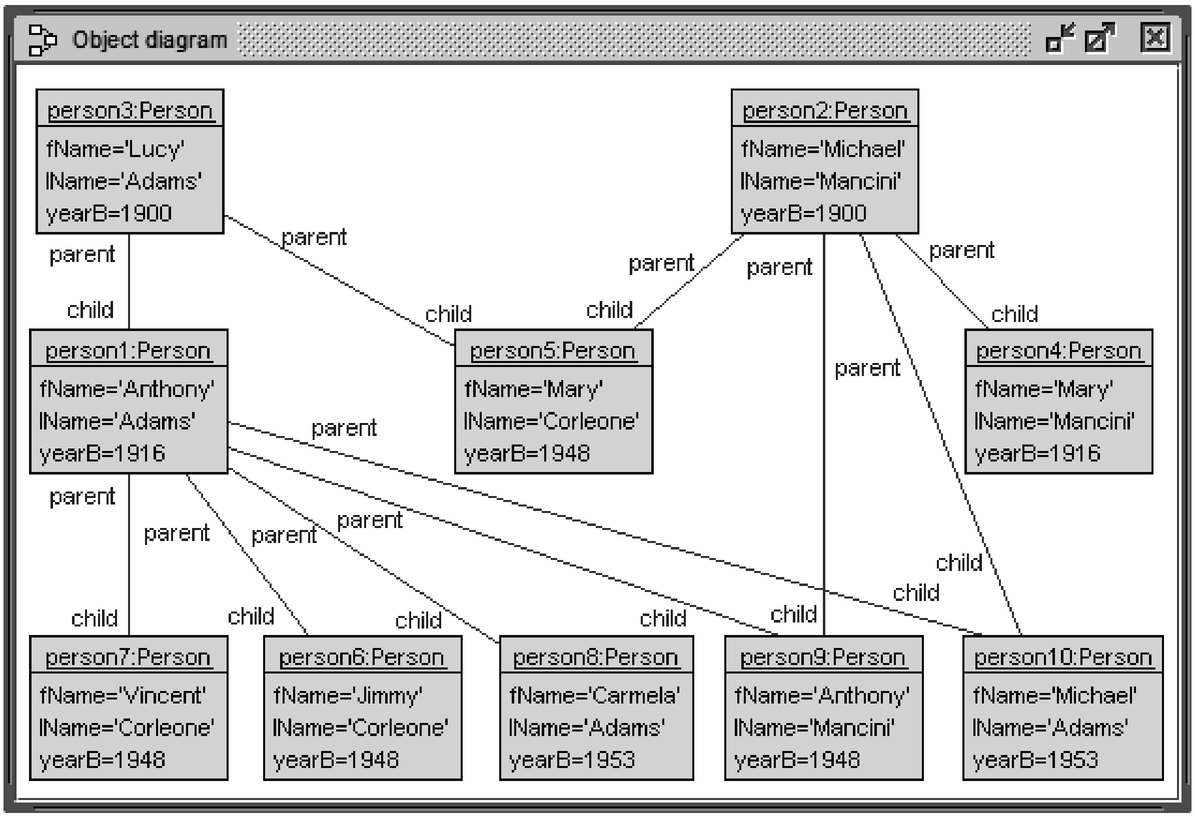


Figure 4: Generated instantiation for model consistency.

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

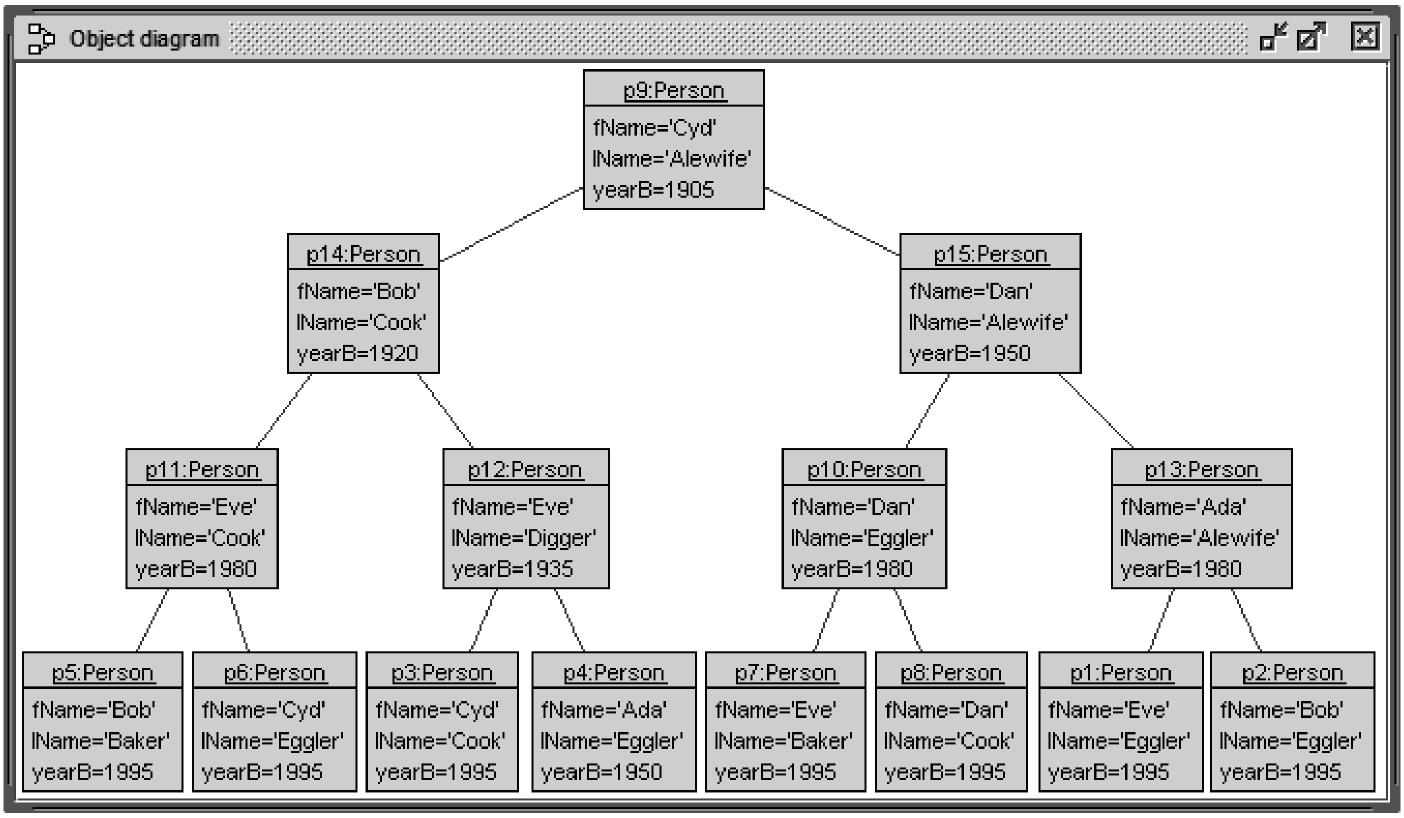


Figure 5: Generated instantiation for property satisfiability.

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 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.

One last general remark: One can check in our approach whether it is possible to construct a system state in which all classes have to be populated and in which the population of associations is left open using the class bounds 1..* and the association bounds 0..*; one can also check whether it is possible to build a system state in which all classes and all association have to be populated using the class bounds 1..* and the association bounds 1..*. One could offer specialized use cases for these two situations.

3.2. Property Satisfiability

Property satisfiability 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 arbitrary 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 satisfiability 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, because all objects except the root need to have one parent.

```
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 = -1 -- upper bound '-1' represents '*'
```

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
```

The use case 'property satisfiability' can also be employed for fault detection and exploration. For example, a new requirement could be that all persons have to have two parents or no parents at all; then one could check whether this invariant is already fulfilled by adding the property `Person.allInstances->exists(parent->size<>2 and parent->size<>0)` and by employing the property satisfiability use case. In this case this would lead to a counterexample disproving the faulty assumption that the new requirement is already granted by the current model.

The difference between the use cases ‘model consistency’ and ‘property satisfiability’ lies in the fact that the first use case involves only the original model whereas the second one involves an additional constraint. The underlying technical mechanism are the same, but it is our aim to offer specialized use cases for specialized problems, even if they apply similar techniques.

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.

The expected result and answer of the use case ‘constraint implication’ (and ‘equivalence implication’ to be discussed further down) is an UNSAT because the implication formula is added in negated form during the quality assurance process. The process has to assert that the reason for the UNSAT is the fact that the logical negation has been made; the reason for the UNSAT should not be the circumstance that the original model or the configuration are unsatisfiable; in the subordinate use case ‘sanity check’ the satisfiability of the original model enriched by the implication formula (in non-negated, positive form) and the configuration is investigated. The use case ‘sanity check’ can employ the original configuration or, alternatively, a narrower configuration, i.e., a configuration in which all bounds are more restricted and thus narrower than the original configuration. If a narrower configuration is satisfiable, then also the original, broader configuration is satisfiable. A narrower configuration might show a more interesting solution, e.g., a solution with links whereas the original configuration did not make any restriction on the links and could allow solutions without any links. In summary and to repeat what was already stated above: if the developer wants to draw conclusions from an UNSAT, the reason for the UNSAT must always be explicitly investigated, and the responsible, ‘guilty’ element (like class model with model-inherent constraints, explicit invariant, loaded invariant, invariant flag, configuration) must be identified.

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.

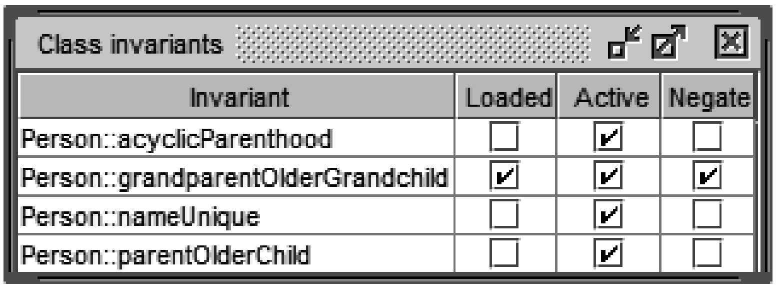
```
context gp : Person inv grandparentOlderGrandchild:
  gp.child.child->forAll( gc | gp.yearB+30 <= gc.yearB )
```

The following configurations `broadConf` and `narrowConf` that are used further down and bind the number of persons and `Parenthood` links. The possible attribute values are as above in the last configuration and are present, but are not repeated in the listing. The first configuration `broadConf` allows up to 6 `Person` objects and does not restrict the links. The second configuration `narrowConf` allows exactly 4 `Person` objects and exactly 4 links.

```
broadConf                narrowConf

Person_min = 0           Person_min = 4
Person_max = 6           Person_max = 4
Parenthood_min = 0       Parenthood_min = 4
Parenthood_max = -1      Parenthood_max = 4
```

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.



Invariant	Loaded	Active	Negate
Person::acyclicParenthood	☐	✓	☐
Person::grandparentOlderGrandchild	✓	✓	✓
Person::nameUnique	☐	✓	☐
Person::parentOlderChild	☐	✓	☐

Figure 6: Status of original and loaded invariants.

```
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 broadConf
     ModelTransformator: Translation time: 296 ms
     ModelValidator: UNSATISFIABLE
         Translation time: 171 ms      Solving time: 2590 ms
```

For the sanity check, we consider the loaded invariant in positive form and run the same model validator call with the narrow configuration that requires exactly 4 Person objects and 4 Parenthood links. The respective configuration name is passed to the model validator. The call is satisfiable. This assures that the reason for the UNSAT before was the negation of the loaded invariant.

```
use> constraints -flags Person::grandparentOlderGrandchild -n
use> constraints -flags
     -- active class invariants:
     Person::acyclicParenthood
     Person::grandparentOlderGrandchild
     Person::nameUnique
     Person::parentOlderChild
use> mv -validate grandparentOlderGrandchild.properties narrowConf
     ModelValidator: SATISFIABLE
         Translation time: 169 ms      Solving time: 62 ms
```

We emphasize that the difference between the two command sequences is the option `+n` and the result `UNSATISFIABLE` in the first sequence and the option `-n` and the result `SATISFIABLE` in the second sequence.

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`[2].

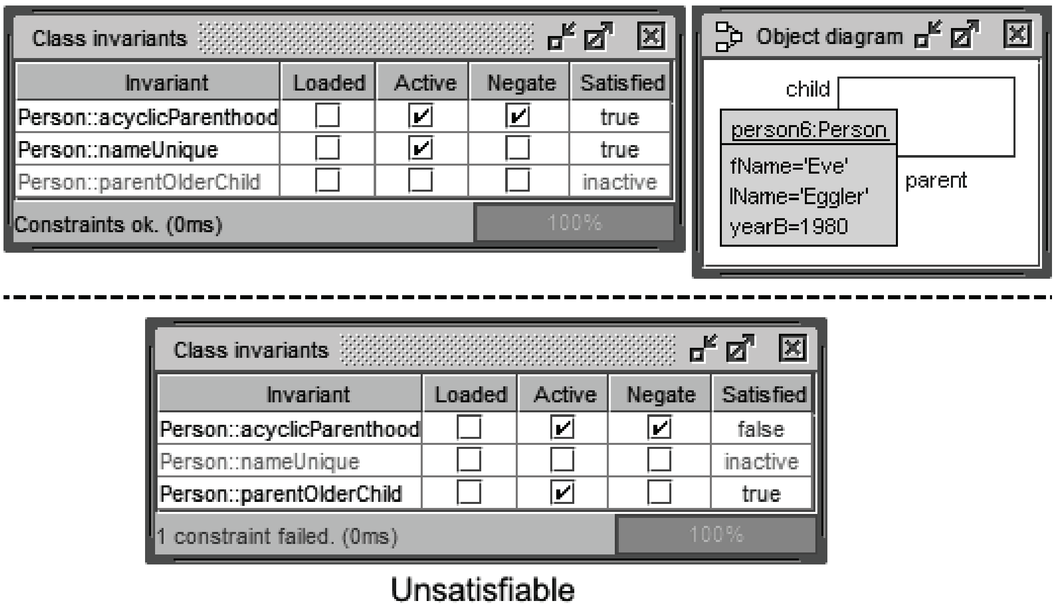
Concerning the example, the configuration for the independence use case is the same as for the constraint implication use case. Below you see the protocol for calling the model validator with the independence option. You see that the invariants are indeed *not* independent. As detailed in the protocol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

```
use> mv -invIndep invIndep.properties all
```

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

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

use> mv -validate invIndep.properties
     ModelValidator: UNSATISFIABLE
```


Class invariants

Invariant	Loaded	Active	Negate	Satisfied
Person::acyclicParenthood	☐	✓	✓	true
Person::nameUnique	☐	✓	☐	true
Person::parentOlderChild	☐	☐	☐	inactive

Constraints ok. (0ms) 100%

Object diagram

person6:Person
fName='Eve'
lName='Eggler'
yearB=1980

Class invariants

Invariant	Loaded	Active	Negate	Satisfied
Person::acyclicParenthood	☐	✓	✓	false
Person::nameUnique	☐	☐	☐	inactive
Person::parentOlderChild	☐	✓	☐	true

1 constraint failed. (0ms) 100%

Figure 7: Invariant status for independence and generated counterexample.

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 associations 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.” [14, p. 16].

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

3.6. Partial Solution Completion

The next 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

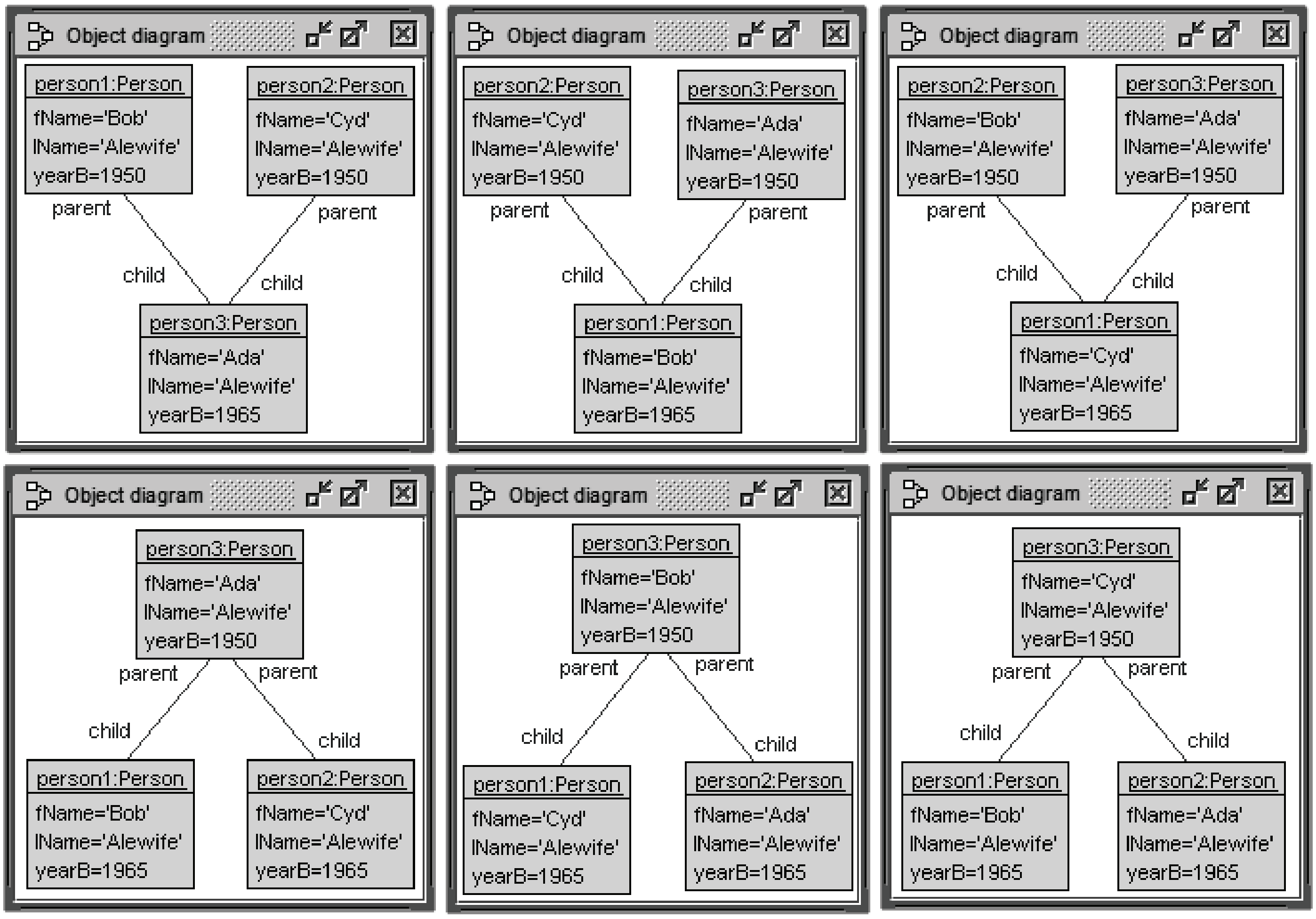


Figure 8: Solution interval with 6 object diagrams.

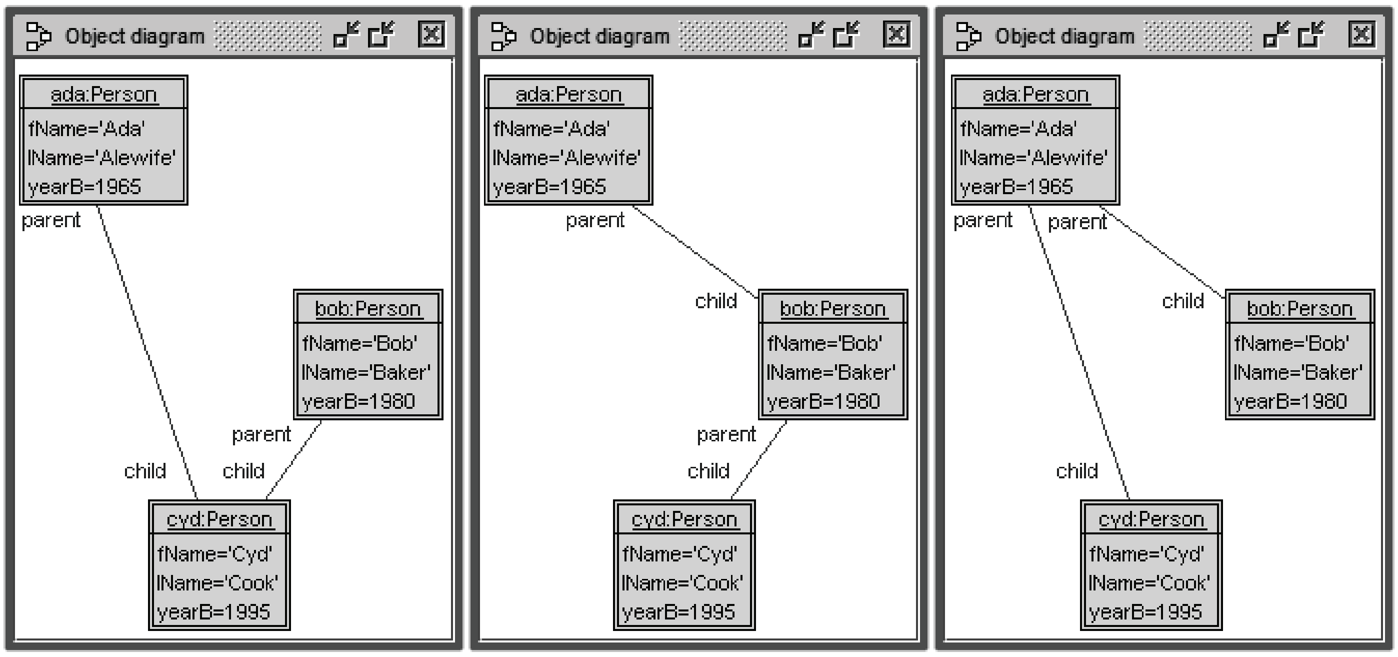


Figure 9: Completions of partially specified solutions.

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. This flag is used by the model validator `mv` command. It can be set during object generation but it must be specified before the model validator is invoked.

The configuration fixes the number of `Person` objects to three and the number of `Parenthood` links to two. Appropriate attribute values are provided, as before.

```
Person_min = 3
Person_max = 3
Parenthood_min = 2
Parenthood_max = 2
```

The following protocol explains the construction of the three objects and fixes their attributes. Alternatively the object diagram construction can be read from a SOIL file. 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> ...                           -- 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. This is emphasized in the figure by the double rectangles surrounding the objects.

3.7. Equivalence Implication

The 'equivalence implication' use case is intended for checking that two additional OCL formulas `A` and `B` express identical requirements, i.e., that under the given model the two formulas are equivalent. This is realized by adding an invariant to the model that expresses the equivalence and by logically negating that invariant. Then it is checked (as in the 'constraint implication' use case) that the model becomes unsatisfiable under the stated configuration in the provided finite search space. A 'sanity check' guarantees that the reason for the unsatisfiability is the logical negation of the loaded invariant. An essential step in this verification use case is adding a new combined invariant `context C inv A_EQUIV_B: (A implies B) and (B implies A)`.

As a concrete example we consider the following two invariants that express in two different ways the multiplicity restriction that a person has up to two parents. The first formulation employs the collection operation `size()`, the second formulation checks for three different `Set` literals.

```
context p:Person inv parent_0_2_size:
  0<=p.parent->size() and p.parent->size()<=2

context p:Person inv parent_0_2_Set: let P=Person.allInstances() in
  p.parent=Set{} or
  P->exists(c | p.parent=Set{c}) or
  P->exists(c | P->exists(d | c<>d and p.parent=Set{c,d}))
```

One now has to construct an invariant that expresses the equivalence between the above formulas. This invariant is the invariant that is given in negated form to the model validator.

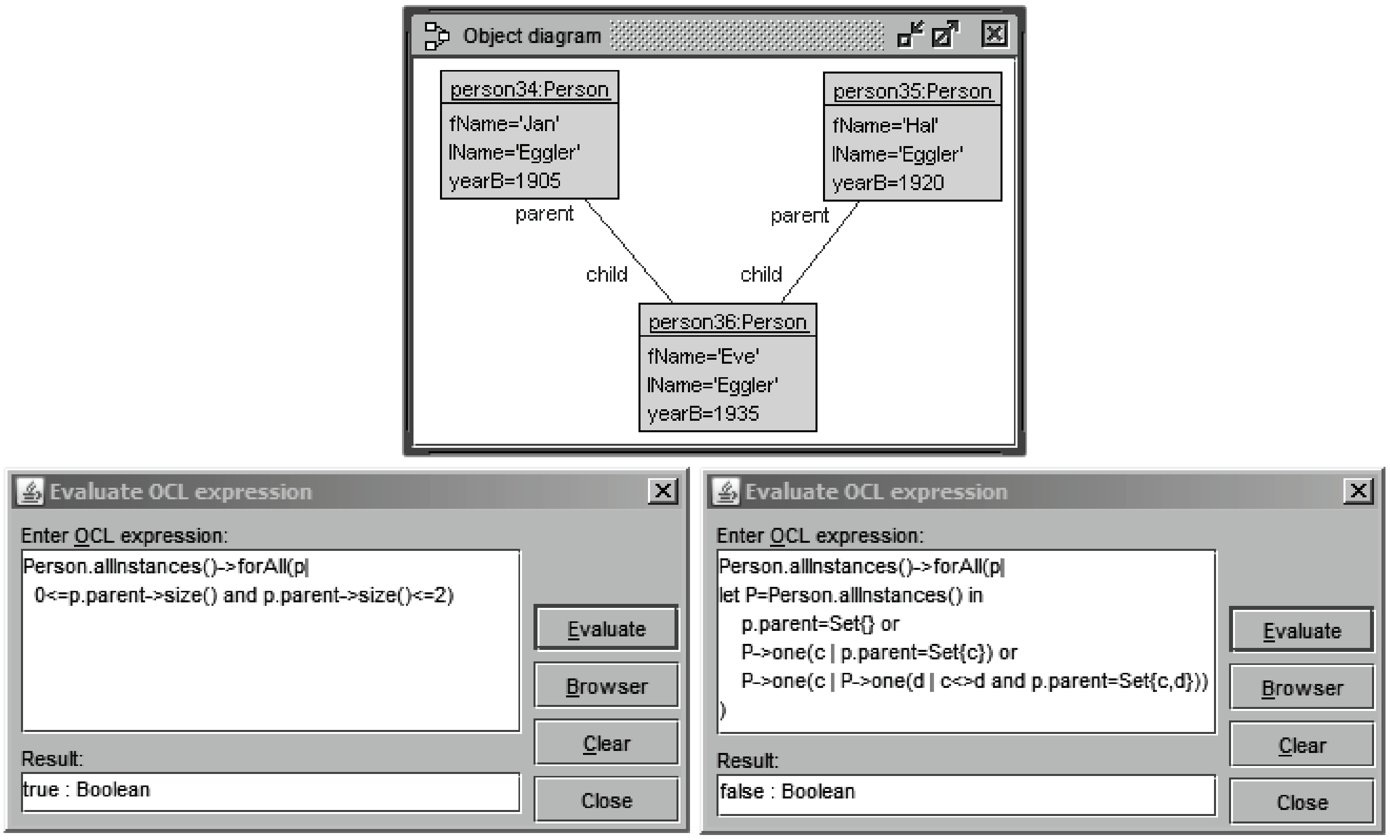
```
context p:Person inv parent_0_2_size_EQUIV_parent_0_2_Set:
  let A=0<=p.parent->size() and p.parent->size()<=2 in
  let B=let P=Person.allInstances() in
        p.parent=Set{} or
        P->exists(c | p.parent=Set{c}) or
        P->exists(c | P->exists(d | c<>d and p.parent=Set{c,d})) in
  (A implies B) and (B implies A)
```

The command sequence that is given to the model validator in this example is the following one. The narrow and broad configuration only differ in the object and link intervals. The narrow configuration requires 6..6 Person objects and 5..5 Parenthood links; the broad configuration requires 0..12 Person objects and 0..* Parenthood links.

```
constraints -load equivalenceImplication.invs
constraints -flags Person::parent_0_2_size                        -d -n
constraints -flags Person::parent_0_2_Set                         -d -n
constraints -flags Person::parent_0_2_size_EQUIV_parent_0_2_Set -d -n
mv -validate equivalenceImplication.properties narrow

constraints -flags Person::parent_0_2_size                        -d -n
constraints -flags Person::parent_0_2_Set                         -d -n
constraints -flags Person::parent_0_2_size_EQUIV_parent_0_2_Set -d +n
mv -validate equivalenceImplication.properties broad
```

The first call to the model validator with the narrow configuration is satisfiable; this means that the same call with the broad configuration is also satisfiable; the second call having as the only difference the negated equivalence invariant is unsatisfiable; the ‘guilty’ element that is responsible for unsatisfiability is the negation flag; thus it is proved in the finite search space that the two subformulas are equivalent.



```
Person.allInstances()->forAll(p|
 0<=p.parent->size() and p.parent->size()<=2)
```

```
true : Boolean
```

```
Person.allInstances()->forAll(p|
let P=Person.allInstances() in
   p.parent=Set{} or
   P->one(c | p.parent=Set{c}) or
   P->one(c | P->one(d | c<>d and p.parent=Set{c,d}))
)
```

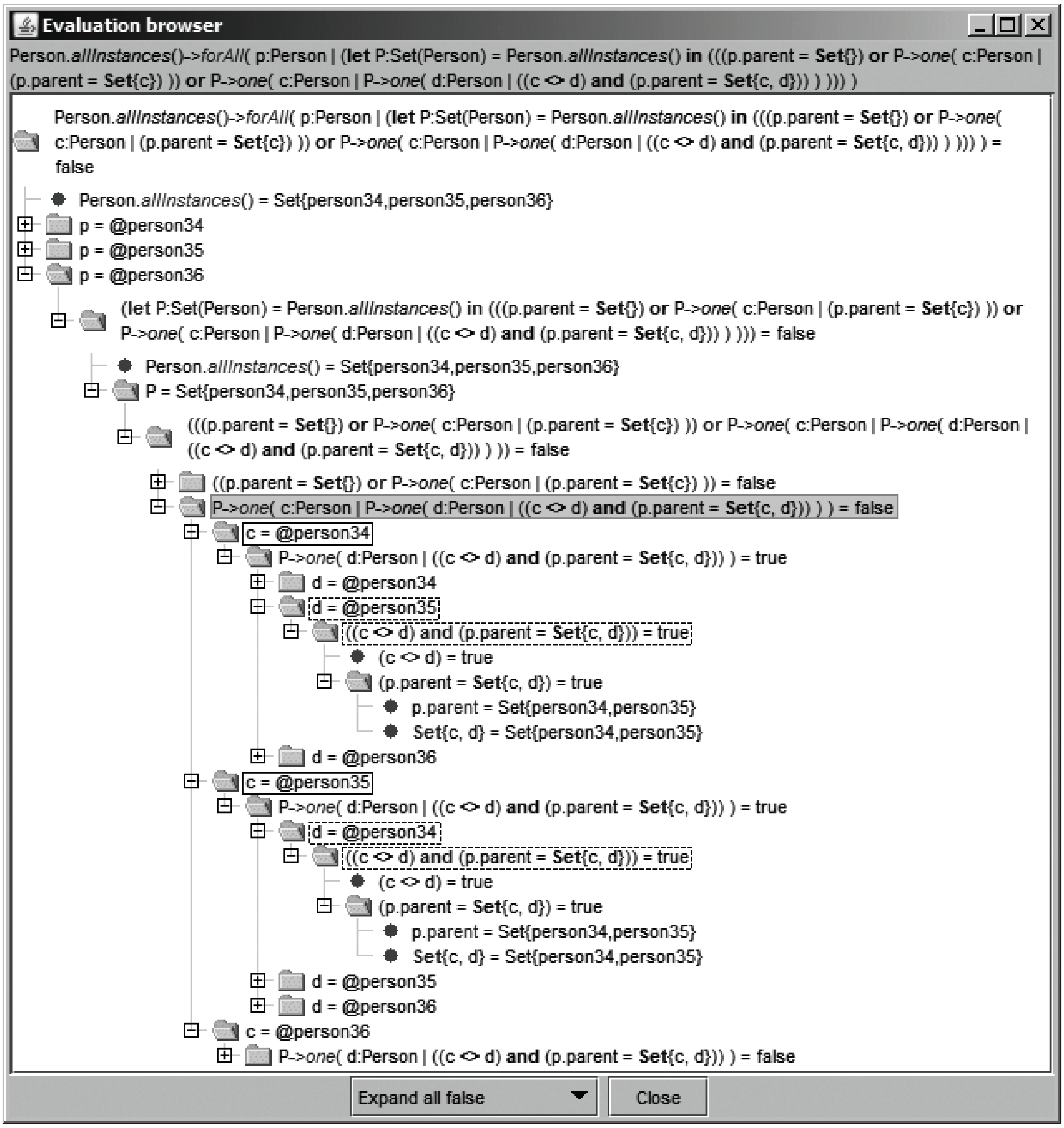
```
false : Boolean
```

Figure 10: Counter example generated for non-equivalent invariants.

An interesting variation comes into play when one slightly modifies the second part of the equivalence: One can replace the OCL `exists` operation by the OCL `one` operation and ask whether the two subformulas are still equivalent.

```
context p:Person inv parent_0_2_size_EQUIV_parent_0_2_Set_ONE:
  let A=0<=p.parent->size() and p.parent->size()<=2 in
  let B=let P=Person.allInstances() in
        p.parent=Set{} or
        P->one(c | p.parent=Set{c}) or
        P->one(c | P->one(d | c<>d and p.parent=Set{c,d})) in
  (A implies B) and (B implies A)
```

When one now asks the model validator to check for the equivalence, the call becomes satisfiable and the model validator finds a counter example as displayed in Fig. 10. In the figure, the left window shows the ‘A’ part, the right window the ‘B’ part of the equivalence; the evaluation is different for A and B. The counter example can be further analyzed with the USE evaluation browser in Fig. 11, which is obtained by clicking the ‘Browser’ button in the right evaluation window. The analysis reveals the reason for the non-equivalence of the two formulas, as indicated in Fig. 11. The subformula ‘`P->one(c | P->one(d | c<>d and p.parent=Setc,d))`’ evaluates to false, because there is not only one substitution for the variable `c` making the subformula true, but there are two substitutions: `c=person34` and `c=person35`. Indeed, when one replaces the second `one`



```
Person.allInstances()->forAll( p:Person | (let P:Set(Person) = Person.allInstances() in (((p.parent = Set{}) or P->one( c:Person |
(p.parent = Set{c}) )) or P->one( c:Person | P->one( d:Person | ((c <> d) and (p.parent = Set{c, d})) ) ))) )

Person.allInstances()->forAll( p:Person | (let P:Set(Person) = Person.allInstances() in (((p.parent = Set{}) or P->one(
c:Person | (p.parent = Set{c}) )) or P->one( c:Person | P->one( d:Person | ((c <> d) and (p.parent = Set{c, d})) ) ))) ) =
false
  Person.allInstances() = Set{person34,person35,person36}
  p = @person34
  p = @person35
  p = @person36
    (let P:Set(Person) = Person.allInstances() in (((p.parent = Set{}) or P->one( c:Person | (p.parent = Set{c}) )) or
    P->one( c:Person | P->one( d:Person | ((c <> d) and (p.parent = Set{c, d})) ) ))) = false
      Person.allInstances() = Set{person34,person35,person36}
      P = Set{person34,person35,person36}
        (((p.parent = Set{}) or P->one( c:Person | (p.parent = Set{c}) )) or P->one( c:Person | P->one( d:Person |
        ((c <> d) and (p.parent = Set{c, d})) ) )) = false
          ((p.parent = Set{}) or P->one( c:Person | (p.parent = Set{c}) )) = false
          P->one( c:Person | P->one( d:Person | ((c <> d) and (p.parent = Set{c, d})) ) ) = false
            c = @person34
              P->one( d:Person | ((c <> d) and (p.parent = Set{c, d})) ) = true
                d = @person34
                d = @person35
                  ((c <> d) and (p.parent = Set{c, d})) = true
                    (c <> d) = true
                    (p.parent = Set{c, d}) = true
                      p.parent = Set{person34,person35}
                      Set{c, d} = Set{person34,person35}
                d = @person36
            c = @person35
              P->one( d:Person | ((c <> d) and (p.parent = Set{c, d})) ) = true
                d = @person34
                  ((c <> d) and (p.parent = Set{c, d})) = true
                    (c <> d) = true
                    (p.parent = Set{c, d}) = true
                      p.parent = Set{person34,person35}
                      Set{c, d} = Set{person34,person35}
                d = @person35
                d = @person36
            c = @person36
              P->one( d:Person | ((c <> d) and (p.parent = Set{c, d})) ) = false
```

Figure 11: Analysis with evaluation browser for non-equivalent invariants.

(binding the second occurrence of the variable `c`) with an `exists`, one obtains again an equivalence between OCL formulas.

3.8. Partitioning with Classifying Terms

The use case 'partitioning with classifying terms' can be used for the automatic construction of test suites and is thus intended to explore and to build many test cases in form of object diagrams. The test cases are constructed and partitioned by so-called classifying terms that are closed OCL query terms over the model. Each object diagram in an equivalence class has the same evaluation result for all classifying terms; two

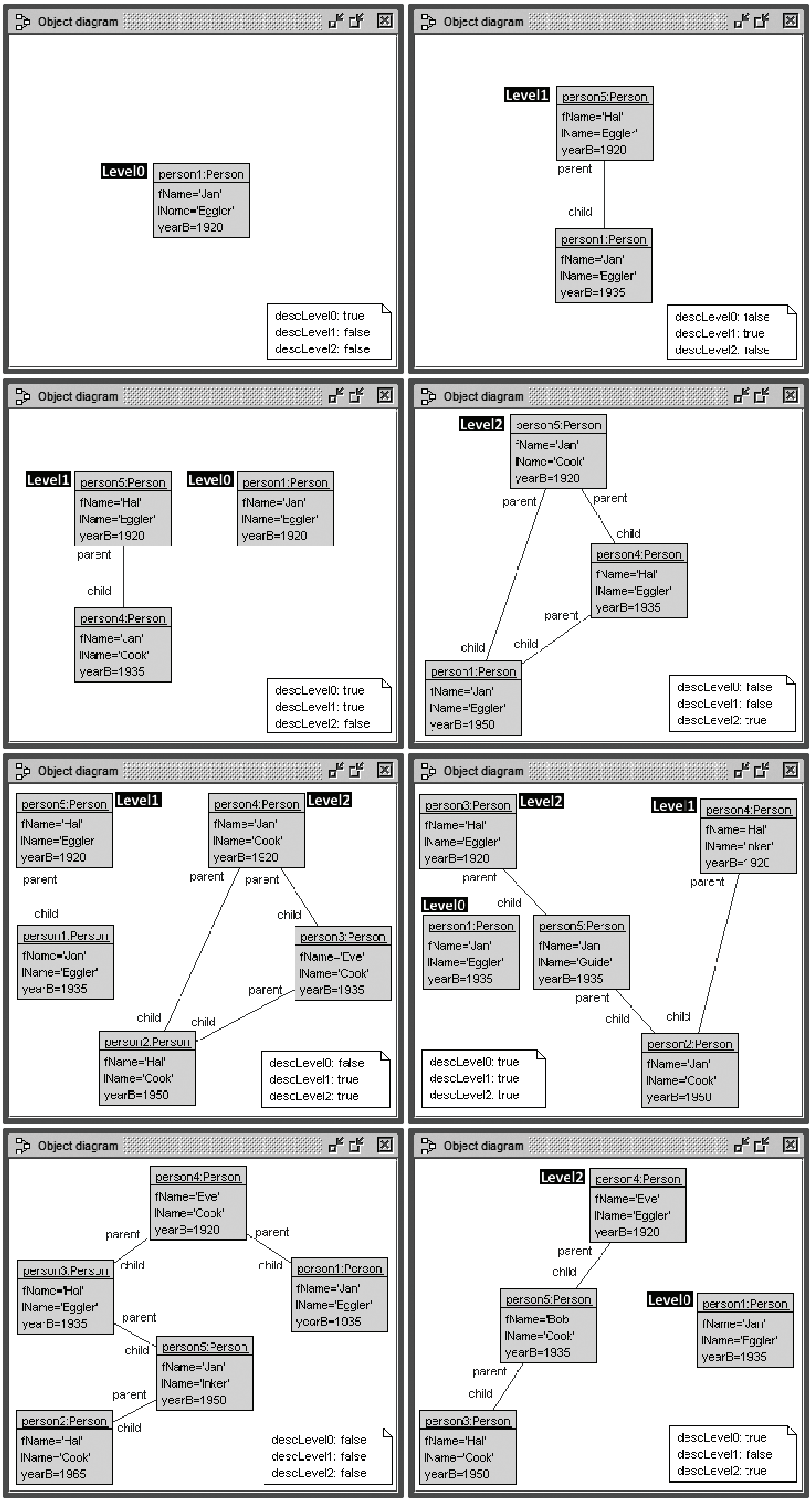


Figure 12: Eight solutions for exploration with classifying terms.

different equivalence classes differ by the value of at least one classifying term. From each class one canonical representative is selected. The use case allows the developer to systematically construct few diverse test cases on the basis of the classifying terms that describe particular properties to be present in the object diagrams. The command `mv -scrollingAllCT <PropertyFile>` instructs the model validator to use classifying terms which can be entered interactively on the shell or which can come from a file. The developer can employ Integer- or Boolean-valued classifying terms. For example, if the developer uses three Boolean-valued classifying terms, each property determined by the respective term can either hold or not hold, resulting in at most eight equivalence classes. If the Boolean properties cannot be arbitrarily combined, fewer classes may occur.

For the example we will also use three Boolean-valued classifying terms. Roughly speaking, the three terms require the existence of a person without parents having descendent level 0, descendent level 1 or descendent level 2, respectively. Descendent level 0 means the person has no descendants, descendant level 1 means the person has children with no descendants, and descendant level 2 means the person has grandchildren with no descendants. The formal definition of the three Boolean-valued classifying terms is as follows. As central requirements we see on level 0 `p.child->size()=0`, on level 1 `p.child.child->size()=0`, and on level 2 `p.child.child.child->size()=0`.

```
descLevel0: Person.allInstances->exists(p |
  p.parent->size()=0 and p.child->size()=0)


descLevel1: Person.allInstances->exists(p |
  p.parent->size()=0 and p.child->size()>0 and
  p.child.child->size()=0)
descLevel2: Person.allInstances->exists(p |
  p.parent->size()=0 and p.child->size()>0 and
  p.child.child->size()>0 and p.child.child.child->size()=0)
```

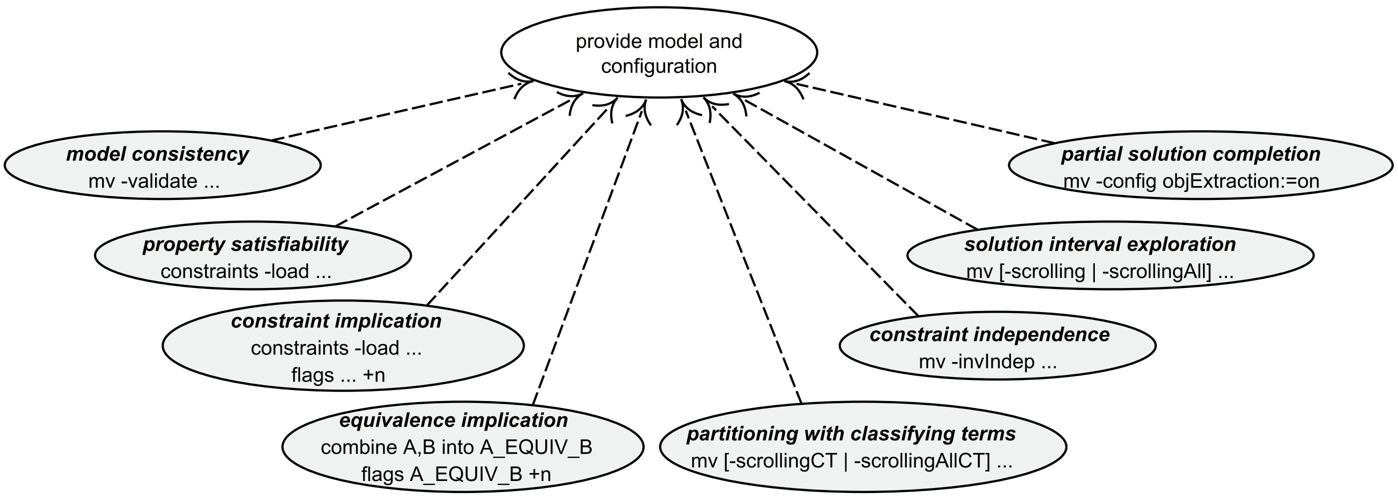
In Fig. 12 the eight found object diagram solutions are shown. Additionally the value of the named classifying terms are stated. In the figure it is also indicated which object from the diagram takes the role descendent level 0, descendent level 1 or descendent level 2 (indicated as Level0, Level1, Level2 in the figure). The respective model validator call on the USE command shell looks as follows.

```
mv -scrollingAllCT descLevel.properties
```

3.9. Viewing the Use Cases and the Model Validator by Technical Details

In Fig. 13 the eight use cases are shown once again, here in an overview manner concentrating on technical details. The figure displays the central commands that are particular to the respective use case and that contribute essentially to the functionality of the use case. Please note that each pair of uses cases has different essential commands. Thereby, it is demonstrated that each use case has a unique functionality.

Table 2 shows a list of UML and OCL features and marks for each one whether the USE model validator plugin supports that feature fully, partially or not at all. The top of the table concentrates on UML features with a wide support for most basic and advanced



provide model and configuration
model consistency
mv -validate ...
property satisfiability
constraints -load ...
constraint implication
constraints -load ...
flags ... +n
equivalence implication
combine A,B into A_EQUIV_B
flags A_EQUIV_B +n
partitioning with classifying terms
mv [-scrollingCT | -scrollingAllCT] ...
constraint independence
mv -invIndep ...
solution interval exploration
mv [-scrolling | -scrollingAll] ...
partial solution completion
mv -config objExtraction:=on

Figure 13: Emphasizing distinctive technical details in the use cases.

structural features. The model validator plugin itself is designed with static validation and verification tasks in mind, however with another USE plugin, which implements the filmstripping approach [15, 16], the capabilities can be used to also check behavior in models in the form of operation calls with pre- and postconditions[3]. The lower part of Table 2 lists OCL features. Of the OCL types, the most commonly used ones are supported by the USE model validator: `Boolean`, `Integer`, `Class` types and the collection type `Set`. Strings are supported as tokens and can only be checked for equality, which affects the supported OCL operations as, e.g., `substring` and `concat` cannot be implemented with this definition of strings. The other OCL operations follow the support of their respective types, where most operations on the supported types are implemented in the tool. However, the tool cannot represent types as such, which means that `OclType` is also not supported and it does not support the powerful operation `iterate`. The latter can occasionally be replaced with the similarily powerful operation `closure`, which is supported.

[3]A detailed introduction of the filmstripping approach is beyond the scope of this paper and not required for the use cases presented.

Table 2: USE Model Validator supported UML/OCL Features

Unified Modeling Language (UML)		
Class features		
✓ Class		
✓ Abstract Class		
✓ Inheritance		
✓ Multiple Inheritance		
✓ Attribute		
✓ Derived Value		
✗ Initial Value		
✓ Enumeration		
✓ Invariant		
Association features		
✓ Binary Association		
✓ N-ary Association		
○ Aggregation	limited support of cycle freeness (otherwise ✓)	
○ Composition	limited support of cycle freeness (otherwise ✓)	
✓ Multiplicity		
✓ Association Class		
✓ Derived Association End		
✗ Qualified Association		
✗ Redefines, Subsets, Union		
Operation features		
✓ Query Operation		
✓ Parameter		
✓ Return Value		
✗ Recursion		
✗ Operation Call (non query)	checking behavior possible via filmstripping approach	
✗ Parameter	∟ with filmstripping approach	
✗ Return Value	∟ with filmstripping approach	
✗ Pre-/Postcondition	∟ with filmstripping approach	
✗ Nested Operation Call		
Object Constraint Language (OCL)		
OCL types		
✓ Boolean	✓ Integer	✓ Class Type
○ String	○ Real	✗ UnlimitedNatural
✓ Set	✗ Bag	✗ Sequence
✗ OrderedSet	✗ Nested collections	✗ Tuple
OCL operations		
✓ Comparison Operators	✓ Boolean Operations	✓ Integer Operations
○ String Operations	✗ substring	✗ concat
✓ <Class>.allInstances	✗ <Assoc>.allInstances	✓ size
✓ isEmpty/notEmpty	✓ includes/excludes	✓ including/excluding
✓ forAll/exists	✓ select/reject	✓ one
✓ isUnique	✓ union/intersection	○ any
✓ collect	✓ closure	✗ iterate
✓ toString	○ sum	✓ oclIsType/KindOf
✓ selectByType/Kind	✓ oclAsType	✗ oclType

✓ supported element – ✗ unsupported element – ○ partially supported element

4. Use Cases Applied in a Larger Example

In order to give a first understanding of the validation and verification use cases with the model validator, the use cases have been explained and illustrated with a relative small example in Sec. 3. To further demonstrate the usefulness and applicability of the use cases, a larger and classical example from the literature is considered and handled now. This section is an in-depth discussion of the results achieved by the model validator, detailing the output parts of Fig. 3 and what these outputs mean for the validation and verification process. Six from the above eight model validation and verification use cases will be explored in a larger example, which includes 10 classes, 12 derived associations and 29 invariants. The invariants introduced in this example are non-trivial, since they express primary key and foreign key requirements from the relational datamodel with nested quantifiers and collection operations. 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 depicted as objects from the UML class diagram.

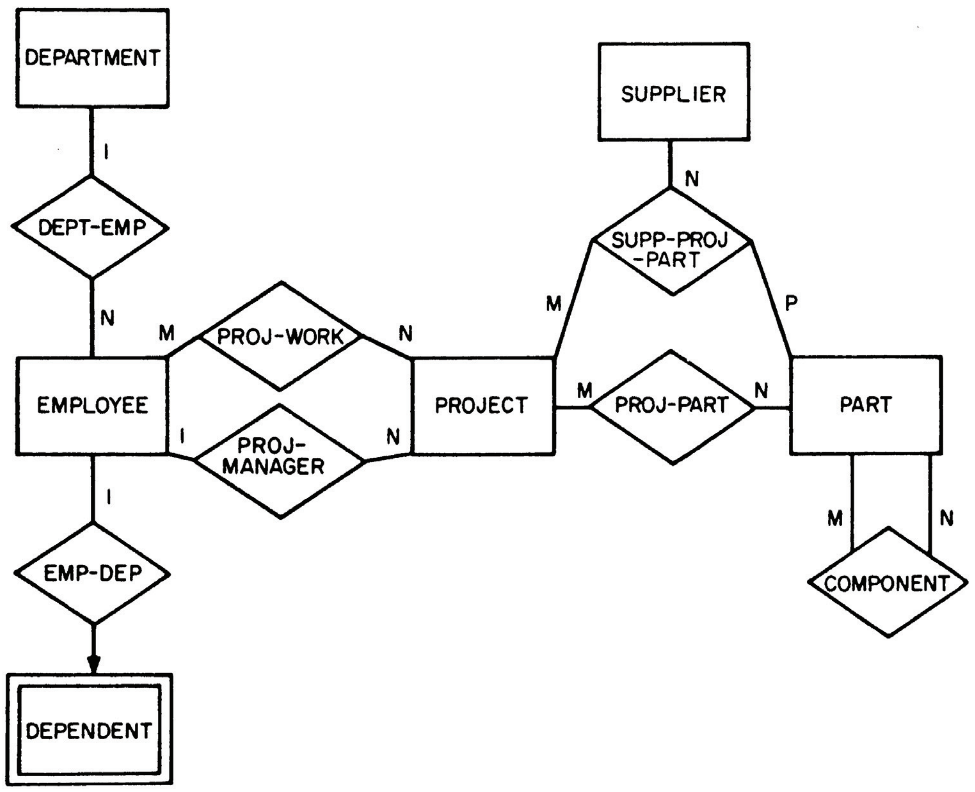
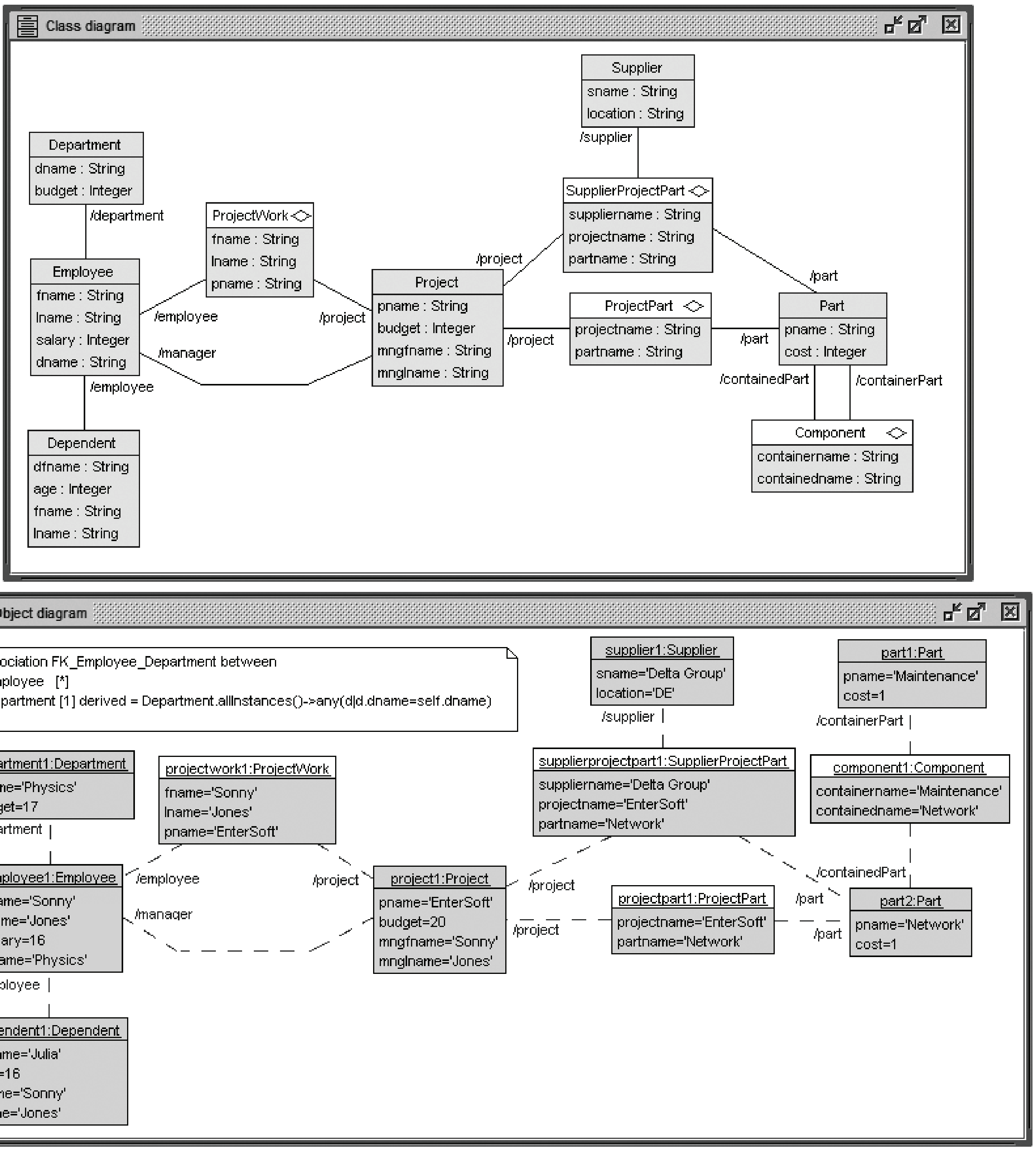


Figure 14: ER example schema from Chen's original paper.

The Entity-Relationship (ER) diagram in Fig. 14 is the running example schema taken from the original paper on the ER model [17]. The example is an Employee-Project-Part world: an employee belongs to a department and can have dependents as, for example, children; employees work on projects and can be their manager; projects deploy parts, and in projects these parts are provided by suppliers; parts can have components that are again parts. The example uses relationships of various kinds: many-to-many, functional, ternary, and part-whole relationship. We use the term functional relationship to denote a many-to-one relationship as a source instance is functionally mapped to at most one or exactly one target instance.



```
association FK_Employee_Department between
  Employee   [*]
  Department [1] derived = Department.allInstances()->any(d|d.dname=self.dname)
end
```

Figure 15: Chen's example as a UML class diagram with classes for relations and derived associations for foreign keys; example object diagram with objects representing tuples.

In Fig. 15 we see how the ER schema is represented as a relational database schema in form of a UML class diagram; additionally, a simple object diagram illustrates the representation of a relational database state with tuples. The core of the transformation from the ER 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 relationship is mapped to (a) an attribute (or many attributes) in the relation resp. the class corresponding to the source entity of the functional relationship and (b) a derived asso-

```
Component::container_name_contained_name_primary_key
Department::dname_primary_key
Dependent::dfname_fname_lname_primary_key
Employee::fname_lname_primary_key
Part::pname_primary_key
Project::pname_primary_key
ProjectPart::project_name_part_name_primary_key
ProjectWork::fname_lname_pname_primary_key
Supplier::sname_primary_key
SupplierProjectPart::supplier_project_part_name_primary_key

Component::contained_name_foreign_key_Part
Component::container_name_foreign_key_Part
Dependent::fname_lname_foreign_key_Employee
Employee::dname_foreign_key_Department
Project::fname_lname_foreign_key_Employee
ProjectPart::part_name_foreign_key_Part
ProjectPart::project_name_foreign_key_Project
ProjectWork::fname_lname_foreign_key_Employee
ProjectWork::pname_foreign_key_Project
SupplierProjectPart::part_name_foreign_key_Part
SupplierProjectPart::project_name_foreign_key_Project
SupplierProjectPart::supplier_name_foreign_key_Supplier

Component::acyclic
Department::budget_positive
Dependent::age_reasonable
Employee::DepartmentBudget_greater_allEmployeeSalary
Employee::salary_positive
Part::cost_positive
Project::ProjectBudget_greater_PartCost
```

Figure 16: Constraints defined in Chen's example.

ciation for the foreign key; a general relationship (many-to-many, ternary, part-whole) is mapped to (a) one relation represented as a class and (b) derived associations for the foreign keys. The white-headed classes with a graphical relationship stereotype originate from relationships; the other classes come from entities.

Thus the UML class diagram shows six classes originating from entities: Department, Employee, Dependent, Project, Supplier, and Part; and the class diagram displays four classes originating from relationships: ProjectWork, ProjectPart, SupplierProjectPart, and Component. The complete USE file of the example can be found in the appendix. The USE file shows all the details of the classes, associations, and invariants.

Fig. 16 shows the OCL constraints: each of the ten classes has a primary key constraint with a name ending in 'primary_key'. For each functional relationship (three relationships) and for each 'arm' of the other relationships (nine 'arms') there is a foreign key constraint with a name containing 'foreign_key' (twelve foreign key constraints); the twelve derived role names are shown in the class diagram; the definition of the derived role name 'department' is shown in the figure in the object diagram as a prototypical example; the other derived associations are defined analogously; there are seven other constraints, among them 'Component::acyclic' which requires the part-whole Component 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.

In order to point out the options available in OCL, we have formulated the ten primary key constraints in ten different ways, using different collection operations, e.g., isUnique(), select(), excluding(), isEmpty(), size(), exists(), forAll(). The details can be traced in the appendix. Thereby, we also indicate that the model validator can work well with these standard OCL collection operations. In the following, we show two formulations of the primary key constraints.

```
context Department inv dname_primary_key:
  dname<>null and Department.allInstances()->isUnique(dname)
context p1:Project inv pname_primary_key:
  pname<>null and Project.allInstances()->select(p2 |
    p2<>p1 and p2.pname=p1.pname)->isEmpty()
```

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 stated with the OCL collection operation one(). We only show the requirement for the foreign key from Employee to Department.

```
context e:Employee inv dname_foreign_key_Department:
  Department.allInstances()->one(d | d.dname=e.dname)
```

Taking together the primary and foreign key constraints, all restrictions on the relational database states have been expressed, and all necessary constraints are stated. In particular, the foreign key connection between the referencing tuple (object) and the referenced tuple (object) are manifested through the respective attribute values. Nothing more is needed. Thus only the objects in the object diagram in Fig. 15 (without the dashed links) completely describe the 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 the derived foreign key association between the Employee and Department. The other derived associations and their roles are formulated analogously.

```
association FK_Employee_Department between
  Employee   [*] role employee
  Department [1] role department
    derived = Department.allInstances()->any(d|d.dname=self.dname)
end
```

If we compare the original ER schema in Fig. 14 and the corresponding relational database schema formulated as a UML class diagram with derived associations in Fig. 15, we see that the graph structures of both diagrams are nearly identical. An eye-catching difference is probably that the three functional relationships Dept-Emp, Emp-Dep, and Proj-Manager are not represented by an independent class, but these relationships are integrated into the relation representing the source entity of the functional original relationship. These three ER relationships are present in the UML class diagram through the referencing foreign key attributes and the derived role names /department, /employee, and /manager.

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. We are not aware of another approach that represents relational foreign keys as derived associations.

Please note that we represent ER relationships in the relational schema by attributes, simply because in a relational schema there are only attributes. In a relational schema (represented as a UML class diagram) there are no explicit relationships and no explicit associations and no explicit association classes.

4.1. Model Consistency

As explained in Sect. 3, 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.

```
Employee_min = 4               Employee_max = 4
Department_min = 4             Department_max = 2
Project_min = 2                Project_max = 2
Dependent_min = 2              Dependent_max = 2
ProjectWork_min = 4            ProjectWork_max = 4
Supplier_min = 2               Supplier_max = 2
Part_min = 4                   Part_max = 4
ProjectPart_min = 4            ProjectPart_max = 4
SupplierProjectPart_min = 2    SupplierProjectPart_max = 4
Component_min = 2              Component_max = 4
```

Fig. 17 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 ten classes and links originating from all twelve foreign key derived associations. We emphasize the fact that, during the construction process, the model validator must take into account the ten classes and the 29 non-trivial invariants defined in the model.

4.2. 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

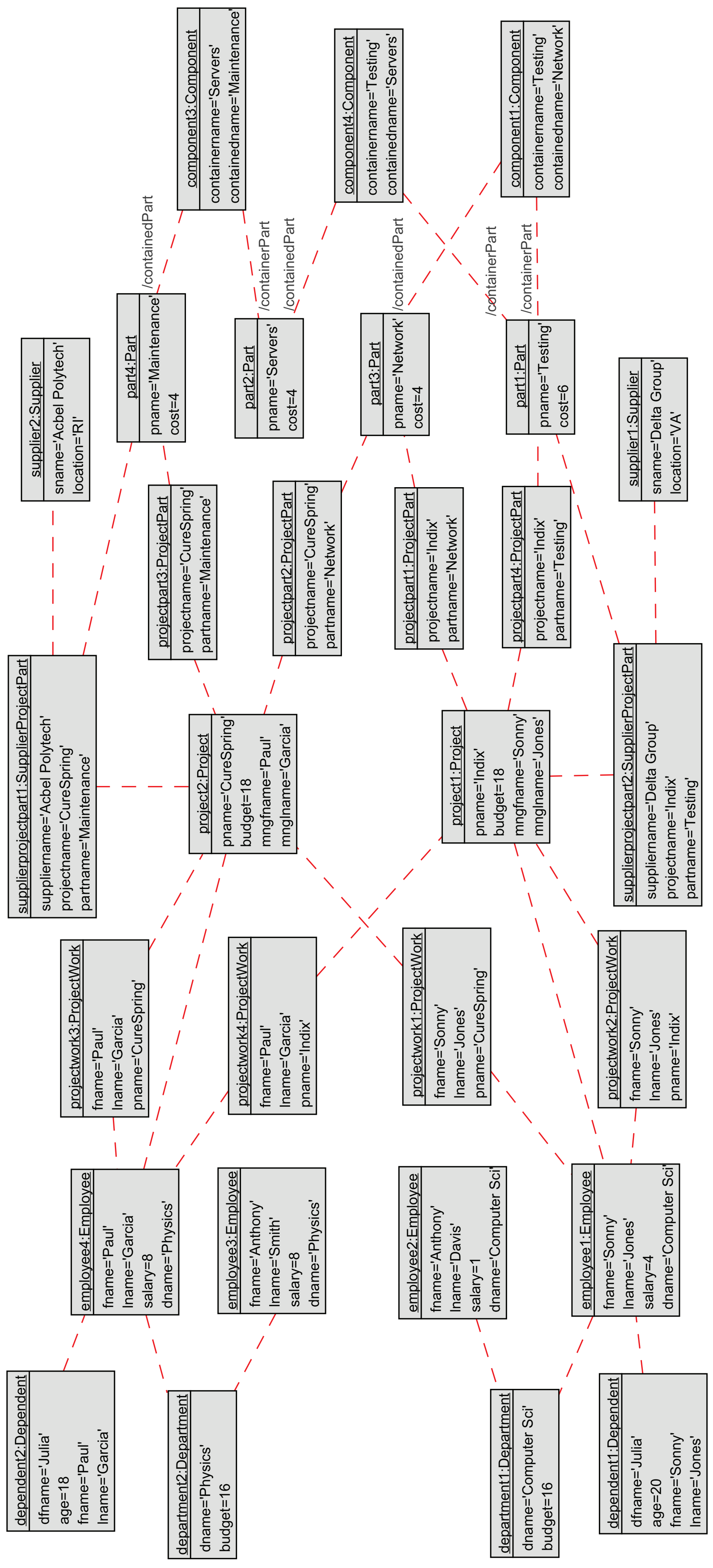


Figure 17: Valid object diagram for model consistency use case.

the answer (1) as an 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 construct a scenario in which every project is an all-department project'. An 'all-department project' means a project in which the employees that are working for the project come from all departments. The property is formulated as the following invariant. Roughly speaking, in this invariant we compare the number of distinct department names (dname) of employees working for a project and the number of department instances.

```
context p:Project inv allDepartment_allProject:
  Employee.allInstances()->select(e |
    ProjectWork.allInstances()->exists(pw |
      pw.pname=p.pname and pw.fname=e.fname and pw.lname=e.lname))->
        collect(dname)->asSet()->size()
  =
  Department.allInstances()->size()
```

Executing the model validator after the model has been enriched with the property invariant, we receive a satisfying scenario as shown in Fig. 18. We here use the same configuration as in the model consistency use case. In order to focus on the property we want to check, Fig. 18 shows only the objects of the relevant classes Department, Employee, ProjectWork, and Project. Both projects have employees from each of the two departments.

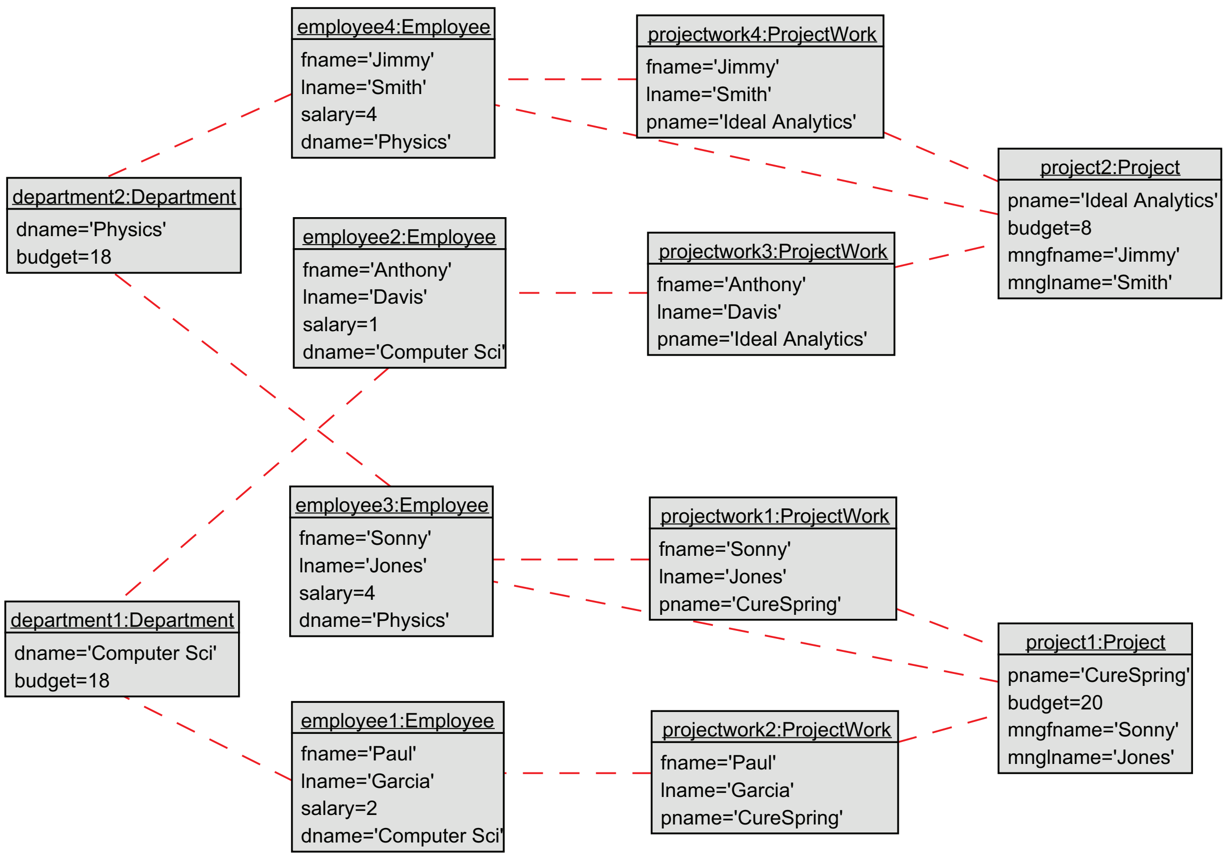


Figure 18: Generated object diagram for property satisfiability use case

4.3. Constraint Implication

As mentioned before, the constraint implication use case realizes a process that checks whether a global constraint is implied by the original model. For the running example model, the invariant that we want to check concerns the Component class. We expect the invariant 'the values of containername and containedname are different' to hold for every instance of class Component.

```
context c:Component inv containerDifferentContained:
  c.containername<>c.containedname
```

First, the invariant containerDifferentContained is loaded to the original model. Inspecting the previously generated object diagram in Fig. 17 we can easily see that this system state fulfills this invariant. This means the model enriched with the containerDifferentContained invariant is satisfiable. Next, we negate the invariant under consideration and then execute the model validator on the modified model. Finally, we receive the answer from the model validator as UNSAT. From this we can conclude that the invariant containerDifferentContained is an implication from the original model.

```
use> constraints -load containerDifferentContained.invs
     Added invariants: Component::containerDifferentContained
use> constraints -flags Component::containerDifferentContained +n
use> mv -validate constraintImplication.properties
     ModelTransformator: Translation time: 2789 ms
     ModelValidator: UNSATISFIABLE
         Translation time: 2894 ms      Solving time: 390 ms
```

In order to check the constraint implication, we only change the number of Part objects and Component objects in the configuration file. The other settings are the same as the configuration for the model consistency use case.

```
Part_min = 0        Part_max = 8
Component_min = 0   Component_max = 8
```

4.4. Equivalence Implication

As an example for proving the equivalence of OCL formulas we consider different formulations of the key constraints. The original formulation for the Department primary key as an invariant was as follows.

```
context Department inv dname_primary_key: dname<>null and
  Department.allInstances()->isUnique(dname)
```

In the use case, two equivalent formulations are added: one formulation using the OCL operation select and one formulation using the OCL operations excluding. The equivalence of the three formulas is stated in cyclic form `(A implies B)` and `(B implies C)` and `(C implies A)` and is also added.

```
context d1:Department inv dname_primary_key_select: dname<>null and
  Department.allInstances()->select(d2|
    d2<>d1 and d2.dname=d1.dname)->isEmpty()
```

```
context d1:Department inv dname_primary_key_excluding: dname<>null and
  Department.allInstances()->excluding(d1)->select(d2|
    d2.dname=d1.dname)->size()=0

context d1:Department
  inv dname_pk_isUnique_EQ_dname_pk_select_EQ_dname_pk_excluding:
  let A = dname<>null and
          Department.allInstances()->isUnique(dname) in
  let B = dname<>null and
          Department.allInstances()->select(d2|
            d2<>d1 and d2.dname=d1.dname)->isEmpty() in
  let C = dname<>null and
          Department.allInstances()->excluding(d1)->select(d2|
            d2.dname=d1.dname)->size()=0  in
  (A implies B) and (B implies C) and (C implies A)
```

The instantiation of the use case follows the general scheme that has been introduced before. First the additional invariants are loaded. Then the first call of the model validator that is satisfiable realizes the use case 'sanity check'. The only modification in the second call, which is unsatisfiable, is the negation of the equivalence formula. That means that in the broader search space, no counter example for the non-equivalence can be found and validity of the equivalence formula is assumed.

```
constraints -load equivalentImplication.invs

constraints -flags Dpt::dnm_primary_key                          -d
constraints -flags Dpt::dnm_primary_key_select                   -d
constraints -flags Dpt::dnm_primary_key_excluding                -d
constraints -flags Dpt::dnm_pk_EQ_dnm_pk_select_EQ_dnm_pk_excl      -d
mv -validate equivImplication.properties narrow

constraints -flags Dpt::dnm_primary_key                          -d
constraints -flags Dpt::dnm_primary_key_select                   -d
constraints -flags Dpt::dnm_primary_key_excluding                -d
constraints -flags Dpt::dnm_pk_EQ_dnm_pk_select_EQ_dnm_pk_excluding -d +n
mv -validate equivImplication.properties broad
```

We have used some shortcuts in the above listing to make the lines shorter and better readable: 'Dpt' stands for the class 'Department', 'dnm' for the attribute 'dname', and 'pk' for 'primary_key'.

4.5. Partitioning with Classifying Terms

Next up, we use classifying terms to understand the large example better. We will see how certain properties, given by OCL formulas, appear in system states and how they interact with each other. In this example, we use three classifying terms (Fig. 19) focusing on the classes `Department`, `Employee`, `Project` and the relations in between them to define three boolean properties that are further analyzed. The term `ManagersAreWorkers`

```
ManagersAreWorkers:
Project.allInstances()->forAll(p |
  ProjectWork.allInstances()->exists(pw | p.pname=pw.pname and
    p.mngfname=pw.fname and p.mnglname=pw.lname))

EmployeesOverlapProjects:
Employee.allInstances()->exists(e |
  ProjectWork.allInstances()->select(pw |
    e.fname=pw.fname and e.lname=pw.lname)->size() > 1)

DepartmentsOverlapProject:
Project.allInstances()->exists(p |
  Department.allInstances()->select(d |
    Employee.allInstances()->exists(e | e.dname=d.dname and
      ProjectWork.allInstances()->exists(pw | pw.fname=e.fname and
        pw.lname=e.lname and pw.pname=p.pname)))->size() > 1)
```

Figure 19: Three classifying terms for the Chen example.

connects the manager and employee relation by evaluating to true if the declared manager of a project is also an employee of it and false otherwise. `EmployeesOverlapProjects` shows the existence of employees that work on multiple projects. And the third term `DepartmentsOverlapProject` shows the existence of projects that are worked on by multiple departments. These properties stand as an example for any property one needs to analyze in a model. Remember that classifying terms can be any closed OCL formula.

The resulting eight system states of the verification are shown in Figs. 20 and 21. Only the relevant sections of the object diagram are shown wrt. the classifying terms which only directly affect these three classes. The fact that there are eight solutions shows that these properties are independent from one another since every combination of result values of the three boolean classifying terms is possible in the model ($2^3 = 8$).

The order in which these system states are generated is random, depending on the random seed of the SAT solver. In this scenario, the first four solutions in Fig. 20 all evaluate the third classifying term `DepartmentsOverlapProject` to false. They have in common that only one department object is present. In contrast, all system states in Fig. 21 have two department objects that all have at least one common project to fulfill the classifying term.

4.6. Partial Solution Completion

In Fig. 22 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 five failing OCL invariants. Among the failing invariants are the Employee primary key constraint and the foreign key constraint from Employee to Department. In this use case, the model validator modifies undefined attributes to defined ones, and through this, links for the derived foreign key association between Employee and Department can be established. The lower part of the figure

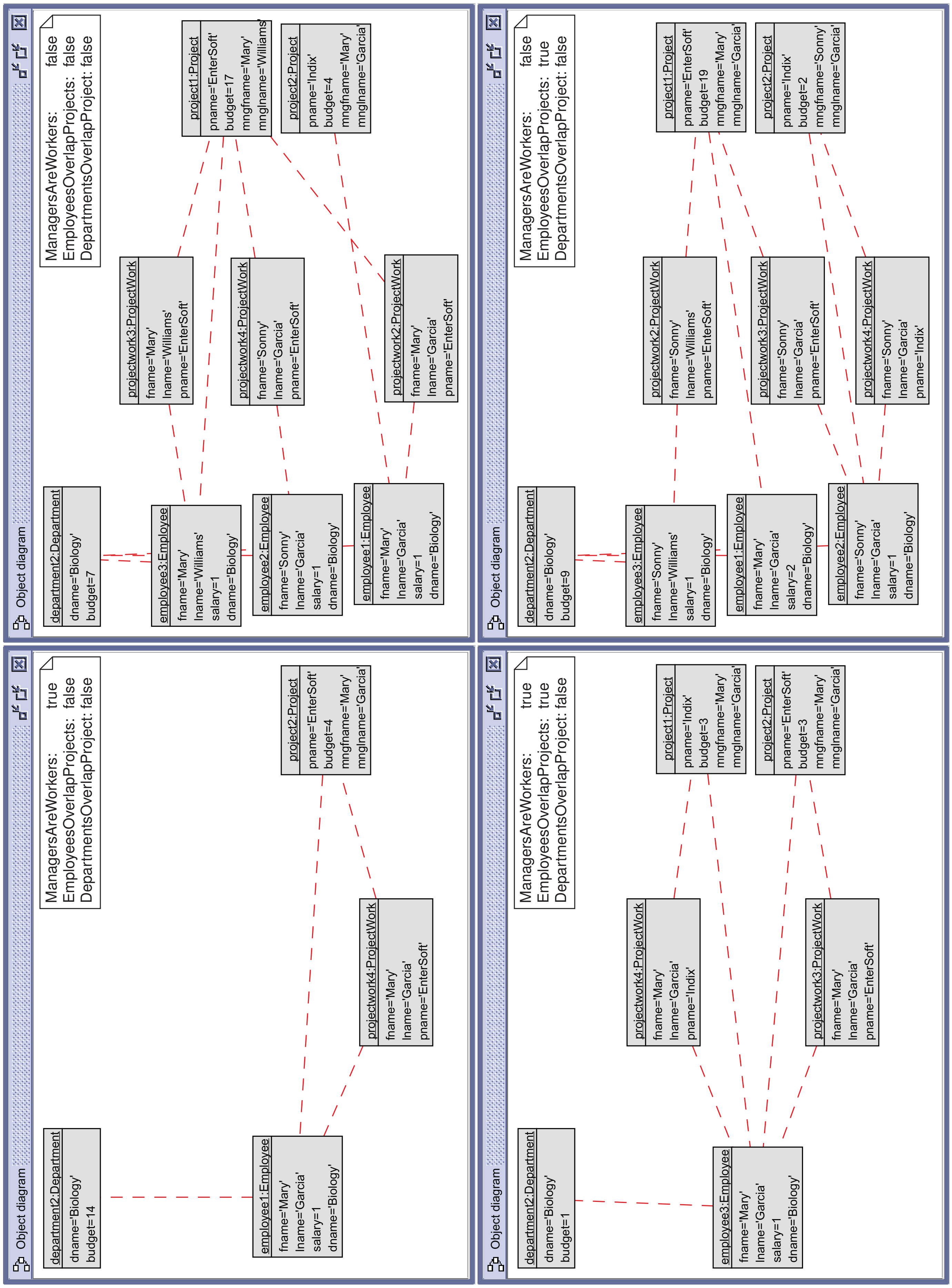
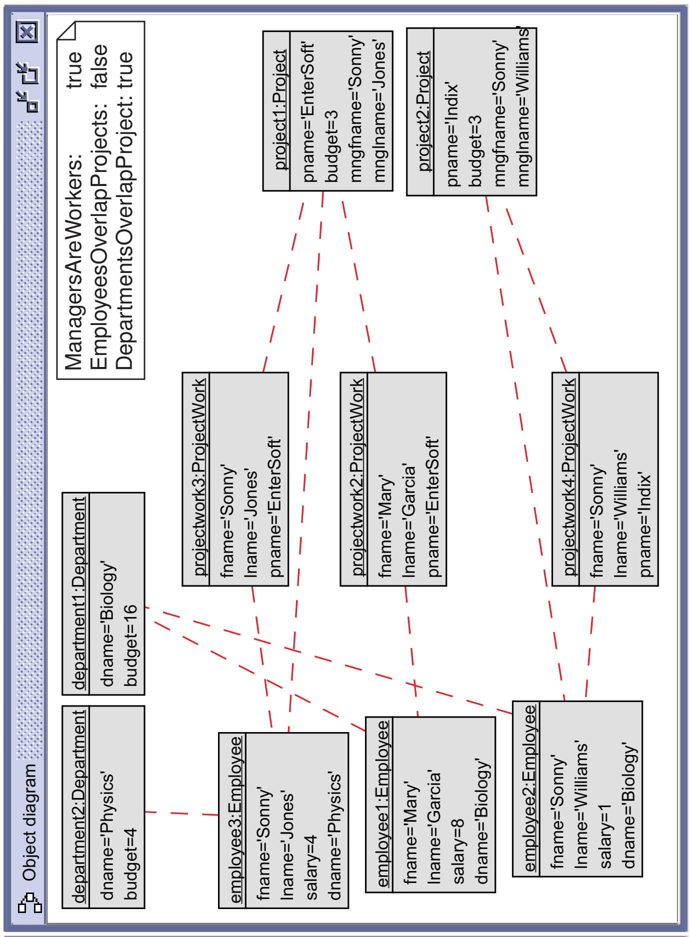
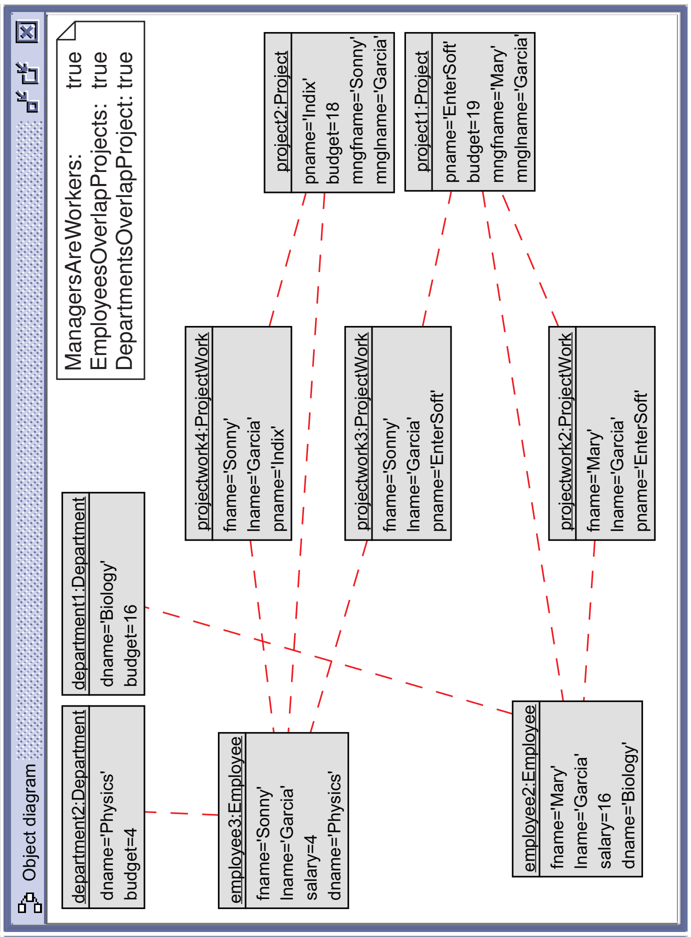
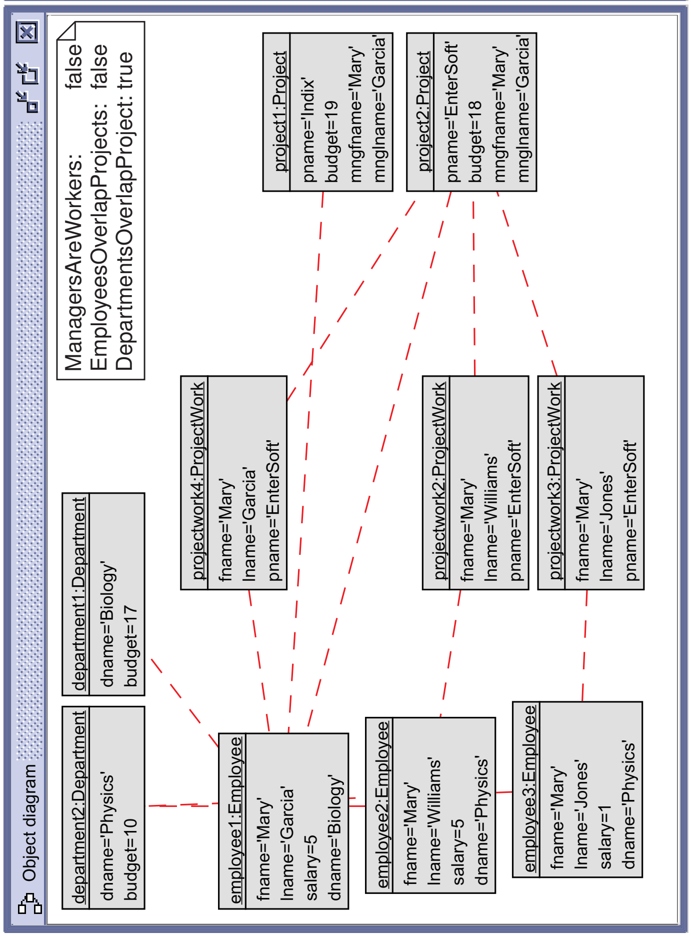


Figure 20: First four solutions for the Chen example showing all classifying term combinations.







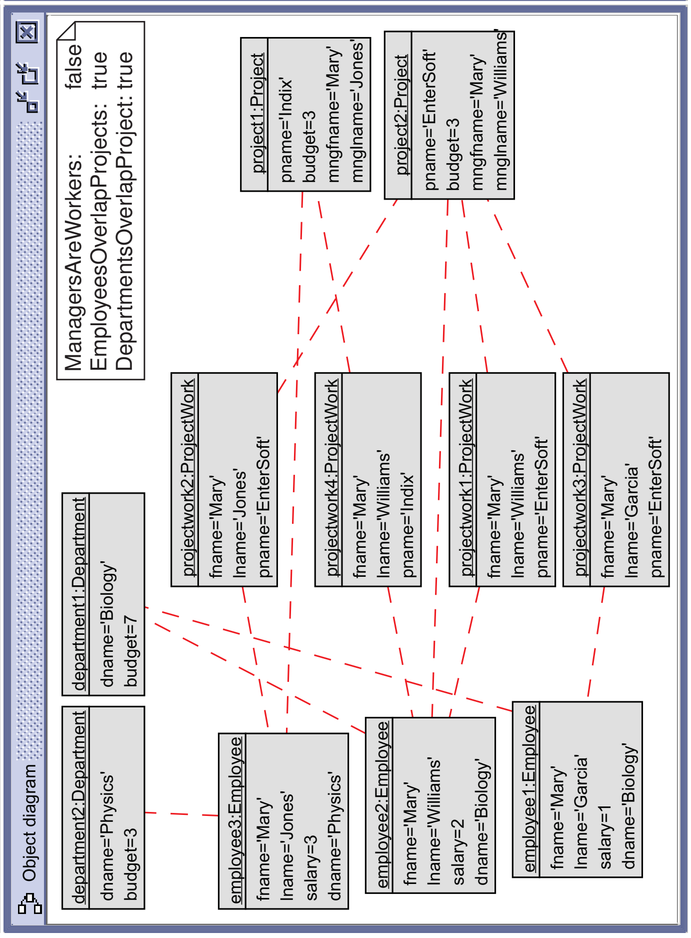
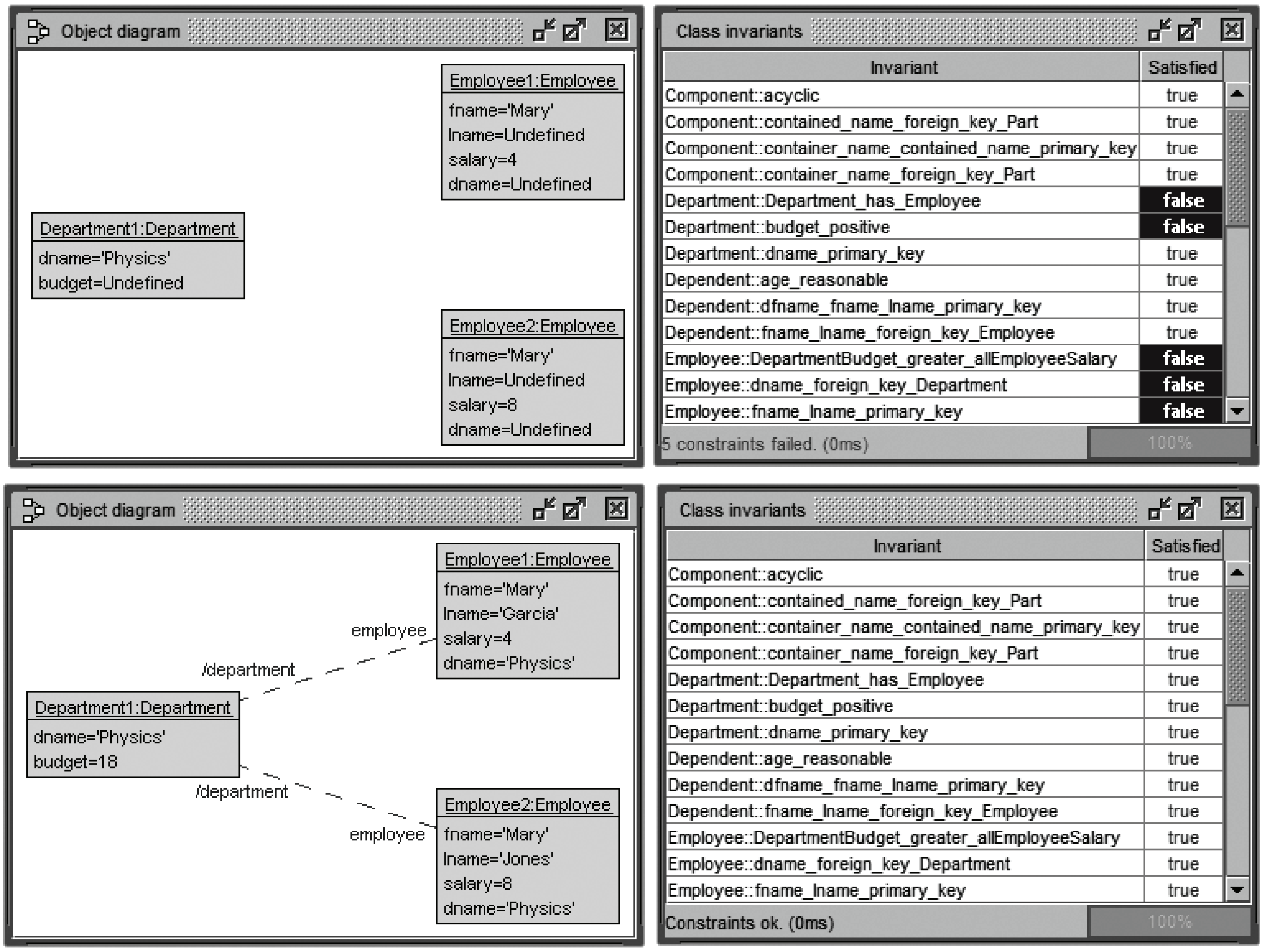


Figure 21: Second four solutions for the Chen example showing all classifying term combinations.



Invariant	Satisfied
Component::acyclic	true
Component::contained_name_foreign_key_Part	true
Component::container_name_contained_name_primary_key	true
Component::container_name_foreign_key_Part	true
Department::Department_has_Employee	false
Department::budget_positive	false
Department::dname_primary_key	true
Dependent::age_reasonable	true
Dependent::dfname_fname_lname_primary_key	true
Dependent::fname_lname_foreign_key_Employee	true
Employee::DepartmentBudget_greater_allEmployeeSalary	false
Employee::dname_foreign_key_Department	false
Employee::fname_lname_primary_key	false

5 constraints failed. (0ms) 100%

Invariant	Satisfied
Component::acyclic	true
Component::contained_name_foreign_key_Part	true
Component::container_name_contained_name_primary_key	true
Component::container_name_foreign_key_Part	true
Department::Department_has_Employee	true
Department::budget_positive	true
Department::dname_primary_key	true
Dependent::age_reasonable	true
Dependent::dfname_fname_lname_primary_key	true
Dependent::fname_lname_foreign_key_Employee	true
Employee::DepartmentBudget_greater_allEmployeeSalary	true
Employee::dname_foreign_key_Department	true
Employee::fname_lname_primary_key	true

Constraints ok. (0ms) 100%

Figure 22: 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. In order to satisfy the Employee key constraint at least two values for the attribute lname have to be available. And, in order to satisfy the invariant

```
context e:Employee inv DepartmentBudget_greater_allEmployeeSalary:
let d = Department.allInstances()->any(dname=e.dname) in
  e.salary <=
  d.budget div Employee.allInstances()->select(dname=d.dname)->size()
```

in the given partial object diagram with the Employee salary values 4 and 8, the Department budget values must include at least one value B satisfying $8 \leq$ B div 2 and with this $8 \cdot 2 = 16 \leq$ B. The value 8 is the highest salary and the value 2 is the number of Employee objects in the given partial object diagram. Thus the Department budget values must at least include the value 16 or higher values in order to achieve a valid object diagram.

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.

4.7. Lessons Learnt

(1) **The validation and verification use cases work properly with non-trivial examples**. The running example in this section is a non-trivial case with 10 classes, 12 derived associations and 29 invariants including primary keys and foreign keys constraints. The execution of the use cases in this example has shown results as expected.

(2) **The applicability of the use cases is reinforced.** The applicability of the model validator for model validation, verification and exploration was illustrated with a quite small example in Sec. 3. However by successfully experimenting with a larger example, the usefulness of these use cases is actually strengthened.

(3) **Exploring the model with object diagrams gives feedback and deeper understanding of a class diagram**. In most use cases, the model validator results include one or several generated object diagrams. Modelers can compare their exptectations with these generated system states. They can adjust the configuration, for example, in the solution interval exploration, partial solution completion or partitioning with classifying terms use cases, in order to obtain different system states. Exploring and comparing these object diagrams provides an in-depth understanding of the class diagram including its constraints and with this a better understanding of the modeled system.

5. Related Work

The transformation of UML and OCL into formal specifications for validation and verification is a widely considered topic. In [18], 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 [19] presents a translation of UML and OCL into first-order predicate logic to reason about models utilizing theorem provers. Similarly, *OCL2MSFOL*, a tool recently introduced in [20], can automatically reason about UML/OCL models through a mapping from UML/OCL to many-sorted first-order logic. The tool can check satisfiability of OCL invariants by applying SMT solvers. There are also other tools that validate model instances against UML and OCL constraints directly, like *DresdenOCL* [21]. Another similar tool is *UML-RSDS* [22], which allows for the validation of UML class diagrams. Several approaches rely on different technological cornerstones like logic programming and constraint solving [23], relational logic and Alloy [24], term rewriting with Maude [25] or graph grammars [26]. In contrast to the tool used in this work, which is based on the transformation of UML and OCL into relational logic [27], these approaches either do not support full OCL (e.g., higher-order associations [24] or recursive operation definitions [23] are not supported) or do not facilitate full OCL syntax checks [25]. Also, the feature to automatically scroll through several valid object models from one verification task is not possible in all of the above approaches. (Semi)-automatic proving approaches for UML class properties have been put forward on the basis of description logics [28], on the basis of relational logic and pure Alloy [24] using a subset of OCL, and in [29] focusing on model inconsistencies by employing Kodkod. A classification of model checkers with respect to verification tasks can be found in [30].

Verification tools use such transformations to reason about models and verify test objectives. *UMLtoCSP* [3] 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 [24], *UML2Alloy* is presented. A transformation of UML and OCL into Alloy [14] 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 [31]. 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 [4], OCL expressions are transformed into graph constraints and instance validation is performed by checking models against the graph constraints. Additionally, in [32], a transformation of OCL pre- and postconditions is presented for graph transformations.

The work in [5] 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 [6], 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 [33]. The authors suggest to use Alloy for the early modeling phase of development due to its better suitability for validation and verification. Additionally, FOML, an F-logic based language, is introduced in [34] as an approach for modeling, analyzing and reasoning about models.

UML together with OCL have been successfully 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 [35], 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 [36], central aspects of an industrial video conferencing system developed by Cisco have been studied. In [37], 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.

Finally, the USE model validator is to a certain degree the successor of the *ASSL* (A Snapshot Specification Language) [38]. 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 [39]. However, the explicit options for formulating the use cases are new, and we employ a new underlying validation engine (Kodkod). In [38] 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.

To summarize, a list of similar proposed model reasoning techniques is presented in the following table and compared with our eight use cases. To our knowledge, the last two use cases, i.e., equivalence implication and partitioning with classifying terms, have not been studied before. There are no similar techniques in the literature.

Table 3: A list of similar proposed model reasoning techniques

Use case	Use case name	Similar works/tools
1	Model Consistency	[3], [6], [10], [18], [20], [21], [24], [28], [29], [34]
2	Property Satisfiability	[3], [6], [10], [18], [20], [24], [34]
3	Constraint Implication	[3], [6], [10], [28]
4	Constraint Independence	[3], [6], [10], [28]
5	Solution Interval Exploration	[10]
6	Partial Solution Completion	[10]
7	Equivalence Implication	–
8	Partitioning With Classifying Terms	–

6. 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 eight use cases: model consistency, property satisfiability, constraint implication, constraint independence, solution interval exploration, partial solution completion, equivalence implication, and partitioning with classifying terms.

The techniques are useful from the early development phase to explore models up to the testing phase where model properties are verified. For example, partitioning with classifying terms has proven useful to present example instantiations of a model. Due to the exhaustive investigation, it is possible to quickly find unexpected corner cases that were not planned by the developer even when using only a small scope. For the interval solution exploration no concrete verification task is required and even new requirements can be found using this method. However, due to the high amount of possible instances, it is desirable to focus the results to a user defined area, minimizing the amount of solutions to a relevant set.

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 general available for developers, including non-experts on formal techniques, such high-level functions, like the options for invariant independence, are desirable for all use cases. We are currently implementing use case templates for the proposed use cases that can be instantiated for direct use. We are also planning support for developers in making formal relationships between configurations available, i.e., to check whether one configuration is narrower than another one. These features can then be provided as a high level API to use the USE tool and the model validator plugin as a backend for other tools, e.g., to use the features in Eclipse/Papyrus.

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

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

[2] B. Boehm, Software risk management, in: C. Ghezzi, J. A. McDermid (Eds.), Proc. 2nd European Software Engineering Conf. (ESEC 1989), Vol. 387 of LNCS, Springer, 1989, pp. 1–19.

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

[4] J. Winkelmann, G. Taentzer, K. Ehrig, J. M. Küster, Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars, ENTCS 211 (2008) 159–170.

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

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

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

[8] E. Torlak, D. Jackson, Kodkod: A Relational Model Finder, in: O. Grumberg, M. Huth (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007, Vol. 4424 of LNCS, Springer, 2007, pp. 632–647.

[9] F. Hilken, M. Gogolla, L. Burgueno, A. Vallecillo, Testing Models and Model Transformations using Classifying Terms, Software and Systems ModelingDOI 10.1007/s10270-016-0568-3.

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

[11] M. Gogolla, L. Hamann, F. Hilken, M. Sedlmeier, Checking uml and ocl model consistency: An experience report on a middle-sized case study, Tech. rep., TAP 2015, LNCS 9154, 129-136 (2015). URL `http://www.db.informatik.uni-bremen.de/publications/intern/GHHS2015.pdf`

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

[13] R. Clarisó, C. A. González, J. Cabot, Towards domain refinement for UML/OCL bounded verification, in: R. Calinescu, B. Rumpe (Eds.), Software Engineering and Formal Methods, Vol. 9276 of Lecture Notes in Computer Science, Springer, 2015, pp. 108–114.

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

[15] M. Gogolla, L. Hamann, F. Hilken, M. Kuhlmann, R. B. France, From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics, in: H. Fill, D. Karagiannis, U. Reimer (Eds.), Proc. Modellierung (MODELLIERUNG'2014), GI, LNI 225, 2014, pp. 273–288.

[16] F. Hilken, L. Hamann, M. Gogolla, Transformation of UML and OCL Models into Filmstrip Models, in: D. D. Ruscio, D. Varró (Eds.), Proc. 7th Int. Conf. Model Transformation (ICMT 2014), Springer, LNCS 8568, 2014, pp. 170–185.

[17] P. P. Chen, The Entity-Relationship Model - Toward a Unified View of Data, ACM Transactions on Database Systems 1 (1) (1976) 9–36.

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

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

[20] C. Dania, M. Clavel, OCL2MSFOL: A Mapping to Many-sorted First-order Logic for Efficiently Checking the Satisfiability of OCL Constraints, in: Proc. ACM/IEEE 19th Int. Conf. Model Driven Engineering Languages and Systems, MODELS '16, ACM, 2016, pp. 65–75.

[21] B. Demuth, C. Wilke, Model and Object Verification by Using Dresden OCL, in: Proc. Russian-German WS Innovation Information Technologies: Theory and Practice, 2009, pp. 687–690.

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

[23] J. Cabot, R. Clarisó, D. Riera, UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models using Constraint Programming, in: Proc. of ASE'07, 2007, pp. 547–548.

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

[25] M. Roldán, F. Durán, Dynamic Validation of OCL Constraints with mOdCL, ECEASST 44.

[26] K. Ehrig, J. M. Küster, G. Taentzer, Generating instance models from meta models, Software and System Modeling 8 (2009) 479–500.

[27] M. Kuhlmann, M. Gogolla, From UML and OCL to Relational Logic and Back, in: R. France, J. Kazmeier, R. Breu, C. Atkinson (Eds.), Proc. 15th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2012), Springer, Berlin, LNCS 7590, 2012, pp. 415–431.

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

[29] R. V. D. Straeten, J. P. Puissant, T. Mens, Assessing the Kodkod Model Finder for Resolving Model Inconsistencies, in: ECMFA, 2011, pp. 69–84.

[30] S. Gabmeyer, P. Brosch, M. Seidl, A Classification of Model Checking-Based Verification Approaches for Software Models, Proc. of the 1st VOLT Workshop (2013).

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

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

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

[34] M. Balaban, M. Kifer, Logic-based model-level software development with F-OML, in: J. Whittle, T. Clark, T. Kühne (Eds.), Proc. MODELS 2011, Vol. 6981 of LNCS, Springer, 2011, pp. 517–532.

[35] P. Ziemann, M. Gogolla, Validating OCL specifications with the USE tool: An example based on the BART case study, ENTCS 80 (2003) 157–169.

[36] S. Ali, M. Z. Z. Iqbal, A. Arcuri, L. Briand, A search-based OCL constraint solver for model-based test data generation, in: M. Núñez, R. M. Hierons, M. G. Merayo (Eds.), Proc. 11th Int. Conf. Quality Software QSIC, IEEE, 2011, pp. 41–50.

[37] OMG – Object Management Group, Unified Modeling Language Specification, Version 2.5 (June 2015).

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

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

Appendix: USE file and textual model for Chen example

```
model Chen

class Employee
attributes
  fname:String
  lname:String
  salary:Integer
  dname:String
end

class Department
attributes
  dname:String
  budget:Integer
end

class Dependent
attributes
  dfname:String
  age:Integer
  fname:String
  lname:String
end

class Project
attributes
  pname:String
  budget:Integer
  mngfname:String
  mnglname:String
end

class ProjectWork
attributes
  fname:String
  lname:String
  pname:String
end

class Supplier
attributes
  sname:String
  location:String -- original thirteen US states
end
```

```
/* location string enum - original thirteen US states - Delaware,
Pennsylvania, New Jersey, Georgia, Connecticut, Massachusetts Bay,
Maryland, South Carolina, New Hampshire, Virginia, New York, North
Carolina, and Rhode Island and Providence Plantations - DE, PA, NJ,
GA, CT, MA, MD, SC, NH, VA, NY, NC, RI */

class Part
attributes
  pname:String
  cost:Integer
operations
  contained():Set(Part)=
    Part.allInstances()->select(p|Component.allInstances->
      exists(c|c.containername=self.pname and c.containedname=p.pname))
  containedPlus():Set(Part)=self.contained()->closure(p|p.contained())
end

class ProjectPart
attributes
  projectname:String
  partname:String
end

class SupplierProjectPart
attributes
  suppliername:String
  projectname:String
  partname:String
end

class Component
attributes
  containername:String
  containedname:String
end

------------------------------------------------------------------------

association FK_Employee_Department between
  Employee   [*]
  Department [1] derived =
    Department.allInstances()->any(d|d.dname=self.dname)
end

association FK_Dependent_Employee between
  Dependent [*]
  Employee  [1] derived =
    Employee.allInstances()->
      any(e|e.fname=self.fname and e.lname=self.lname)
end

association FK_ProjectWork_Employee between
  ProjectWork [*]
  Employee    [1] derived =
    Employee.allInstances()->
      any(e|e.fname=self.fname and e.lname=self.lname)
end

association FK_Project_Manager between
  Project  [*]
  Employee [1] role manager derived =
    Employee.allInstances()->
      any(e|e.fname=self.mngfname and e.lname=self.mnglname)
end

association FK_ProjectWork_Project between
  ProjectWork [*]
  Project     [1] derived =
    Project.allInstances()->any(p|p.pname=self.pname)
end

association FK_ProjectPart_Project between
  ProjectPart [*]
  Project     [1] derived =
    Project.allInstances()->any(p|p.pname=self.projectname)
end

association FK_SupplierProjectPart_Project between
  SupplierProjectPart [*]
  Project             [1] derived =
    Project.allInstances()->any(p|p.pname=self.projectname)
end

association FK_SupplierProjectPart_Part between
  SupplierProjectPart [*]
  Part                [1] derived =
    Part.allInstances()->any(p|p.pname=self.partname)
end

association FK_SupplierProjectPart_Supplier between
  SupplierProjectPart [*]
  Supplier            [1] derived =
    Supplier.allInstances()->any(s|s.sname=self.suppliername)
end
```

```
association FK_ProjectPart_Part between
  ProjectPart [*]
  Part        [1] derived =
    Part.allInstances()->any(p|p.pname=self.partname)
end

association FK_Component_Container between
  Component [*] role containerComponent
  Part      [1] role containerPart derived =
    Part.allInstances()->any(p|p.pname=self.containername)
end

association FK_Component_Contained between
  Component [*] role containedComponent
  Part      [1] role containedPart derived =
    Part.allInstances()->any(p|p.pname=self.containedname)
end

------------------------------------------------------------------------

constraints

context e1,e2:Employee inv fname_lname_primary_key:
  e1.fname<>null and e1.lname<>null and
  e2.fname<>null and e2.lname<>null and
  ((e1<>e2) implies (e1.fname<>e2.fname or e1.lname<>e2.lname))
  -- pk01: C->forAll(c1,c2 | c1<>c2 implies c1.a<>c2.a)

  -- pk01: C->forAll(c1,c2 | c1<>c2 implies c1.a<>c2.a)
  -- pk02: C->isUnique(c | c.a)
  -- pk03: C->forAll(c1,c2 | c1.a=c2.a implies c1=c2)
  -- pk04: C->forAll(c1| C->select(c2| c1<>c2 and c1.a=c2.a)->isEmpty())
  -- pk05: C->forAll(c1| C->excl(c1)->select(c2| c1.a=c2.a)->isEmpty())
  -- pk06: C->forAll(c1| C->select(c2| c1<>c2 and c1.a=c2.a)->size()=0)
  -- pk07: C->forAll(c1| C->excl(c1)->select(c2| c1.a=c2.a)->size()=0)
  -- pk08: C->forAll(c1| not C->exists(c2| c1<>c2 and c1.a=c2.a))
  -- pk09: C->forAll(c1| C->forAll(c2| c1<>c2 implies c1.a<>c2.a))
  -- pk10: C->forAll(c1| C->forAll(c2| c1.a=c2.a implies c1=c2))

context e:Employee inv dname_foreign_key_Department:
  Department.allInstances()->one(d|d.dname=e.dname)

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

context Department inv dname_primary_key:
  dname<>null and Department.allInstances()->isUnique(dname)
  -- pk02: C->isUnique(c | c.a)

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

context d1,d2:Dependent inv dfname_fname_lname_primary_key:
  d1.dfname<>null and d1.fname<>null and d1.lname<>null and
  d2.dfname<>null and d2.fname<>null and d2.lname<>null and
  ((d1.dfname=d2.dfname and d1.fname=d2.fname and d1.lname=d2.lname)
    implies (d1=d2))
  -- pk03: C->forAll(c1,c2 | c1.a=c2.a implies c1=c2)

context d:Dependent inv fname_lname_foreign_key_Employee:
  Employee.allInstances()->one(e|e.fname = d.fname and e.lname=d.lname)

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

context p1:Project inv pname_primary_key:
  pname<>null and Project.allInstances()->
    select(p2|p2<>p1 and p2.pname=p1.pname)->isEmpty()
  -- pk04: C->forAll(c1| C->select(c2| c1<>c2 and c1.a=c2.a)->isEmpty())

context p:Project inv fname_lname_foreign_key_Employee:
  Employee.allInstances()->
    one(e|e.fname=p.mngfname and e.lname=p.mnglname)

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

context pw1:ProjectWork inv fname_lname_pname_primary_key:
  pw1.fname<>null and pw1.lname<>null and pw1.pname<>null and
  ProjectWork.allInstances()->excluding(pw1)->
    select(pw2|pw2.fname=pw1.fname and pw2.lname=pw1.lname and
               pw2.pname=pw1.pname)->isEmpty()
  -- pk05: C->forAll(c1| C->excl(c1)->select(c2| c1.a=c2.a)->isEmpty())

context pw:ProjectWork inv fname_lname_foreign_key_Employee:
  Employee.allInstances()->one(e|e.fname=pw.fname and e.lname=pw.lname)

context pw:ProjectWork inv pname_foreign_key_Project:
  Project.allInstances()->one(p|p.pname=pw.pname)

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

context s1:Supplier inv sname_primary_key:
  s1.sname<>null and Supplier.allInstances()->
    select(s2|s2<>s1 and s2.sname=s1.sname)->size()=0
  -- pk06: C->forAll(c1| C->select(c2| c1<>c2 and c1.a=c2.a)->size()=0)
-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

context p1:Part inv pname_primary_key:
  p1.pname<>null and
  Part.allInstances()->excluding(p1)->
    select(p2|p2.pname=p1.pname)->size()=0
  -- pk07: C->forAll(c1| C->excl(c1)->select(c2| c1.a=c2.a)->size()=0)

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

context pp1:ProjectPart inv project_name_part_name_primary_key:
  pp1.projectname<>null and pp1.partname<>null and
  (not ProjectPart.allInstances()->exists(pp2|
    pp2<>pp1 and pp2.projectname=pp1.projectname and
                 pp2.partname=pp1.partname))
  -- pk08: C->forAll(c1| not C->exists(c2| c1<>c2 and c1.a=c2.a))

context pp:ProjectPart inv project_name_foreign_key_Project:
  Project.allInstances()->one(p|p.pname=pp.projectname)

context pp:ProjectPart inv part_name_foreign_key_Part:
  Part.allInstances()->one(p|p.pname=pp.partname)

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

context sp1:SupplierProjectPart
  inv supplier_project_part_name_primary_key:
  sp1.suppliername<>null and sp1.projectname<>null and
  sp1.partname<>null and
  SupplierProjectPart.allInstances()->forAll(sp2| sp2<>sp1 implies
    (sp2.suppliername<>sp1.suppliername or
     sp2.projectname<>sp1.projectname or sp2.partname<>sp1.partname))
  -- pk09: C->forAll(c1| C->forAll(c2| c1<>c2 implies c1.a<>c2.a))

context sp:SupplierProjectPart inv supplier_name_foreign_key_Supplier:
  Supplier.allInstances()->one(s|s.sname=sp.suppliername)

context sp:SupplierProjectPart inv project_name_foreign_key_Project:
  Project.allInstances()->one(p|p.pname=sp.projectname)

context sp:SupplierProjectPart inv part_name_foreign_key_Part:
  Part.allInstances()->one(p|p.pname=sp.partname)

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

context c1:Component
  inv container_name_contained_name_primary_key:
```

```
  c1.containername<>null and c1.containedname<>null and
  Component.allInstances()->forAll(c2|
    (c2.containername=c1.containername and
     c2.containedname=c1.containedname) implies c2=c1)
  -- pk10: C->forAll(c1| C->forAll(c2| c1.a=c2.a implies c1=c2))

context c:Component inv container_name_foreign_key_Part:
  Part.allInstances()->one(p|p.pname=c.containername)

context c:Component inv contained_name_foreign_key_Part:
  Part.allInstances()->one(p|p.pname=c.containedname)

------------------------------------------------------------------------

context e:Employee inv salary_positive:
  e.salary>0

context Department inv budget_positive:
  budget>0

context d:Dependent inv age_reasonable:
  0<d.age and d.age<100

context p:Part inv cost_positive:
  p.cost>0

context c:Component inv acyclic:
  let p=Part.allInstances()->any(pname=c.containername) in
  p.contained().containedPlus()->excludes(p)

context e:Employee inv DepartmentBudget_greater_allEmployeeSalary:
  let d=Department.allInstances()->any(dname=e.dname) in
  e.salary
  <=
  d.budget div (Employee.allInstances()->select(dname=d.dname)->size())

context p:Project inv ProjectBudget_greater_PartCost:
  ProjectPart.allInstances()->select(projectname=p.pname)->
    forAll(pp|Part.allInstances()->one(p1|
      p1.pname=pp.partname and p1.cost<p.budget))

------------------------------------------------------------------------
```