# Contradiction Analysis for Inconsistent Formal Models

Nils Przigoda[1] Robert Wille[1,2] Rolf Drechsler[1,2]

[1]Group for Computer Architecture, University of Bremen, 28359 Bremen, Germany

[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany

{przigoda,rwille,drechsle}@informatik.uni-bremen.de

***Abstract*—Modeling languages such as UML or SysML in combination with constraint languages such as OCL allow for an abstract description of a system prior to its implementation. But the resulting system models can be highly non-trivial and, hence, errors in the descriptions can easily arise. In particular, too strong restrictions leading to an inconsistent model are common. Motivated by this, researchers and engineers developed methods for the validation and verification of given formal models. However, while these methods are efficient to detect the existence of an inconsistency, the designer is usually left alone to identify the reasons for it. In this contribution, we propose an automatic method which efficiently determines reasons explaining the contradiction in an inconsistent UML/OCL model. For this purpose, all constraints causing the contradiction are comprehensibly analyzed. By this, the designer is aided during the debugging of his/her model.**

## I. Introduction

Modeling languages, such as the *Unified Modeling Language* (UML) [1] and the *Systems Modeling Language* (SysML) [2], [3] as some of the best-known representatives, received much attention in the past. They allow for a precise description of a system at a high level of abstraction before precise implementation steps are performed. For this purpose, UML provides appropriate models which hide precise implementation details while being expressive enough to specify a complex system. Within UML, the *Object Constraint Language* (OCL) [4] enables the enrichment of the respective models by textual constraints which add further information to the description. Using OCL, it is possible to define invariants which restrict valid system states or to describe further properties as well as relations between the specified components[1].

The resulting models may be composed of numerous different components with various relations, dependencies, or constraints and usually lead to non-trivial descriptions where errors can easily arise. This may lead to an inconsistent description, where certain UML and OCL constraints contradict each other. Then, no valid system state can be derived from the model anymore – obviously a serious design error. Detecting such flaws in early stages of the development is an important task since correcting an inconsistent UML/OCL model is easier than fixing a resulting error in the actual implementation.

Motivated by this, researchers and engineers developed corresponding methods and tools for the validation and verification of system descriptions given as a formal model. As an example, the *UML-based Specification Environment* (USE) [5] provides well-established methods that can be applied e. g. to automatically generate test cases for the respective UML/OCL models [6]. Besides that, researchers began to exploit formal methods for the verification of UML/OCL models. Approaches based on theorem provers like PVS [7], HOL-OCL/Isabelle [8], and KeY [9] have been applied for this purpose. They are capable of checking very large models, but often require a strong formal background of the designer. As a consequence, researchers started to investigate the application of fully automatic proof engines including methods based on constraint programming (CSP) [10], [11], [12], description logic [13], [14], the modeling language Alloy based on relational logic [15], [16], or Boolean satisfiability (SAT) [17], [18].

These approaches are particularly helpful in order to determine a valid system state of the model. Furthermore they are useful, even if the description is inconsistent, since these approaches help to detect the existence of a contradiction. But, in such cases they do not provide any further details on the reason for the contradiction. Hence, the designer is left alone to debug the model, i. e. to identify the UML/OCL constraints that caused the contradiction.

In this contribution, we propose an automatic method which aids the designer in this process. A methodology is introduced which efficiently determines so-called reasons of the contradiction, i. e. a subset of all UML/OCL constraints of a model which forms a contradiction and, hence, explains the inconsistency. For this purpose, we exploit the formal proof techniques which have already successfully been applied to initially check for the existence of a contradiction. The resulting problem formulation is extended so that UML/OCL constraints can be disabled in order to derive a valid system state from the model. By exhaustively analyzing all possible combinations that got disabled in order to allow a valid system state, the desired reasons can automatically be determined. Previously proposed approaches for debugging UML/OCL models (discussed in detail in Section III-B) only provided approximations of the reasons. In contrast to this, the solution proposed in this work determines *all* minimal reasons explaining the problem.

The remainder of this work is structured as follows: The next section briefly introduces the terminology used in the following. Afterwards, a precise problem formulation and discussion of related work is provided in Section III. The general concept of the proposed solution is then sketched and illustrated in Section IV, before details on its implementation and evaluation are given in Section V. Finally, the paper is concluded in Section VI.

## II. Preliminaries

In order to keep the paper self-contained, this section provides a brief review on UML/OCL and introduces the notation used to describe the respective models and system states.

*Definition 1: A* model $\mathcal{M} = (\mathcal{C}, \mathcal{R})$ *is a tuple of* classes $\mathcal{C}$ *and* relations $\mathcal{R}$ *(also known as* associations*). A* class $c \in \mathcal{C}$ *contains attributes and operations. A relation* $r = (c_1, c_2, (l, u))$ *consists of two classes* $c_1$ *and* $c_2$ *from* $\mathcal{C}$*. The tuple* $(l, u)$ *represents the* lower and the upper bound*, i. e. each instance of* $c_1$ *shall be connected with at least* $l$ *but at most* $u$ *instances of* $c_2$*. Such a pair of bounds is called* UML constraints *in the following. The lower bound is an arbitrary natural number, while the upper bound is either a positive natural number or infinity. Besides that, the model can additionally be enriched by textual constraints which can be provided in OCL and are denoted as* invariants*. The set* $\mathcal{I}$ *represents all OCL invariants of a model. Each invariant* $i \in \mathcal{I}$ *belongs to a class* $c \in \mathcal{C}$ *and defines an* OCL constraint*.*

*Example 1: Consider the model given in Figure 1 which consists of two classes,* A *and* C*. Both are connected by two relations. The first relation* $r_1$ *states that each object of* C

[1]In this paper, UML/OCL is used as modeling language, but the presented approach can easily be transferred to other modeling languages, e. g. SysML.

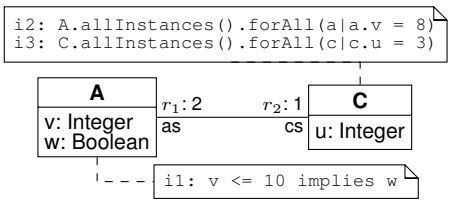
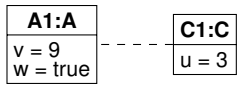
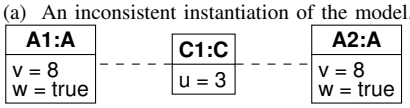



Figure 1: A model example




(a) An inconsistent instantiation of the model.




(b) An consistent instantiation of the model.

Figure 2: Two model instantiations

*is connected with exactly two objects of* A*. Due to the fact that the lower and upper bound are equal, only one bound is notated. The second relation states that each object of* A *is connected with exactly one object of* C*. Additionally, three OCL invariants are provided which restrict the values of the attributes.*

Note that, in the following, we mainly make use of the convention which denotes sets by upper case letters and single elements by lower case letters.

Furthermore, for the sake of simplicity, we restrict ourselves to binary associations. This restriction is not too strong. In fact, it has been shown that models containing $n$-ary associations can be mapped into an semantically equivalent model solely composed of binary associations by adding some invariants to the affected classes [19]. Modeling languages such as EMF [20] do not support $n$-ary associations at all.

*Definition 2: An instantiation of a model* $\mathcal{M} = (\mathcal{C}, \mathcal{R})$ *with a set of invariants* $\mathcal{I}$ *is called* system state*. Furthermore, a system state is called* valid*, if all UML and OCL constraints are satisfied. If at least one of the different constraints is not satisfied, the instantiation is called* invalid*. A valid instantiation must be also composed of at least one object of any class. If a valid instantiation of a model exists, then the model is called* consistent*.*

*Example 2: Consider again the model shown in Figure 1. The instantiation of the model presented in Figure 2a is invalid, because the restriction given by the relation* $r_1$ *is not satisfied for the object* C1*, which is an instance of class* C*. In fact, it is not possible to determine a valid instantiation composed of one object of class* A *and one object of class* C*, because the object of* C *can never be connected with two different objects of class* A*. Additionally, the invariant* $i_2$ *is also not satisfied, because the attribute* $v$ *of object* A1 *is assigned 9. But in order to satisfy* $i_2$*, attribute* $v$ *is assigned 8 for all objects of class* A*.*

*In contrast, a valid instantiation of the model is shown in Figure 2b. Here, object* C1 *is connected with exactly two objects of class* A*, namely object* A1 *and object* A2*. Each object of class* A *is connected with only one object of class* **C***, i. e. the UML constraints* $r_1$ *and* $r_2$ *are satisfied. However, a valid instantiation must also satisfy the remaining three constraints given by the invariants. Invariant* $i_1$ *requires that, for every object of class* A*, where the attribute* $v$ *is smaller or equal to 10, the Boolean attribute* $w$ *is set to true. As* $v$ *is 8 for both instances and* $w$ *is true, invariant* $i_1$ *is satisfied. The invariant* $i_2$ *(*$i_3$*) is also satisfied, because the attribute* $v$ *(*$u$*) of all object instances of class* A *(class* B*) are set to 8 (3). Therefore a valid system state has been determined and this also shows that the model provided in Figure 1 is consistent.*

## III. Problem Formulation and Related Work

In this section, the design task of debugging inconsistent model descriptions is introduced and motivated. For this purpose, a model is provided which works as running example throughout the remainder of this paper. Besides that, we briefly review previously proposed solutions to this problem and discuss their potential for improvement which motivated the work at hand.

### A. Debugging Erroneous Models

Although system models based on UML/OCL rely on a high level of abstraction (hiding a significant amount of implementation details), the respective descriptions can be highly non-trivial and, hence, complex to comprehend. Consequently, errors in the descriptions can easily arise and are often hard to detect. This may lead e. g. to an inconsistent description from which no valid instantiation can be derived. Motivated by this, researchers and engineers developed various methods aiming at the verification and validation of UML/OCL models (see e. g. [5], [10], [16], [17], [18]).

These methods are very efficient in detecting whether all descriptions in a UML/OCL model are consistent or not. For consistent models, they often even determine a valid system state of the model and, by this, provide a so-called witness confirming that an instantiation of the model indeed is possible. However, in cases where the description is inconsistent, methods such as introduced in [5], [10], [16], [17], [18] only report that a contradiction exists – without any details on the reason(s) for this. In these cases, the designer is left alone to debug the model, i. e. to identifiy the reasons for the contradiction.

*Example 3: Consider the UML/OCL model as shown in Figure 3 which is used as running example in the remainder of this paper. The model describes a system composed of four classes including several attributes which are constrained by several UML constraints (i. e. relations together with their multiplicities) as well as OCL constraints (i. e. invariants). Additionally, it is assumed that 2, 5, 3, and 7 objects shall be instantiated from the classes* A*,* B*,* C*, and* D*, respectively. However, although maybe not obvious at a first glance, this model is highly over-constrained. In fact, the following contradictions prohibit a valid instantiation of this model*[2]*:*

1) *A trivial contradiction is inherent in the invariant* $i_5$*. Here, the logical expression by itself is not valid.*
2) *Somewhat harder to debug are contradictions which span over several invariants. In the model, this is the case for the invariants* $i_3$*,* $i_1$*, and* $i_2$*. The invariant* $i_3$ *enforces all attributes* v *of each instance of class* A *to be set to the value* 8*. Adding invariant* $i_1$*, the attribute* w *always has to be* true*. However, invariant* $i_2$ *requires this attribute to be* false *for exactly one connected instance of class* A*.*
3) *In a similar fashion, UML constraints may lead to a contradiction. This is the case for the association between the classes* A *and* C*. Since 2 objects are derived from Class* A *and 3 objects are derived from class* C*, the multiplicities of this relation (represented by* $r_1$ *and* $r_2$*) are violated.*
4) *Finally, contradictions may have its origins in the combination of both, UML and OCL constraints. For example, this is the case for the combination of the invariants* $i_4$ *and* $i_7$ *and the relation* $r_7$*. Both, the invariants and the relation, enforce restrictions on the number of links from instantiations of class* D *to instantiations of class* C *which are in violation to each other.*

Without any additional help, debugging an inconsistent UML/OCL description is a cumbersome and time-consuming task. Hence, methods that automatically approximate or even exactly determine reasons for a contradiction are desired. A

[2]Note that the model was built to serve as a proper example including various types of contradictions which may occur in practice.

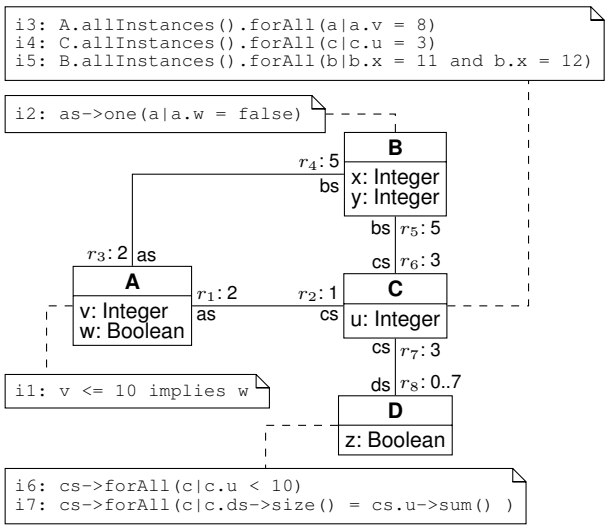



Figure 3: Running example

*reason* is thereby a subset of all UML/OCL constraints which forms a contradiction and, hence, have to be considered by the designer for debugging. The definition for a reason is given as follows.

*Definition 3: A UML/OCL model is* over-constrained *or* inconsistent *if the conjunction of all UML constraints such as relations including their multiplicities specified in $\mathcal{R}$ as well as OCL constraints such as invariants specified in $\mathcal{I}$ are not satisfiable and, thus, no valid instantiation is possible. Then, a* reason $R$ *for the contradiction is a non-empty set of constraints $R \subseteq \mathcal{R} \cup \mathcal{I}$ such that the conjunction of all constraints $c_i \in R$ forms a contradiction, i. e.*

$$\bigwedge_{c_i \in R} c_i$$

*is equivalent to* $0$.

*Example 4: As discussed in Example 3, the model shown in Figure 3 is inconsistent due to four reasons, namely: 1. $R_1 = \{i_5\}$, 2. $R_2 = \{i_1, i_2, i_3\}$, 3. $R_3 = \{r_1, r_2\}$, and 4. $R_4 = \{i_4, i_7, r_7\}$.*

How to efficiently determine reasons for an inconsistent UML/OCL model is considered in this work.

### B. Previously Proposed Solutions

Determining reasons for inconsistent UML/OCL models has been considered before. For example, approaches presented in [21] and [22] provide respective solutions. Both utilize the same solving engine which is already applied in the first place to check whether a contradiction exists.

Torlak et al. have proposed an approach [21] which relies on a technique called *unsatisfiable core extraction* (see also e. g. [23], [24]). First, they check whether a given model is consistent, i. e. free of a contradiction. For that purpose, they translate the declarative UML/OCL description into a propositional formula which is given to a solving engine (e. g. a SAT solver). In case of a contradiction, the solver will prove that no satisfying solution and, by this, no valid instantiation exists. Then, unsatisfiable core extraction determines a sub-set of the propositional formula (a so-called unsatisfiable core) which still is contradictory[3]. This subset is further optimized until a *minimal* core results. Since each expression in the formula represents invariants or relations of the UML/OCL model, a minimal reason for the contradiction can be determined from the resulting subset. However, a main problem of this approach is that only one contradiction is determined in each turn.

[3]Many solving engines such as MATHSAT and Z3 already provide unsat core determination as "in-house" functionality.

*Example 5: Applied to the running example, the approach presented in [21] may deliver the reason $R_1$, i. e. the information that $i_5$ is self-contradictory. Even after fixing this error the designer is still left alone with a contradictory UML/OCL model. In fact, the designer has to explicitly invoke this approach four times until all contradictions have been identified and fixed. Considering that unsat core extraction is a computationally expensive process (i. e. requires significant run-times), this provides potential for further improvement.*

In [22], a complementary approach has been proposed. Here, the propositional formula is enriched by additional logic allowing to *disable* certain constraints. The number of constraints to be disabled is thereby restricted by an integer $k$.

At the beginning, $k$ is fixed to 1 (i. e. one invariant may be disabled). If this still leads to a contradiction (checked by the solving engine), $k$ is increased. This procedure is repeated until the solving engine determines the first solution. Then, all solutions for the current value $k$ are determined. This eventually leads to so-called *contradiction candidates*, i. e. a set of UML/OCL constraints which, once disabled, would lead to a contradiction-free model. However, while this may pin-point the designer to possible problems in the model, this approach only provides limited explanations of the contradiction.

*Example 6: Applying the approach from [22] will lead to the contradiction candidates $r_1, r_2, r_7, i_1, i_2, i_3, i_4, i_5$, and $i_7$. While this includes all contradictory constraints, no relation between them is provided. In fact, this result only represents the union of the actual reasons, i. e. $R_1 \cup R_2 \cup R_3 \cup R_4$. The designer is left alone figuring out their relations to each other.*

Overall, while the approaches proposed before already offer proper solutions which aid the designer in debugging contradictory UML/OCL models, they still offer room for improvement. Either only a single reason is determined or all contradictory UML/OCL descriptions are provided at once. In this work, we propose an alternative, which determines *all* minimal reasons and additionally groups them according to their contradictions.

## IV. Proposed Approach

In this section, we describe the proposed approach for an advanced analysis of contradictions in UML/OCL models. The general idea is thereby based on the previously proposed solutions described above, i. e. we rely on a solving engine which already has been applied to identify the existence of a contradiction and we enrich the corresponding propositional formula.

More precisely, given a UML model $\mathcal{M} = (\mathcal{C}, \mathcal{R})$ with OCL constraints $\mathcal{I}$, we assume the model has been checked for the existence of a contradiction by transforming it into the propositional formula

$$f_{\text{con}} = \Phi(\mathcal{M}) \wedge \bigwedge_{r \in \mathcal{R}} [\![r]\!] \wedge \bigwedge_{i \in \mathcal{I}} [\![i]\!], \text{where} \qquad (1)$$

- $\Phi(\mathcal{M})$ is a propositional sub-formula representing all UML components in a system state such as objects, attribute values, and links,
- $[\![r]\!]$ is a propositional sub-formula representing the given UML relation $r \in \mathcal{R}$, and
- $[\![i]\!]$ is a propositional sub-formula representing the given OCL invariant $i \in \mathcal{I}$.

For this purpose, existing approaches for UML/OCL validation and verification such as [5], [10], [16], [17], [18] can be utilized. If the respectively applied solving engine cannot determine a valid assignment for this instance, it has been proven that the model is inconsistent and the designer must have a look at all constraints in order to fix the model.

In order to provide the designer with the desired reasons, we exploit the idea from [22], i. e. we enrich the corresponding propositional formula so that UML relations or OCL invariants

Table I: Possible solutions derived from the running example

| $r_1$ | $r_2$ | $r_3$ | $r_4$ | $r_5$ | $r_6$ | $i_6$ | $r_8$ | $i_1$ | $i_2$ | $i_3$ | $i_5$ | $i_4$ | $i_7$ | $r_7$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| - | 1 | - | - | - | - | - | - | - | 1 | - | 1 | - | 1 | - |
| - | 1 | - | - | - | - | - | - | - | 0 | 1 | 1 | - | 1 | - |
| - | 1 | - | - | - | - | - | - | - | - | 1 | 1 | 1 | 0 | - |
| - | 1 | - | - | - | - | - | - | 1 | - | 0 | 1 | 1 | 0 | - |
| - | 1 | - | - | - | - | - | - | 1 | 0 | 0 | 1 | - | 1 | - |
| 1 | 0 | - | - | - | - | - | - | - | 1 | - | 1 | - | 1 | - |
| - | 1 | - | - | - | - | - | - | - | 1 | - | 1 | 0 | 0 | 1 |
| - | 1 | - | - | - | - | - | - | 0 | 1 | 0 | 1 | 1 | 0 | - |
| - | 1 | - | - | - | - | - | - | - | 0 | 1 | 1 | 0 | 0 | 1 |
| - | 1 | - | - | - | - | - | - | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | - | - | - | - | - | - | - | 1 | - | 1 | 1 | 0 | - |
| 1 | 0 | - | - | - | - | - | - | - | 1 | - | 1 | 0 | 0 | 1 |
| 1 | 0 | - | - | - | - | - | - | - | 0 | 1 | 1 | - | 1 | - |
| 1 | 0 | - | - | - | - | - | - | - | 0 | 1 | 1 | - | 0 | 1 |
| 1 | 0 | - | - | - | - | - | - | - | 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | - | - | - | - | - | - | 1 | 0 | 0 | 1 | - | 1 | - |
| 1 | 0 | - | - | - | - | - | - | 1 | 0 | 0 | 1 | - | 0 | 1 |
| 1 | 0 | - | - | - | - | - | - | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| at least one 1 | | – | | | | | | at least one 1 | | | **1** | at least one 1 | | |

may be disabled. More precisely, the formula from Eq. 1 is extended to

$$f'_{\text{con}} = \Phi(\mathcal{M}) \wedge \bigwedge_{r \in \mathcal{R}} (s_r \vee [\![r]\!]) \wedge \bigwedge_{i \in \mathcal{I}} (s_i \vee [\![i]\!]), \qquad (2)$$

where $s_c$ is a new free variable corresponding to a constraint $c$. However, in contrast to [22], we do not restrict the total number $k$ of constraints to be disabled. Besides this, we also need to extend Definition 3 as we are interested in minimal reasons.

*Definition 4: A reason $R$ of an inconsistent model – as defined in Definition 3 – is called* minimal *if no subset $R' \subsetneq R$ exists such that*

$$\bigwedge_{c \in R'} c_i$$

*is already equivalent to* 0.

Now, passing the formula from Eq. 2 to a solving engine allows to investigate how enabling or disabling UML/OCL constraints keeps or does not keep the model contradiction-free. For example, an assignment determined by the solving engine such as

$$s_{r_1} = 1,\ s_{r_2} = 1,\ s_{r_3} = 1,\ s_{r_4} = 1,\ s_{r_5} = 1,$$
$$s_{r_6} = 1,\ s_{r_7} = 1,\ s_{r_8} = 0,\ s_{i_1} = 1,\ s_{i_2} = 1,$$
$$s_{i_3} = 1,\ s_{i_4} = 1,\ s_{i_5} = 1,\ s_{i_6} = 1, \text{ and } s_{i_7} = 1.$$

allows the conclusion that disabling all UML/OCL constraints except $r_8$ would lead to a valid instantiation of the model. Obviously, this alone does not significantly help in identifying the reasons for a contradiction. But observing *all* such possibilities indeed provides a proper basis for a comprehensive analysis[4].

To illustrate this, Table I lists all possible assignments to the respective $s_c$-variables that have been obtained by applying the formula from Eq. 2 to the running example. More precisely, a 1 denotes that the respective constraint has been disabled, while a 0 denotes that it has been enabled. A – represents a *don't care*, i. e. the case that, for the considered assignment, the value of the corresponding select variable (stating whether the respective constraint is enabled or disabled) does not matter. This allows to represent all solutions in a compact fashion. The row at the bottom summarizes some of the properties which are exploited for the analysis and discussed next.

In fact, some very substantial conclusions can be drawn from Table I. For example, it can be observed that invariant $i_5$ is disabled in *all* solutions. Hence, this invariant must be self-contradictory; otherwise at least one solution where all constraints except $i_5$ are disabled should exists. This is summarized by a **1** in the bottom row of the column $i_5$. Similarly, it can be concluded that the relations $r_3, r_4, r_5, r_6, r_8$ and the invariant $i_6$ are never part of a minimal reason; otherwise the value of the corresponding $s_c$*-variable* could not arbitrarily be switched. This is summarized by a – in the bottom row of the respective columns.

Another conclusion may be drawn from the first two columns of Table I. Here, it is evident that at least one of the two constraints is disabled in all solutions. This constitutes an indication that $r_1$ and $r_2$ are part of a reason; otherwise at least one solution should exists in which both are enabled. In a similar fashion, this holds for $i_1$, $i_2$, and $i_3$ as well as $i_4, i_7$, and $r_7$. All these cases are summarized by *at least one 1* in the bottom row of the respective columns.

All these observations represent single snapshots and properties which can be exploited to determine the reasons of the contradiction. This eventually leads to the main idea of the proposed approach for contradiction analysis: Using the formula from Eq. 2 and an efficient solving engine, all possible combinations of constraints are iteratively checked starting with the smallest ones. For each combination, the $s_c$-variables are pre-assigned in a fashion that the respective combinations are enabled. If no valid model can be derived assuming this assignment, then a reason for a contradiction has been determined. By excluding all supersets from already determined reasons, minimality is guaranteed.

## V. Implementation and Evaluation

This section describes the implementation of the proposed contradiction analysis based on the ideas discussed above. A big drawback of a straight-forward implementation is thereby that the analysis relies on iteratively checking all possible combinations of constraints and, hence, is rather time-consuming. Accordingly, an optimized approach is additionally proposed which addresses this problem. Both implementations are evaluated by means of the running example.

### A. Straightforward Approach

*1) Implementation:* The implementation of the proposed ideas is explained with the help of the pseudo code depicted in Figure 4. The algorithm starts with a proper representation of the formula from Eq. 2 ($f'_{con}$) and the two sets $R$ and $S$ which are both initialized to the empty set. $R$ stores all reasons that are found, while $S$ is used to save all $s_c$-variables that are passed to the detailed analysis later.

Then, the number of combinations to be considered is simplified based on the observations discussed in Section IV. In other words, for each constraint it is checked if it is self-contradictory (line 8) or not. If yes, the respective constraint is added to the set of reasons $R$ (line 10). Each of these checks is conducted using the applied solving engine together with an extended version of the formula from Eq. 2, i. e. to check if $s_c$ is 1 for all solutions, the conjunction $f'_{con} \wedge (s_c = 0)$ is carried out. If no valid solution can be determined from that, it can be concluded that $s_c$ is always 1. Thus, $c$ becomes a reason.

If such a check fails (line 11), then the respective constraint $c$ can be a part of a contradiction caused by the conjunction of $c$ with one or more other constraints. Thus, $c$ is passed to the detailed analysis by inserting the respective $s_c$ into $S$ (line 13).

The next step is to calculate all possible assignments for the select variables which makes the model consistent. For this purpose the extended formula from Eq. 2 is passed to the solving engine and allSAT (line 15) will return all the desired assignments $A$. This can be done naivly by just blocking the last found solution and ask the solving engine for another solution until the solver engine recognizes that there are no more solutions.

Finally, the detailed analysis for all elements in $S$, representing the remaining constraints, is performed (line 17 to 24). First, the power set $\mathcal{P}(S)$ of $S$ is created resulting in all subsets (i. e. combinations) of constraints considered for detailed analysis. Note that we exclude the empty set as well as all sets which only contain one element from the power

[4]In a similar fashion, contradictions in inputs for constraint-based random simulation are analyzed [25].

**ContradictionAnalysis**
**Input:** $f'_{con}$
1: // initialization
2: // set of all minimal contradiction reasons
3: $R \leftarrow \emptyset$
4: // $s_c$ variables for detailed analysis
5: $S \leftarrow \emptyset$
6: // first step, simple analysis
7: **for all** $c \in \mathcal{R} \cup \mathcal{I}$ **do**
8: **if** $(f'_{con} \wedge s_c = 0) \equiv 0$ **then**
9: // $c$ is self-contradictory
10: $R \leftarrow R \cup \{\{c\}\}$
11: **else**
12: // $c$ is selected for detailed analysis
13: $S \leftarrow S \cup \{s_c\}$
14: // calculating all possible assignments
15: $A \leftarrow \text{allSAT}(f'_{con})$
16: // detailed analysis
17: **for all** $X \in \mathcal{P}(S)$ **do**
18: // from the smallest to the largest
19: **if** $\exists X' \subset X : X' \in R$ **then**
20: // ensure minimality
21: **continue**
22: **if** $\nexists\, a \in \mathrm{A} : \forall\, s_c \in X \;:\; a(s_c) = 0$ **then**
23: // reason found
24: $R \leftarrow R \cup \{X\}$
25: **return** $R$;

Figure 4: Straightforward implementation

set (this is already covered by the simplifications before). Furthermore, the elements of the power set are ordered according to their cardinality. Then, for each subset $X$ (i. e. for each combination), the conjunction of the respective constraints is tested for a contradiction. If $X$ is a contradiction, then it is not possible to get an assignment $a \in A$, where all corresponding select variables are assigned 0. Thus the non-existence of such an assignment implies that $X$ is a contradiction (line 22). To ensure minimality, each contradiction test of a subset $X$ is only carried out if no subset of $X$, i. e. $X' \subset X$, has already been detected as a reason for a contradiction $X' \in R$ (line 19-21).

In summary, the presented contradiction analysis procedure computes all minimal reasons $R$ that explain the contradiction of the considered UML/OCL model.

*2) Evaluation:* In order to evaluate the algorithm shown in Figure 4, we realized this approach as an Eclipse plugin using both, Java and Xtend. As solving engine, we utilized *Satisfiability Modulo Theories* (SMT) [26] which has already been successfully applied in order to verify and validate UML/OCL models [18].

Using the resulting tool, we were successful in *automatically* determining all minimal reasons explaining the contradiction in the running example as discussed in Example 4. By this, engineers are significantly aided in the debugging of contradictory UML/OCL models. In fact, the resulting reasons are more precise and complete compared to the results of previously proposed solutions as discussed in Section III-B. On the contrary, following this straight-forward scheme is computationally expensive: For the running example, a total of 9408 possible assignments have been analyzed requiring a total run-time of approx. 8 CPU hours[5]. In order to address this, an optimized implementation is proposed next.

### B. Optimized Approach

*1) Implementation:* The huge number of assignments to be analyzed represents a crucial obstacle of the proposed approach. This is caused by the simple determination of all assignments in Line 15 of the algorithm from Figure 4: After the determination of a satisfying assignment, a so-called blocking constraint is added which prevents the solving engine from determining the same solution again. By this, all possible assignments are solely obtained. However, for the purposes considered here, explicitly determining *all* assignments is not necessarily required. In fact, a *reduced set of assignments* is sufficient.

*Definition 5: Let $a \in A$ be one assignment which was obtained by the solving engine and $S$ the set of all $s_c$-variables. Furthermore, let $S_a$ be a subset of $S$ ($S_a \subseteq S$) composed of those $s_c$-variables which are assigned 1 in $a$. Then $a' \in A$ is called an* overset[6] *of $a$, iff $S_a \subseteq S_{a'}$. Blocking the partial assignment including only the variables assigned 1 rather than the complete assignment $a$, prevents the solving engine not only from obtaining the same assignment again but also all of its oversets. This results in a* reduced set of assignments *(denoted $A'$).*

Considering the reduced set of assignments significantly shrinks the total number of assignments to be analyzed. At the same time, the reduced set of assignments is still sufficient to determine all minimal reasons for the contradiction.

*Lemma 1:* Checking not all possible assignments $a \in A$ but only the assignments $a' \in A'$ of the reduced set, still allows the determination of all minimal reasons for a contradiction.

*Proof 1:* As it is obivious that $A'$ is not emtpy, it suffices to prove that no smaller reasons can wrongly be discovered by the algorithm. This will be proven by contradiction. Let $R_j$ be a minimal reason to be detected by the algorithm using the complete set of assignments $A$. Now assume that a smaller reason $R_{j'} \subsetneq R_j$ would erroneously be detected by relying on the reduced set $A'$.

Then, there would be an assignment $a \in A$ in the complete set such that, for all constraints in the smaller reason $R_{j'}$, the corresponding $s_c$-variables are all assigned 0, i. e. $\forall\, c_i \in R_{j'} : a(s_{c_i}) = 0$ (otherwise, this reason would have been detected using the complete set $A$). Then, we can distinguish two cases:

1) This assignment is in $A'$ (i. e. $a \in A'$): Then, the algorithm would have not detected $R_{j'}$ as reason since it violates the corresponding condition in line 22.
2) This assignment is not in $A'$ (i. e. $a \notin A'$): Then, $a$ must have been blocked before by another assignment $a' \in A'$ with $S_{a'} \subsetneq S_a$. But such an assignment can only have a smaller number of $s_c$-variables set to 1. Consequently, all $s_c$-variables in $a$ assigned 0 will also be assigned 0 in $a'$. Again, the algorithm would have not detected $R_{j'}$ as reason since it violates the corresponding condition in line 22.

*Remark 1:* Consider again the minimal reason $R_j = \{c_1, c_2, \ldots, c_k\}$. In an optimal case, the algorithm adds only $k$ blocking constraints. In the worst case, up to $2^k$ blocking constraints are needed.

The actual number of blocking constraints added for $R_j$ depends only on the used solving engine and its internal learning techniques which depend on heuristic methods.

In order to improve such an heuristic behaviour, the number of disabled constraints is restricted as explained in [22]. At first all assignments with only one disabled constraint are obtained by the solver engine, if they exist. Then the number of disabled constraints is increased by one and the solver engine is called again. Note that, in contrast to [22], we will not stop after the first bunch of assignments. Instead, we call the solving engine for every possible number of disabled constraints and, by this, we obtain a minimal list of assignments. Now, by blocking only oversets of determined assignments and a structured search for every possible number of disabled constraints, we can still ensure that the collected assignments are a reduced set of assignments as described in Definition 5.

According to these results, the algorithm shown in Figure 4 can easily be optimized by modifying Line 15. Here, instead of an `allSAT` scheme which determines a complete set of assignments, simply a reduced set of assignments according

[5]The experiment has been carried out on a Intel i5 with 2.6 GHz cores and 16 GB memory using a 3.11 kernel Linux.

[6]We are using the word *overset* to avoid confusions with the term *superset*.

Table II: Experimental results

| Model | $\lvert\mathcal{C}\rvert$ | $\lvert\mathcal{R}\rvert$ | $\lvert\mathcal{I}\rvert$ | $\lvert\mathcal{O}\rvert$ | #R | Sizes | #solve | Run-time |
|---|---|---|---|---|---|---|---|---|
| CivStat | 1 | 2 | 6 | 5 | 2 | 4 | 11 | < 1 s |
| CivStat | 1 | 2 | 6 | 5 | 2 | 4, 4 | 11 | < 1 s |
| CivStat | 1 | 2 | 7 | 5 | 2 | 4, 4, 4 | 13 | 3.3 s |
| CivStat | 1 | 2 | 6 | 5 | 2 | 4, 4 | 11 | < 1 s |
| CivStat | 1 | 2 | 6 | 5 | 2 | 4 | 11 | < 1 s |
| CivStat | 1 | 2 | 7 | 5 | 2 | 4, 4, 4 | 13 | 2.7 s |
| CivStat | 1 | 2 | 7 | 5 | 2 | 2, 3 | 11 | 2.2 s |
| CivStat | 1 | 2 | 7 | 5 | 2 | 2, 3 | 11 | 2.2 s |
| CivStat | 1 | 2 | 7 | 5 | 2 | 3, 3 | 11 | < 1 s |
| CarRental2 | 9 | 14 | 7 | 5 | 1 | 3, 3 | 11 | 2.5 s |
| RunningExample | 4 | 8 | 7 | 7 | 4 | 1, 2, 3, 3 | 34 | 5.3 s |

*Legend:*
$\lvert\mathcal{C}\rvert$: Number of classes $\lvert\mathcal{R}\rvert$: Number of relations $\lvert\mathcal{I}\rvert$: Number of invariants
$\lvert\mathcal{O}\rvert$: Maximal number of considered object instantiations per class
#R: Number of reasons sizes: a list with the sizes of each reason
#solve: the number of sat solver calls

to Definition 5 is obtained. This is sufficient to perform the following detailed analysis but, at the same time, significantly reduces the number of analysis steps.

*2) Evaluation:* In order to evaluate the optimized approach, the refined algorithm has been realized as an Eclipse plugin as well. Here, the determination of assignments as described above has been adjusted, while the rest of the algorithm remains identical.

Also the improved tool enabled the automatic determination of all minimal reasons explaining the contradiction in the running example as discussed in Example 4. But this time, only 18 rather than 9408 assignments were sufficient for this purpose. As the solving engine had to be invoked respectively fewer times, this led to a significant reduction in the required run-time: Instead of more than 8 hours, the reasons have been determined in just a few seconds. By this, the designer is clearly aided during the debugging of his/her model.

In order to further evaluate the optimized solution, the Eclipse plugin has also been tested with the well-known models from the *USE* package [5]. For this purpose the CivStat model has been edited nine times such that all nine models became inconsistent and different. Furthermore, the (inconsistent) CarRental2 model has been considered which is a variation of the (consistent) CarRental (both are provided by the *USE* package as well).

Table II summarizes the obtained results. The first columns provide the name of the respective models followed by their number of classes ($\lvert\mathcal{C}\rvert$), their number of relations ($\lvert\mathcal{R}\rvert$), their number of invariants ($\lvert\mathcal{I}\rvert$), their maximal number of considered object instantiations per class ($\lvert\mathcal{O}\rvert$), the number of detected reasons (#R), the sizes of the detected reasons (Sizes), the number of solver calls (#solve), and the over-all run-time. Note that the number of solver calls includes all calls with an `UNSAT` as result and thus, is not equal to the number of assignments which have to be analyzed in order to get all minimal reasons.

The numbers clearly show that the optimization of the proposed solution can not only determine all minimal reason but also does this in feasible run-time for the considered models. The detection of one and more reasons for all considered models does never require more than some seconds.

## VI. Conclusion

In this contribution, we considered the automatic determination of reasons explaining the contradiction in an inconsistent UML/OCL model. Most of the solutions for the validation and verification of UML/OCL descriptions allow efficient checks which show the existence of an inconsistency. But after that, the designer is usually left alone to identify the reasons for it. Previously proposed solutions for debugging of UML/OCL models only provided either a single reason or all contradictory UML/OCL descriptions at once.

We proposed an alternative which determines *all* minimal reasons and additionally groups them according to their contradictions. For this purpose, all possible combinations of UML/OCL constraints are analyzed. It has been shown that a naive implementation of the proposed idea provides the desired result but requires a significant amount of run-time. Accordingly, an optimization has additionally been introduced addressing this issue. The resulting solution is capable of efficiently providing the designer with a set of minimal reasons which aids him/her during the debugging of inconsistent UML/OCL models.

## Acknowledgements

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SPECifIC under grant no. 01IW13001, the German Research Foundation (DFG) within the Reinhart Koselleck project under grant no. DR 287/23-1, and a research project under grant no. WI 3401/5-1, as well as the Siemens AG.

## References

[1] J. Rumbaugh, I. Jacobson, and G. Booch, Eds., *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., 1999.
[2] G. Martin and W. Müller, *UML for SOC Design*. Springer, 2005.
[3] T. Weilkiens, *Systems Engineering with SysML/UML:Modeling, Analysis, Design*. Morgan Kaufmann, 2007.
[4] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise modeling with UML*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
[5] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 27–34, 2007.
[6] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, Independence and Consequences in UML and OCL Models," in *Tests and Proof*, 2009, pp. 90–104.
[7] M. Kyas, H. Fecher, F. S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons, and H. Kugler, "Formalizing UML Models and OCL Constraints in PVS," *Electronic Notes in Theoretical Computer Science*, vol. 115, pp. 39–47, 2005.
[8] A. D. Brucker and B. Wolff, "A Proposal for a Formal OCL Semantics in Isabelle/HOL," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science, vol. 2410. Springer, 2002, pp. 99–114.
[9] B. Beckert, R. Hähnle, and P. H. Schmitt, *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.
[10] J. Cabot, R. Clarisó, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," in *ICST Workshops*, 2008, pp. 73–80.
[11] T. Mancini, "Finite satisfiability of UML class diagrams by constraint programming," in *Description Logics*, 2004.
[12] H. Malgouyres and G. Motet, "A UML model consistency verification approach based on meta-modeling formalization," in *Symposium on Applied Computing*. ACM, 2006, pp. 1804–1809.
[13] D. Berardi, D. Calvanese, and G. De Giacomo, "Reasoning on UML class diagrams," *Artif. Intell.*, vol. 168, no. 1, pp. 70–118, 2005.
[14] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers, "Using Description Logic to Maintain Consistency between UML Models," in *The Unified Modeling Language, Modeling Languages and Applications*, 2003, pp. 326–340.
[15] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.
[16] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A Challenging Model Transformation," in *Int'l Conf. on Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, vol. 4735. Springer, 2007, pp. 436–450.
[17] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *TACAS*, ser. Lecture Notes in Computer Science, O. Grumberg and M. Huth, Eds., vol. 4424. Springer, 2007, pp. 632–647.
[18] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using Boolean satisfiability," in *Design, Automation and Test in Europe*, 2010, pp. 1341–1344.
[19] M. Gogolla and M. Richters, "Expressing UML Class Diagrams Properties with OCL," in *Object Modeling with the OCL*, ser. Lecture Notes in Computer Science, T. Clark and J. Warmer, Eds., vol. 2263. Springer, 2002, pp. 85–114.
[20] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.
[21] E. Torlak, F. S.-H. Chang, and D. Jackson, "Finding Minimal Unsatisfiable Cores of Declarative Specifications," in *Formal Methods*, 2008, pp. 326–341.
[22] R. Wille, M. Soeken, and R. Drechsler, "Debugging of inconsistent UML/OCL models," in *Design, Automation and Test in Europe*, 2012, pp. 1078–1083.
[23] J. Zhang, S. Shen, J. Zhang, W. Xu, and S. Li, "Extracting minimal unsatisfiable subformulas in satisfiability modulo theories," *Comput. Sci. Inf. Syst.*, vol. 8, no. 3, pp. 693–710, 2011.
[24] R. Gershman, M. Koifman, and O. Strichman, "An approach for extracting a small unsatisfiable core," *Formal Methods in System Design*, vol. 33, no. 1–3, pp. 1–27, 2008.
[25] D. Große, R. Wille, R. Siegmund, and R. Drechsler, "Contradiction Analysis for Constraint-based Random Simulation," in *Forum on specification and Design Languages*, 2008, pp. 411–416.
[26] S. Ranise and C. Tinelli, "The satisfiability modulo theories library, SMT-LIB," 2008. [Online]. Available: http://www.smt-lib.org