Debugging of Inconsistent UML/OCL Models

Robert Wille* Mathias Soeken* Rolf Drechsler*§
*Institute of Computer Science, University of Bremen
28359 Bremen, Germany
{rwille,msoeken,drechsle}@informatik.uni-bremen.de
§Cyber-Physical Systems, DFKI GmbH
28359 Bremen, Germany
rolf.drechsler@dfki.de

***Abstract*—While being a de-facto standard for the modeling of software systems, the *Unified Modeling Language* (UML) is also increasingly used in the domain of hardware design and hardware/software co-design. To ensure the correctness of the specified systems, approaches have been presented which automatically verify whether a UML model is consistent, i.e. free of conflicts. However, if the model is inconsistent, these approaches do not provide further information to assist the designer in finding the error.**

In this work, we present an automatic debugging approach which determines contradiction candidates, i.e. a small subset of the original model explaining the conflict. These contradiction candidates aid the designer in finding the error faster and therefore accelerate the whole design process. The approach employs different satisfiability solvers as well as different debugging strategies. Experimental results demonstrate that, even for large UML models with up to 2500 classes and constraints, the approach determines a very small number of contradiction candidates to be inspected.

I. Introduction

In the recent years, the *Unified Modeling Language* (UML) has been widely accepted as the standard language for modeling and documentation of software systems [1]. With the ongoing trend towards the design at the *Electronic System Level* (ESL), UML also offers promising applications in the domain of hardware design and hardware/software co-design [2]. The desired system can initially be specified at a high level of abstraction, before precise implementation steps are performed. Therefore, UML provides appropriate models which hide concrete implementation details while being expressive enough to specify a complex system. Additionally, the *Object Constraint Language* (OCL) [3] can be applied to refine a UML model with textual constraints describing further properties and relations between the specified components.

During the specification of a complex system, numerous different components with various relations, dependencies, and constraints are defined. This leads to a nontrivial description where errors can easily arise. Therefore, researchers started the investigation of appropriate verification techniques. Approaches based on enumeration [4], theorem provers [5], [6], or automatic proof engines [7], [8], [9] have been introduced for this purpose. They enable the detection of design flaws already in early stages of the system's development, which is especially crucial when considering shortening time-to-market demands. However, even if these approaches help to detect the existence of an error, they provide no support for determining the source of this flaw. That is, in case of an erroneous specification, the designer has to debug the respective model manually, which often is a complex and time consuming task.

In this paper, approaches for debugging of inconsistent UML/OCL models are presented. A model (specification) is inconsistent, if it is contradictory in itself and, thus, no valid system state (instantiation) can be generated. In particular, static descriptions, i.e. class diagrams and their system states, are considered. Design flaws in such descriptions can result in a contradiction typically caused by (1) wrong UML constraints, i.e. errors in the specification of the relation between the respective components, or (2) wrong OCL constraints, i.e. errors in the additional properties specified by the textual OCL constraints.

A two-stage method is proposed which narrows down the number of possible candidates for the contradiction. First, it is determined whether the error occurs either because of contradictory UML constraints or contradictory OCL constraints. Afterwards, the source of error is further confined leading to *contradiction candidates*, i.e. a set of components of the specification explaining the error. They can be used to pinpoint to the error source. Solvers for *Linear Integer Arithmetic* (LIA) [10] and *Bit-Vector logic* (BV) [11] are utilized in the proposed approach.

Experiments demonstrate that using the proposed approach the reason for a contradiction can be narrowed down to a small set of components. The approach is applicable even for very large UML/OCL models with more than 2500 classes and constraints.

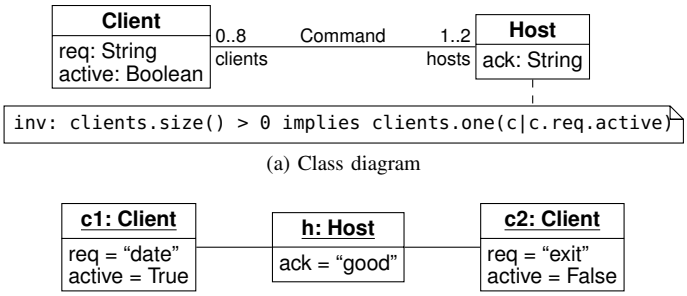
The remainder of this paper is structured as follows. UML/OCL models are briefly reviewed in the next section. Section III motivates the considered problem, while Section IV introduces the general flow of the proposed approach. The respective approaches determining the contradiction candidates are described in detail in Section V and evaluated in Section VI. Finally, related work is discussed and conclusions are drawn in Section VII and Section VIII, respectively.

II. Models and System States

When modeling systems, one differentiates between the *model* and its *system states*. The model describes the structure of the system on an abstract level. The precise instantiation of the model is called system state, and for one particular model, several system states may exist. The main components of a model are *classes* and *associations*. Classes describe what information can be handled within the modeled system and how this information is structured. Inside a class, *attributes* define the single data elements. Associations describe the relation between classes, where each association is annotated by multiplicities which specify its type, i.e. a *one-to-one*, a *one-to-many*, or a *many-to-many* relation.

Since the associations in a model are restrictive, they are also called *UML constraints*. More complex constraints can be defined using the OCL, which offers textual constructs to express complex properties of classes, their attributes, and their relations. One possible use case of OCL constraints are invariants, which are attached to a class in a model.

978-3-9810801-8-6/DATE12/©2012 EDAA



(a) Class diagram

(b) Object diagram

Fig. 1. UML class diagram and object diagram

In the following, we denote a model by $\mathcal{M}$, its set of classes by $\mathcal{C}$, and invariants are referred to as $\mathcal{I}$. Since this paper addresses the usage of UML models for specifications, the terms *model* and *specification* are synonymously interchanged in the remainder of this paper.

A model can be visualized by a UML class diagram. An example of such a class diagram is given in Fig. 1(a), which consists of the two classes *Client* and *Host*. The *Client* class has two attributes *req* and *active* which represents a request sent by a client and the state of the client, respectively. Further, the *Host* class has one attribute *ack*, which represents an acknowledge returned by the respective host. Both classes are set into relation by an association *Command*, which expresses that each host can be connected to up to eight clients and that each client has to be connected to one or two hosts. Finally, one OCL invariant expresses, that if there are clients connected to a host, then exactly one of them has to be active. The OCL offers further operators for the construction of constraints. For a detailed consideration, the reader is referred to [3].

As mentioned above, a precise instantiation of a model is called a *system state*. It consists of *objects*, which are precise instances of a class and *links*, which are precise instances of an association. Analogously to a class diagram, a system state can be visualized by an *object diagram*. Fig. 1(b) shows an object diagram representing a valid system state derived from the model shown in Fig. 1(a).

In this paper, debugging of inconsistent UML models is considered, where consistency is defined by means of valid system states.

Definition 1 (Validity of system states): A system state is called *UML-valid* if its links comply the UML constraints implied by the model's associations. Similarly, a system state is called *OCL-valid* if it satisfies all OCL invariants. If a system state is both *UML-valid* and *OCL-valid*, a system state is called *valid*.

Based on the definition of validity, the definition of consistency can be formulated.

Definition 2 (Consistency of a model): A model is *consistent*, if it is possible to create a non-empty valid system state. Analogously, a model is *UML-consistent* (*OCL-consistent*), if there exists a non-empty UML-valid (OCL-valid) system state. A model is *inconsistent*, if it is not consistent.

Checking whether a model is consistent is of importance, since for an inconsistent model, each attempt to create a system state fails a priori.

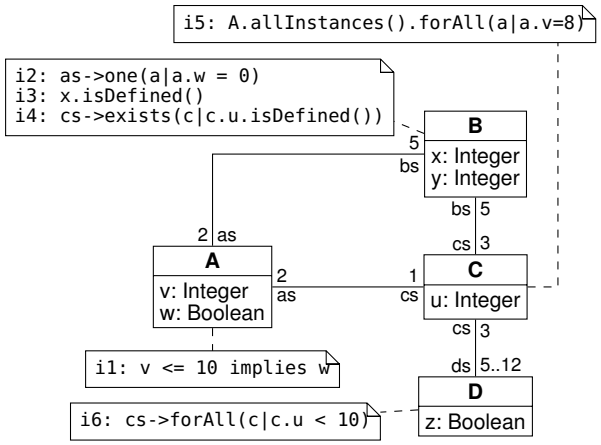


Fig. 2. Running example

III. Problem Formulation

Designing complex systems is a nontrivial task where errors can easily arise. In particular, the application of additional constraints for refinement purposes is crucial. While constraints enable to render the system in a more precise way, they can lead to an over-constrained specification. That is, too many or wrong constraints being applied make it impossible to generate a valid system state from the specification.

Considering UML models, over-constrained specifications are typically caused by (1) wrong UML constraints, i.e. associations in a model, or (2) wrong OCL constraints, i.e. textual properties further constraining the data and relation between elements.

Example 1: Throughout this paper, the abstract model shown in Fig. 2 is used as a running example. It consists of four classes, four associations, and six OCL invariants. This model is neither UML-consistent nor OCL-consistent, which is not directly evident at first glance.

The first flaw results from the associations between the classes *A*, *B*, and *C*. For each object of class *C*, two objects of class *A* are required. Further, with each two objects of class *A*, another five objects of class *B* are required which imply the existence of three objects of class *C*. Summing up, for each object of class *C*, three objects of class *C* are needed, which is contradictory. This conflict can be fixed by changing the multiplicity 1 at the association between class *A* and class *C* to 3.

The other contradiction in the specification is "hidden" in three OCL constraints. The invariant i_5 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*. This leads to a contradiction and, therefore, the model is also not OCL-consistent. The conflict can be fixed by changing invariant i_5 such that it does not call `forAll` on all instances of class *A*, but on the connected objects of class *A* only. That is, invariant i_5 should be changed to `as->forAll(a|a.v = 8)`.

In order to detect inconsistent specifications as early as possible, verification approaches are already applied in the early design stages. For this purpose, several methods have

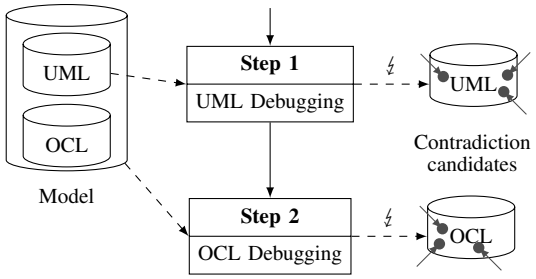


Fig. 3. General flow

been introduced in the past (see e.g. [5], [8], [6], [9], [4], [7], [12]). Using the UML model along with all OCL constraints as an input, they try to generate a non-empty, valid system state. If this is possible, the existence of such a system state is witnessed by an object diagram.

However, if no such witness can be generated, the specification has been proven to be over-constrained. In order to identify the reason and fix the error, the designer has to debug the specification – often a complicated and cumbersome task, which results in a manual and time consuming procedure. In the worst case, all classes and constraints have to be inspected. While this might be feasible for the simple model discussed in Example 1, it becomes highly inefficient for specifications composed of several hundreds of classes and constraints.

In contrast, the source of a contradiction can often be limited to very few components. For example, the contradiction discussed in Example 1 was simply caused by a single association and three OCL constraints. Having this information, large parts of the specification can be ignored in order to debug a contradictory UML/OCL model. Motivated by this, in this paper we address the following question:

> *How can we automatically detect an as small as possible subset of a contradictory UML/OCL model explaining the non-existence of a valid system state?*

Different methods are introduced narrowing down the set of possible reasons for contradictions in a given inconsistent specification. First, it is determined which type of contradiction occurred, i.e. whether either the UML constraints or the OCL constraints should be inspected in detail. Afterwards, more precise contradiction candidates are generated. A contradiction candidate is a component of the model (e.g. an association or an OCL invariant) explaining the absence of a valid system state. By means of these contradiction candidates, the number of components which have to be manually considered is significantly reduced, and accordingly, the time spent on debugging is decreased.

IV. General Flow

In order to obtain candidates explaining the contradiction, a two-stage debugging flow is proposed. In the following, the respective steps are briefly introduced by means of Fig. 3 and the precise methods are described in detail in the next section.

Starting with an inconsistent UML model, first it is determined whether the reason for the contradiction is due to the UML constraints of the model. For this step, the OCL invariants of the model are not required. In case the model is UML-inconsistent, candidates in terms of associations responsible for the contradiction are returned, which can be used by the designer to fix the problem. In contrast, if it has been shown that the model is UML-consistent, a large amount of contradiction candidates can be excluded for further consideration. Then, the OCL constraints have to be the reason for the contradiction. Therefore, a method for OCL debugging is executed in the second step. Here, a subset of all OCL invariants is determined, whose deactivation leads to a consistent system state. This set is then returned as contradiction candidates to be inspected by the designer.

Using this flow, the designers are pinpointed to UML or OCL constraints explaining the contradiction in the model. With this information, large parts of the model can be classified to be irrelevant to debug the error.

V. Debugging Methods

While the previous section sketched the general idea of the proposed approach, in the following the respective methods are introduced in detail. Following the flow shown in Fig. 3, debugging of UML constraints is described first, before OCL debugging is considered.

A. Debugging UML Constraints

UML constraints, i.e. associations along with multiplicities, define the relations between classes and, therefore, they restrict the number of objects instantiated from a class. As an example, the association between class *A* and class *C* depicted in Fig. 2 enforces that the number of objects derived from class *A* always has to be twice the number of objects derived from class *C*. Implications of such constraints may lead to dependencies which eventually cannot be satisfied any longer – in particular, if more than two classes are involved. As an example, consider again the contradiction caused by the three associations between the classes *A*, *B*, and *C* in Fig. 2.

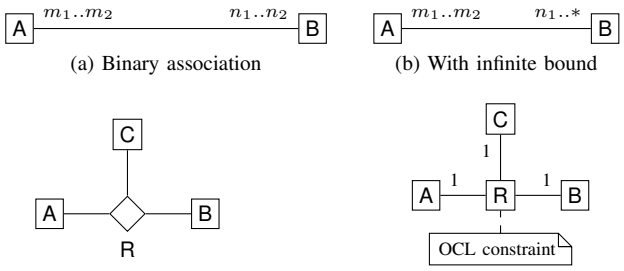
In order to determine the reason for a contradictory UML model, it has to be checked whether it is possible to instantiate the appropriate number of objects from each class, so that all restrictions enforced by the UML constraints are adhered to. This is formulated as an instance of the satisfiability problem encoded using *Linear Integer Arithmetic* (LIA) [10]. If the resulting formulation is satisfiable, a number of object instantiations can be derived satisfying all UML constraints for each class. If in contrast no solution exists, the reason for its absence is analyzed. From the result of this analysis, classes and associations responsible for the contradiction are derived.

1) Encoding: To encode the outlined problem, a formula

$$f_{\text{uml}} : \mathbb{N}_0 \times \mathbb{N}_0 \times \cdots \times \mathbb{N}_0 \to \mathbb{B}$$

is created. For this purpose, variables $x_C \in \mathbb{N}_0$ are introduced for each class $C \in \mathcal{C}$ in the considered UML model. Each x_C-variable represents the number of objects derived from class C. Using these variables, all restrictions enforced by the associations are encoded.

Fig. 4(a) shows a generic binary UML association including multiplicities defined over intervals. This UML constraint expresses that each object of class *B* must be linked to at least m_1 (lower bound), but at most m_2 (upper bound) objects of class *A* ($0 \leq m_1 \leq m_2$). The same applies to class *A* analogously. To encode this, the conjunction of the following LIA constraints is created.



(a) Binary association (b) With infinite bound

(c) Ternary association (d) Alternative repr.

Fig. 4. UML constraints

First, constraints ensuring the existence of the minimal number of objects are added. This is expressed by

$$x_A \geq \max\{1, m_1\} \quad \wedge \quad x_B \geq \max\{1, n_1\}. \tag{1}$$

The terms $\max\{1, m_1\}$ and $\max\{1, n_1\}$ imply that each class is instantiated at least once. This is necessary, since empty system states are not considered.

Second, the correlation of x_A and x_B is constrained. For the case of $m_1 = m_2 = 1$ and $n_1 = n_2$, constraining the correlation is straightforward. Then, for each object derived from class A, n_1 objects from class B are needed. To encode this correlation, the LIA constraint $x_B = n_1 x_A$ needs to be added. If additionally $m_1 = m_2 > 1$, this constraint is extended to $m_1 x_B = n_1 x_A$. Having this, the generic LIA constraint additionally considering intervals (i.e. $m_1 < m_2$ and $n_1 < n_2$) can be deduced:

$$m_2 x_B \geq n_1 x_A \quad \wedge \quad m_1 x_B \leq n_2 x_A. \tag{2}$$

Example 2: Applying Eq. (1) and Eq. (2) to the model in Fig. 2 leads to the following encoding:

$$\begin{array}{llll} & (x_A \geq 2) & \wedge & (5x_A = 2x_B) \\ \wedge & (x_B \geq 5) & \wedge & (3x_B = 5x_C) \\ \wedge & (x_C \geq 3) & \wedge & (2x_C = x_A) \\ \wedge & (x_D \geq 5) & \wedge & (5x_C \leq 3x_D) \\ & & \wedge & (12x_C \geq 3x_D) \end{array}$$

Using these formulations most of the UML constraints can be encoded. Beyond that, only the following special cases have to be addressed separately:

- *Infinite upper bounds*
  In fact, infinite upper bounds (i.e. associations with $m_2 = \infty$ or $n_2 = \infty$) weaken the restrictions on the number of objects derived from a class. Accordingly, parts of the LIA constraints from Eq. (2) can be removed. As an example, for the association depicted in Fig. 4(b) with $n_2 = \infty$, the term $m_1 x_B \leq n_2 x_A$ evaluates to $\lim_{n_2 \to \infty} m_1 x_B \leq n_2 x_A = m_1 x_B \leq \infty$. This is always true and, thus, the term can be omitted. Analogously, this can be done for $m_2 = \infty$.
- *Reflexive binary associations*
  Reflexive binary associations represent a special case of binary associations. Their mapping to LIA constraints is already covered by the encoding from Eq. (2). Note that reflexive associations are only valid, if they define an n-*to*-n relation or if they have infinite bounds.
- *n-ary associations*
  Arbitrary n-ary associations (with $n > 2$) can be mapped to LIA constraints in a recursive manner. To illustrate this, consider the ternary association given in Fig. 4(c). According to [13], this can be transformed to equivalent binary associations by adding a *helper class* (denoted by R) and the following OCL constraint (see also Fig. 4(d)):

```
R->forAll(r,r'|
  (r.ra=r'.ra and r.rb=r'.rb and r.rc=r'.rc)
  implies r=r'
)
```

  Further, n-ary associations with $n > 3$ can be transformed accordingly applying this method recursively. From this representation, the respective LIA constraints can be derived. Note that the OCL constraint is not considered in the LIA instance and, therefore, this may lead to false positives. However, contradictions caused by such an association will then be detected in the next debugging step, where OCL constraints are inspected.

2) Analyzing the Result: Using the encoding introduced above an LIA instance results, which can be passed to a respective solve engine (e.g. [14]). Then, this solver tries to determine an assignment to all variables x_C satisfying all constraints or to prove that no such assignment exists.

If the instance is satisfiable, the resulting values from the x_C-variables directly correspond to the number of objects derived from class $C \in \mathcal{C}$. These values can be used for example to help consistency checkers pruning the search space by initially setting a valid number of objects.

In contrast, if there is no solution to the LIA instance, the associations cause the contradiction in the model. Then, a technique called *unsatisfiable core extraction* is applied (see e.g. [15]) in order to determine a set of contradiction candidates among all associations. This technique determines a subset of the instance (a so-called unsatisfiable core), which already is unsatisfiable. From this subset all occurring variables are extracted. Since each variable directly corresponds to a class, the respective classes and therefore the respective associations are obtained. These components, which are usually a very small subset compared to the overall model, explain the contradiction and should be inspected in detail.

Example 2 (continued): Using an LIA-solver, the instance from above is determined to be unsatisfiable. Applying unsatisfiable core extraction leads to the following subset of the instance:

$$(5x_A = 2x_B) \wedge (2x_C = x_A) \wedge (x_C \geq 3) \wedge (3x_B = 5x_C)$$

These are four clauses from the original instance. Extracting all variables from them leads to x_A, x_B, x_C and, hence, to the classes *A*, *B*, and *C* as well as their UML constraints. That is, the designer is pinpointed exactly to the components which are responsible for the contradiction (according to the discussion from Example 1).

B. Debugging OCL Constraints

If the debugging process passes the first step (i.e. if UML constraints are excluded as a reason for the contradiction), the OCL constraints are considered. Again, satisfiability solvers are applied for this purpose.

More precisely, the complete UML model including its OCL constraints is encoded as an instance of the satisfiability problem using *Bit-Vector logic* (BV) [11]. To this purpose, several formulations have been introduced in the past (see e.g. [7], [8], [9]). However, since the model already has been proven to be contradictory, the resulting BV instance is unsatisfiable. Thus, each BV constraint representing an OCL invariant is extended with additional logic allowing to disable it. The number of OCL invariants to be disabled is restricted to k, where the value of k is iteratively incremented (starting from $k = 1$), until a satisfying solution for the instance is determined. From this solution, k OCL invariants can be determined whose deactivation resolves the contradiction. These invariants represent a contradiction candidate.

1) Encoding: In order to encode the outlined idea, we use the formulation introduced in [7], where a Boolean formula is created representing all UML/OCL components and encoding the consistency checking problem. If the given model is consistent, the formula is satisfiable and a valid system state can be obtained from the assignments to all variables. Otherwise, the formula is unsatisfiable proving the non-existence of such a system state. To keep the encoding of the debugging formulation simple, we refer to [7] for a detailed description of the encoding and simplify it as follows:

Given a UML model $\mathcal{M}$ with OCL constraints $\mathcal{I}$, the formula encoding the consistency checking problem is

$$f_{\text{con}} = \Phi_{\text{model}}(\mathcal{M}) \wedge \bigwedge_{i \in \mathcal{I}} \Phi_{\text{inv}}(i), \text{where}$$

- Φ_{model} is a set of BV constraints representing all UML components in a system state such as objects, attribute values, and links, and
- Φ_{inv} is a set of BV constraints representing a given OCL invariant.

As mentioned above, this formula is now extended by additional logic allowing to deactivate invariants. Therefore, a *select variable* s_i is introduced for each invariant $i \in \mathcal{I}$. If s_i is set to 0, then the invariant i is active. Otherwise (if $s_i = 1$), the invariant is disabled. More formally, the resulting debugging formulation is expressed as

$$f_{\text{ocl}} = \Phi_{\text{model}}(\mathcal{M}) \wedge \bigwedge_{i \in \mathcal{I}} (s_i \vee \Phi_{\text{inv}}(i)) .$$

Therefore, if invariant i is causing a contradiction, the solve engine can assign $s_i = 1$ and, thus, still finds a satisfying solution. From the assignment to s_i, the respective invariant can be deduced. Further, the number of disabled invariants is restricted to k:

$$f_{\text{ocl}} = \Phi_{\text{model}}(\mathcal{M}) \wedge \bigwedge_{i \in \mathcal{I}} (s_i \vee \Phi_{\text{inv}}(i)) \wedge \sum_{i \in \mathcal{I}} s_i = k$$

2) Analyzing the Result: Using the encoding described above, it is first attempted to obtain a satisfying assignment for $k = 1$. If no solution exists, k is iteratively increased by one until such an assignment is determined. Then, from all select variables $s_{i_1}, \ldots, s_{i_k}$ set to 1, a set of invariants can be derived, which therefore is the minimal number of contradiction candidates. Disabling all these invariants leads to a valid system state, i.e. these invariants are responsible for the contradiction.

TABLE I
EVALUATED BENCHMARKS

Model	Classes	Associations	Invariants
PyQt4	631	757	736
Android	713	1153	373
Java6	2458	17995	2623

Example 3: Applying this approach to the model in Fig. 2, the solver returns a satisfying assignment for $k = 1$, with $s_{i_1} = 1$, and $s_{i_j} = 0$ ($j = 2, \ldots, 6$). That is, removing the invariant i_1 would resolve the conflict.
However, the conflict could be caused by other, correlated invariants. Thus, further contradiction candidates should be generated. In order to do so, previously found solutions are excluded and the instance is solved again. Therefore, the formula is extended to

$$f_{\text{ocl}} \wedge \overline{(s_{i_1} \wedge \ldots \wedge s_{i_k})}.$$

The correlation between contradiction candidates can provide additional information to aid the designer in determining the error in the model.

Example 3 (continued): Applying the extended formula, three invariants for $k = 1$ are identified as contradiction candidates: i_1, i_2, and i_5. In fact, the conflict in the model is caused by the conjunction of all three of them. (On the other hand, it can be concluded, that the conflict is definitely not caused by the remaining three invariants i_3, i_4, and i_6, i.e. any modification to these constraints cannot fix the conflict.) By pinpointing the designer to these contradiction candidates, the wrong invariant (namely i_5 as discussed in Example 1) can be identified.

VI. EXPERIMENTAL RESULTS

Experimental results are discussed in this section. To evaluate the proposed approaches on large benchmarks, UML models with OCL constraints have been created by re-engineering existing software libraries using introspection tools. Each class in these libraries has been transformed to a UML class, while methods have been used to generate UML associations. Class attributes were built in a similar way. Further, OCL invariants, that restrict the values of the attribute to be valid, have been added. As respective software libraries, we used the Qt4 bindings for Python, the Android SDK, and all classes in the `java.*` namespace of the Java6 SDK. An overview of the resulting UML models is given in Table I listing the number of classes, the number of associations, and the number of invariants. All experiments were carried out on a 64-bit 3 GHz Dual Core AMD processor with 3 GB main memory running Linux 2.6.

In the first evaluation, UML debugging as introduced in Section V-A has been considered. MathSAT4 [14] was applied as the underlying LIA-solver. To generate UML-inconsistent models, associations with arbitrary multiplicities have randomly been injected to the benchmarks introduced above. By this means, ten different instances to be evaluated have been created. Debugging these instances with the proposed approach leads to a number of contradiction candidates as shown in Table II (distinguished between the minimal, maximal, and average number). That is, the numbers of classes and models to be inspected can be reduced from some hundreds (thousands) to no more than 1-2 dozens in just a few seconds (in around half an hour).

TABLE II
RESULTS FOR DEBUGGING OF UML CONSTRAINTS

Model	#contr. cand. min	max	avg	Time (in CPU seconds) min	max	avg
PyQt4	5	14	10.10	8.52	19.44	14.15
Android	3	15	9.90	6.31	19.73	11.65
Java6	5	18	11.00	2077.95	2276.48	2177.09

TABLE III
RESULTS FOR DEBUGGING OF OCL CONSTRAINTS

Model	#contr. cand. min	max	avg	Time (in CPU seconds) min	max	avg
PyQt4	6	6	6	53.88	86.65	59.11
Android	6	6	6	7.52	10.27	8.60
Java6	6	6	6	2116.80	3807.14	3268.58

In the second evaluation, OCL debugging as introduced in Section V-B has been considered. Here, we applied Boolector [11] as the underlying BV-solver. By inverting three randomly selected invariants per model, ten OCL-inconsistent models have been generated. Thus, the contradiction candidates can be found with $k = 3$. Nevertheless, as described in Section V-B, we applied the approach starting from $k = 1$ performing a full debugging run. The obtained results are summarized in Table III. As can be seen, also in this step the majority of invariants can be excluded for further consideration in a very short time. In fact, instead of 736, 373, or 2623, only 6 invariants to be inspected remain for debugging in the respective models. In each case, the injected errors have been unveiled by the contradiction candidates.

VII. RELATED WORK

In the past, debugging techniques based on formal methods have been proposed in the domain of logic circuit design [16] or the design modelling based on declarative specifications such as Alloy [17] and KodKod [18]. However, these approaches address languages and problems different from UML/OCL and, thus, do not consider the two-staged approach proposed in this work.

In the domain of UML/OCL design, only specific debugging problems have been considered so far. For example in [19], [20], it was checked why so called *consistency rules* [21] between different diagrams are invalid. This enables to examine why e.g. a *given* object diagram is inconsistent with respect to a *given* class diagram. In our approach, a broader problem is considered. We aid the designer in determining reasons why a model itself is inconsistent. This does not require any further diagrams except of the model (i.e. the class diagram) itself and helps not only to detect the error in a certain sceneario, but in the overall design.

VIII. CONCLUSIONS

In this work, approaches for debugging inconsistent UML/OCL models have been presented. Different solving as well as debugging techniques are utilized in order to determine contradiction candidates which pinpoint the designer to the source of the error. By means of these contradiction candidates large parts of the model can be excluded from further consideration.

This was also demonstrated by an experimental evaluation. Even for models with more than 2000 classes as well as nearly 18000 association and about 2500 invariants, a small number of components to be inspected is automatically determined in moderate run-time.

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (DFG) (DR 287/23-1).

REFERENCES

[1] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual*. Addison-Wesley Longman, Jan. 1999.
[2] Y. Vanderperren, W. Müller, and W. Dehaene, "UML for electronic systems design: a comprehensive overview," *Design Automation for Embedded Systems*, vol. 12, no. 4, pp. 261–292, Aug. 2008.
[3] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise modeling with UML*. Addison-Wesley Longman, Mar. 1999.
[4] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, Independence and Consequences in UML and OCL Models," in *Tests and Proofs*. Springer, July 2009, pp. 90–104.
[5] A. D. Brucker and B. Wolff, "The HOL-OCL Book," ETH Zurich, Tech. Rep. 525, 2006.
[6] B. Beckert, R. Hähnle, and P. Schmitt, *Verification of Object-Oriented Software: The KeY Approach*. Springer, Oct. 2007.
[7] 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*, Mar. 2010, pp. 1341–1344.
[8] 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*, Oct. 2007, pp. 436–450.
[9] J. Cabot, R. Clarisó, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," in *IEEE Int'l. Conf. on Software Testing Verification and Validation Workshop*, Apr. 2008, pp. 73–80.
[10] B. Dutertre and L. M. de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," in *Int'l Conf. on Computer Aided Verification*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, Aug. 2006, pp. 81–94.
[11] R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays," in *Tools and Algorithms for Construction and Analysis of Systems*, Mar. 2009, pp. 174–177.
[12] M. Soeken, R. Wille, and R. Drechsler, "Verifying Dynamic Aspects of UML Models," in *Design, Automation and Test in Europe*, Mar. 2011, pp. 1077–1082.
[13] 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, 2002, pp. 85–114.
[14] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MathSAT 4 SMT Solver," in *Int'l Conf. on Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Gupta and S. Malik, Eds., vol. 5123, July 2008, pp. 299–303.
[15] R. Bruni, "Approximating minimal unsatisfiable subformulae by means of adaptive core search," *Discrete Applied Mathematics*, vol. 130, no. 2, pp. 85–100, Aug. 2003.
[16] A. Smith, A. G. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, Oct. 2005.
[17] E. Torlak, F. S.-H. Chang, and D. Jackson, "Finding Minimal Unsatisfiable Cores of Declarative Specifications," in *Int'l Symp. on Formal Methods*, ser. Lecture Notes in Computer Science, J. Cuéllar, T. S. E. Maibaum, and K. Sere, Eds., vol. 5014. Springer, May 2008, pp. 326–341.
[18] R. V. D. Straeten, J. P. Puissant, and T. Mens, "Assessing the Kodkod Model Finder for Resolving Model Inconsistencies," in *European Conf. on Modelling Foundations and Applications*, ser. Lecture Notes in Computer Science, R. B. France, J. M. Küster, B. Bordbar, and R. F. Paige, Eds., vol. 6698. Springer, June 2011, pp. 69–84.
[19] A. Nöhrer, A. Reder, and A. Egyed, "Positive effects of utilizing relationships between inconsistencies for more effective inconsistency resolution: NIER track," in *Int'l Conf. on Software Engineering*, R. N. Taylor, H. Gall, and N. Medvidovic, Eds., May 2011, pp. 864–867.
[20] A. Egyed, E. Letier, and A. Finkelstein, "Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models," in *Int'l Conf. on Automated Software Engineering*, Sept. 2008, pp. 99–108.
[21] A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multperspective Specifications," *IEEE Trans. on Software Engineering*, vol. 20, no. 8, pp. 569–578, Aug. 1994.