Semantic Difference Analysis with Invariant Tracing for Class Diagrams Extended by OCL

Bernhard Rumpe
Chair of Software Engineering,
RWTH Aachen University
Aachen, Germany
rumpe@se-rwth.de

Max Stachon
Chair of Software Engineering,
RWTH Aachen University
Aachen, Germany
stachon@se-rwth.de

Sebastian Stüber
Chair of Software Engineering,
RWTH Aachen University
Aachen, Germany
stueber@se-rwth.de

Valdes Voufo
Chair of Software Engineering,
RWTH Aachen University
Aachen, Germany
valdes.voufo@rwth-aachen.de

Abstract

Models are the primary source-artifacts in Model Driven Development (MDD) and are thus subject to changes and evolution throughout the development process. To better understand these model-changes, semantic differencing operators can be employed. In this paper, we present an approach for automatically detecting the semantic differences of Class Diagrams (CDs) that have been extended with Object Constraint Language (OCL) constraints. Previous works regarding OCL models focused mostly on validation and satisfiability of OCL invariants and conditions, not analyzing semantic differences between subsequent versions of CDs and OCL models. While implementations of semantic differencing operators for CDs already exist, they have yet to integrate OCL models in their analysis. Using a translation of CDs and OCL constraints to Satisfiability Modulo Theories (SMT), we developed a tool for detecting semantic differences between two compositions of CD and OCL models. The differences are reported in the form of Object Diagrams (ODs) that describe valid instances of one model but not the other. Additionally, invariants are traced across models. The implementation of this tool is publicly available.

CCS Concepts

• **Software and its engineering → Formal software verification**; **Domain specific languages**; *Object oriented development*; **Software development methods**.

Keywords

OCL, CD, Analysis, Semantics, Differences, Tracing, UML, Model-Driven

ACM Reference Format:
Bernhard Rumpe, Max Stachon, Sebastian Stüber, and Valdes Voufo. 2024. Semantic Difference Analysis with Invariant Tracing for Class Diagrams Extended by OCL. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3652620.3687818

1 Introduction

Effective change management is crucial for successful software development. In MDD the primary development artifacts are models and their evolution becomes a major concern of the development process. During the requirements elicitation phase, models are created as abstract representations of the system's functionalities and constraints [21]. In the following development phases, the design of the system progresses, and models are refined and expanded to include more detailed information about the system's components and interactions. During the implementation, code generators are used to automatically generate code from source models [43, 45]. Over the course of this development process, semantic differencing operators can be employed to better understand the impact of changes made to models, thereby assisting developers in change management [30, 40].

We focus in this paper on CDs and OCL of the Unified Modeling Language (UML) [20], more specifically the UML/P variant described in [42]. CDs model the structure of object-oriented software systems via classes and relations between them. However, their expressiveness is limited, which is why OCL is used to specify additional properties and behaviors using invariants, *i.e.,* constraints that should always hold during the system's lifetime, as well as operation-specific constraints in the form of pre- and post-conditions.

We ask the following research questions:

(RQ1) Can semantic differences between CDs extended by OCL be automatically detected.
(RQ2) Can corresponding tooling be used in the context of MDD to determine or exclude refinement relations between model versions.

Our main contribution is a semantic differencing operator for compositions of CDs and OCL models that utilizes a translation to

SMT [4] and the solver Z3 [11]. The semantic difference is reported via a diff-witness in the form of an OD. These witnesses represent valid instances of the new model version that are not permitted by the old version (or the other way around if their positions in the input are switched). The operator is also capable of tracing invariants, *i.e.,* detecting which invariants of the new model version imply invariants of the old version. Furthermore, we have extended the operator to detect the semantic difference of operation constraints and produce diff-witnesses in the form of pairs of ODs, representing the data-states before and after execution.

The remainder of this paper is structured as follows: In the next section, we discuss related work. Then in section 3, we outline preliminary concepts and technologies. Section 4 introduces a running example that we will refer to in the rest of the paper. Section 5 and section 6 describe translation of CD and OCL constraints to SMT. In section 7, we detail our approach for semantic differencing. The tool is then evaluated regarding its ability to determine refinement between model versions in section 8. A conclusion and outlook on future work is given in section 9.

2 Related Work

In the following, we discuss related work. We focus on analysis of CDs and OCL models, as well as semantic differencing.

According to [27], a precise semantics is needed in order to perform automatic verification of CDs and OCL constraints and [39] provides such a formalization for OCL expressions in the form of functions that map a variable assignment and a class to an object identifier of a corresponding data instance. We chose a more depictable notion of semantics in the form of object structures representing valid data states.

The same authors also provide an approach for verifying CDs annotated with OCL in [38], where snapshots of a system's data-state are checked for consistency regarding the data model and corresponding constraints. Another verification-approach for OCL models that reduces the problem to SAT-solving is introduced in [44]. Similarly, [28] presents a translation of UML CDs annotated with OCL constraints into relational logic, as well as a translation back from relational instances to object structures. This translation is then used in conjunction with the SAT-based constraint solver Kodkod [47] for analysis. We believe that this translation should also allow for semantic difference analysis. However, for our implementation of the differencing operator, we decided to use SMT due to its better support of the String data type.

Our translation of CDs to SMT was in part inspired by [36]. The authors present an approach for verification of data models in the context of web-applications using the Model-View-Controller (MVC) pattern. They are able to automatically extract formal data models from Object Relational Mapping (ORMs), then verify them regarding corresponding verification queries by using a SMT solver such as Z3 [11].

In [7] and [8], the Cabot et al. present translations of CDs annotated with OCL models to Constraints Satisfiability Problems (CSPs). This allows verifying the compliance of the model with respect to several correctness properties using constraint programming. Similarly, Pérez and Porres [37] presents a framework for reasoning about CDs annotated by OCL constraints based on constraint logic programming that utilizes the formal specification language FORMULA [22] and the SMT solver Z3[11]. However, a limitation of all these approach is the explicit boundedness of the search space. We instead opted for a direct but modularized translation of CDs and OCL constraints to SMT, which allows us to mix and match translation strategies and avoid explicitly defined bounds on the search space if needed.

In a recent publication [48], Hao Wu presents QMaxUSE, a tool for incrementally verifying the satisfiability of a CD with OCL constraints that allows for concurrent verification of user queries. According to Wu, the concurrent incremental verification improves performance when compared to similar approaches. The translation of CDs and OCL invariants is based on a previous translation described in [49] in order to detect inconsistencies in domain / metamodels and determine a maximum set of consistent features based on a ranking of model elements. The tool MaxUSE is build on top of the modeling tool USE [18] which offers a variety of verification options for CDs with OCL constraints [19]. At the time of publication the translation for MaxUse did not include some of the features we currently support, such as constraints on strings, closure operators and variable declarations using the context keyword.

Previous literature describes a variety of existing semantic differencing operators that focus primarily on a single modeling language each:

CDDiff is a semantic differencing operator introduced in [33] that is able detect the semantic differences of two CDs. If a semantic difference is detected, CDDiff outputs diff-witnesses in the form of ODs. These witnesses describe valid instances of the first CD that are not permitted by the second CD. The operator utilizes a translation to Alloy [23, 25] in order to find these instances using the Alloy Analyzer. Originally, CDDiff could only operate under a closed-world assumption on CD semantics, *i.e.,* object structures were not allowed to contain instances of types, attributes, or associations that were not explicitly modelled in the diagram. This limits the applicability of semantic differencing in analyzing the model evolution in early development phases, as the addition of new model elements would not be considered a refinement [35]. To address this issue, CDDiff was later extended to be able to operate under an open-world assumption [40], *i.e.,* consider CDs as underspecified and thus permit additional objects, links, and attribute instances within object structures.

Drave et al. [14] presents a semantic differencing operator for Feature Models (FMs) that can compute witnesses in the form of feature configurations. The diff operation for FMs can be performed under the closed-world assumption as well as the open-world assumption, *i.e.,* either prohibiting or permitting features not modelled in the FM in valid feature configurations.

Maoz et al. [31] introduces ADDiff, a semantic differencing operator for Activity Diagrams (ADs) that uses a translation to SMV [32] to determine diff-witnesses in the form of execution traces. [26] presents an alternative approach to ADDiff that considers a smaller subset of ADs and reduces the problem of semantic differencing to language inclusion checking. The same approach is used for semantic differencing of State Charts (SCs) in [13] and [24] as well as Time-Synchronous Port-Automata (TSPA) in [6].

3 Preliminaries

In this section, we outline existing concepts and technologies that are relevant for this paper.

3.1 Class Diagrams and their Semantics

CDs are widely used to model the structure of object-oriented software systems. They define the set of possible object structures that comprise a potential data state of a system [41]. A CD consists of a finite set of type declarations, which are either classes, interfaces or enumerations, as well as a finite set of associations between types. A class may contain attributes and method signatures, while an interface may only contain the later. Furthermore, a class may extend other classes and implement interfaces. Interfaces, however, may only extend other interfaces. Classes can also be declared abstract such that they cannot be instantiated directly. An enumeration defines a set of constants. An association references exactly two classes and may have a role-name as well a cardinality for each side. A cardinality imposes constraints on the number of allowed instances, *i.e.,* links, referencing the same object.

We consider the semantics of a CD to be the set of its valid instances, *i.e.,* the object structures that it permits. The asymmetric semantic difference of two CDs is then the set of object structures permitted by the first CD and not the second. Accordingly, we refer to such object structures as diff-witnesses. If the semantics of a CD is included in the semantics of another CD, we consider the former to be a refinement of the latter. If both are refinements of one another, we consider them to be refactorings. For a more detailed discussion on the semantics of CDs refer to [9, 15, 29, 35].

Object structures consists of objects and links between objects. An object may also contain attributes with specific types and values. We model object structures as ODs.

3.2 Object Constraint Language

OCL is a textual modeling language that is used in conjunction with other modeling languages for adding logical constraints to the semantics of an existing model. It can therefore be used to add additional restrictions on the object structures defined by a CD, *e.g.,* the range of an integer attribute can be restricted to a value between 0 and 100. An *invariant* is a constraint that should always hold. An operation constraint is usually given in the form of a *precondition* and a *postcondition.* The postcondition must hold after the execution of an operation if the precondition was fulfilled beforehand. For a more in-depth discussion on the UML/P variant of OCL refer to [42, 43].

3.3 Satisfiability Modulo Theorem

SMT-solvers determine whether a given mathematical formula is satisfiable. While the satisfiability problem is, in general, NP-hard [10], in practice, SMT-solvers are often able to find a solution quickly. The syntax of SMT inputs is standardized in SMTLib2 [4]. We opted to use `Z3`[11], one of the most prominent SMT-solvers that is utilized in many applications [5, 17, 34, 46]. To solve an SMT problem, Z3 tries to instantiate variables and functions so that the Boolean formulas are satisfied. The solver returns `SAT` and a model (values of variable and functions) if the formulas are satisfiable. It returns `UNSAT` and an `UNSAT-CORE` if the formulas are not satisfiable. The UNSAT-CORE is a subset of assertions, which are in conjunction not satisfiable. For example, take the following three assertions from listing 1 over an integer-attribute `balance`:

A1) `balance>10`;
A2) `even(balance)`;
A3) `balance<4`.

There is no possible solution where all three assertions are satisfied. Hence, the SMT-solver returns UNSAT and an UNSAT-CORE. In this example the UNSAT-CORE consists of assertions A1) and A3), since they directly contradict each other.

```
(set-option :produce-unsat-cores true)
(declare-const balance Int)
(assert (! (> balance 10)       :named A1))
(assert (! (=(mod balance 2) 0) :named A2))
(assert (! (< balance 4)        :named A3))
(check-sat)
;unsat
(get-unsat-core)
;(A1 A3)
```

Listing 1: SMT example with UNSAT-CORE

4 Motivating Example

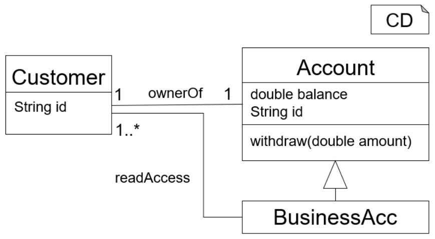

Figure 1: Class Diagram of a Bank Management System

We consider a bank management system modelled with a CD that uses OCL to ensure the integrity and consistency of the data. The development team uses MDD methods to automatically generate code and configuration files from the models. In this example, the CD is used to generate a SQL-data schema and Java classes. Validator-methods are generated from the OCL model, which check whether the data conforms to the constraints. This ensures that the data in the database is always consistent. The development of the bank management system is ongoing, and requirements might be added, modified or removed.

The CD is displayed in fig. 1. It consists of the classes `Customer`, `Account`, and `BusinessAcc` (business account). A `BusinessAcc` is a special kind of `Account` which has at least one `Customer` with read access. A `Customer` must own exactly one `Account` and an `Account` is owned by exactly one `Customer`.

The OCL constraints in listing 2 contains two invariants that model the following requirements: 1) the `id` of a `Customer` is unique (cf. lines 1-3 in listing 2) 2) the `balance` of a `Account` must always be positive (cf. lines 4-6 in listing 2). Additionally, the precondition of `withdraw(double amount)` function ensures that the withdraw-amount is less or equal to the current balance (cf. lines 8-9 in listing 2).

```
inv CustomerID_unique:
  forall Customer c1, c2:
    c1 != c2 implies c1.id != c2.id;
inv Balance_positive:
  forall Account a:
    a.balance >= 0;

context Account.withdraw(double amount)
pre: balance >= amount;
// post condition not (yet) specified
```

Listing 2: Old OCL specification

These models are used to generate a prototype. The generated code is extended with handwritten functionality, *e.g.,* a concrete implementation of `withdraw` in Java. After demonstrating the prototype to the customer, it becomes clear that the constraints must be changed. For one, the `id` of an `Account` must be unique (cf. lines 1-3 in listing 3). Next, the `id` of a `Customer` has to match the `id` of their `Account` (cf. lines 4-6 in listing 3). Also, `Accounts` may have a negative balance. Finally, the `withdraw(double amount)` postcondition is specified (cf. lines 10-12 in listing 3).

```
1  inv AccountID_unique:
2    forall Account a1, a2:
3      a1 != a2 implies a1.id != a2.id;
4  inv AccountID_Eq_CustomerID:
5    forall Account a:
6      a.id == a.customer.id;
7
8  // Balance_positive missing
9
10 context Account.withdraw(double amount)
11 pre:  balance >= amount;
12 post: balance == balance@pre - amount;
```

Listing 3: New OCL specification

The developer can now re-generate the system with the new constraints. But questions arises on how these OCL changes impact the previously developed infrastructure: Can previously written data still be loaded from the database, or are there new inconsistencies? Are there new situations, which the handwritten code never considered? Which of the new constraints relate to the old constraints? Was there an old constraint that was forgotten?

The tool presented in this paper helps developers answer such questions by analysing the two model versions and demonstrating the impacts of the changes. Supported analysis are:

Refinement. If the new invariants imply the previous invariants, we refer to these new invariants as *refinements* of the previous ones. This means that all input-assumptions in the previously handwritten code are still fulfilled. If, on the other hand, the previous invariants are a refinement of the new invariants, all data can be loaded from the database as it fulfills the new invariants.

In the example from listings 2 and 3, the previous model and the new models are not in a refinement relation, as the value of an `Account`’s `balance` is no longer required to be positive in the new model. But when looking at the individual constraint, we can identify refinements between them. The constraints `AccountID_unique` and `AccountID_Eq_CustomerID` in conjunction constitute a refinement of the previous constraint `CustomerID_unique`. Therefore, handwritten code that assumes the uniqueness of the customer-id is still correct.

Tracing. With *tracing* we are able to identify logical implication of constraints across different versions of OCL models. Tracing also shows which constraints are no longer part of the new version.

As described above, the conjunction of `AccountID_unique` and `AccountID_Eq_CustomerID` constitutes a refinement of the previous constraint `CustomerID_unique`. Consequently, they are in a trace relationship and the developer is informed that the new constraints imply the previous constraints. The previous constraint `Balance_positive` is not implied by the new constraints, hence it is not in a trace-relationship.

Diff-Witness. When a constraint is not refined by another constraint there exists a *diff-witness*, *i.e.,* an object structure which satisfies the latter, but not the former. This witness is presented to the developer in the form of an OD so that he can better understand the semantic difference between the two constraints. In addition, the OD can be used to generate additional test cases [43].

For the example above, one diff-witness is an OD containing an `Account`-object with negative `balance`. Hence, the `withdraw` operation can be executed with a negative `balance` according to the new specification of the system. Since this situation never occurred before, it is also untested and the method might behave unexpectedly. The modeler can use the diff-witness to automatically create a corresponding test-case.

5 From Class Diagram to SMT

There are several possible approaches for translating CDs to SMT. As such, we make use of the strategy pattern [16] and decompose the transformation of the CD into smaller strategies. We implemented strategies to transform classes and interfaces, strategies for associations, and strategies for inheritance relations. In the first step, the *class-strategy* translate the classes, interfaces, and their attributes. In the second step, the *association-strategy* uses the declaration of the class-strategy to translate associations. Finally, in the third step, the *inheritance-strategy* transforms inheritance relations between classes and interfaces. The strategy pattern allows to quickly define a new strategy and use it in the transformation. In the following, we show one example algorithm for each strategy type.

Example of a Class-Strategy. For each class `A`, we introduce a new SMT sort symbol S_a. For each attribute `attr` of class `A` with the type `T`, we introduced a function symbol f_{attr} with range S_t and domain S_a, where S_t is the SMT representation of the type `T`. Types like `int, double, char, String` and `boolean` are predefined in SMT. Listing 4 shows how the `Customer` class from fig. 1 is translated into SMT.

```
(declare-sort S_Customer)
(declare-fun Customer_id (S_Customer) (String))
```

Listing 4: Translation of `Customer` class to SMT

There is one notable difference between a class in a CD and a sort in SMT: SMT sorts are never empty. Hence this example strategy always instantiates every class at least once.

Example of an Association-Strategy. For each association `assoc` between classes or interfaces `A` and `B`, we introduced a new function

f_{assoc} in SMT. The function takes two values as parameter. One value has the type `S_B` and the other the type `S_A`. A boolean is returned. As an interpretation, the function returns true iff both objects are linked by the association. The cardinality of the association (e.g. `[1...*]`) is translated as a additional assertion. Note that the direction of associations is ignored, since OCL can always navigate in both directions.
For the association `readAccess` between `Customer` and `BusinessAcc` in fig. 1, the SMT translation is shown in listing 5.

```
(declare-fun f_readAccess
                  (S_Customer S_BusinessAcc) (Bool))
; cardinality constraint [1..*]:
(assert (forall ((ba S_BusinessAcc))
   (exists ((c S_Customer)) (f_readAccess c ba))))
```

Listing 5: Translation of `readAccess` relation into SMT

Example of an Inheritance-Strategy. Since SMT has no concept of inheritance, the translation is not as straightforward as the previous ones. Instead, we create a virtual object for each level of the inheritance hierarchy and relate each sub-object with its corresponding super-object. In practise, this means declaring functions for the navigation between sub-object and super-object, introducing enumeration data-types to denote the sub-type of a super-object, and defining constraints to guarantee a one-to-one correspondence between them. Unlike a simple flattening of the inheritance hierarchy, this allows us to consider constraints concerning super-types without duplicating them for each sub-type. E.g., for the inheritance relation between `BusinessAcc` and `Account` (cf. fig. 1), we have the following transformation:
1) A function symbol `super_BAAccount`, that casts objects of the type `S_BusinessAcc` to `S_Account` (cf. line 1-3 in listing 6). This function is used to access the properties of the super-class.
2) A new datatype `Account_sub` that represents an enumeration of the sub-classes of `Account` (cf. line 4-6 in listing 6).
3) A function `Account_type` from `S_Account` to `Account_type`, which returns the concrete type of an account (cf. line 7-9 in listing 6).
4) Assertions over the functions `super_BAAccount` and `Account_type` to ensure that parent objects are unique and the type is correct (cf. line 10-13 in listing 6).

```
1  ; Cast to super class
2  (declare-fun super_BusinessAcc
3                         (S_BusinessAcc) (S_Account))
4  ; Possible subtypes of Account
5  (declare-datatypes ((Account_sub 0))
6                     (((T_BusinessAcc) (T_Account))))
7  ; Concrete type of Account
8  (declare-fun Account_type
9                         (S_Account) (Account_sub))
10 ; After casting to superclass, type is correct
11 (assert (forall ((ba S_BusinessAcc))
12          (= (Account_type (super_BusinessAcc ba))
13                T_BusinessAcc)))
13 ; [...] more asserts omitted
```

Listing 6: Translation of inheritance between `BusinessAcc` and `Account` into SMT

6 From OCL to SMT

Conceptually, there is a large overlap between expressions in OCL and SMT. OCL `let`-expressions can be directly translated into SMT `let`-expressions, and Z3 even has a built-in transitive-hull operator. Consequently, we will only detail the translation of two particular concepts: set comprehension and operation constraints. Nonetheless, it should be noted that, as in our translation from CD to SMT detailed in 5, for the sake of modularity, we again make use of the strategy pattern [16].

6.1 Set Comprehension

Types like `Boolean`, `Real`, `Int`, and `String` are already supported by SMT, but translating the commonly used set-comprehension expressions from OCL to SMT is a bit more difficult. Consider, *e.g.*, the expression in listing 7 which restricts the `id` of a `Customer` to the values `{"id_a", "id_b", "id_c"}`.

```
inv Customer_id_value:
  forall Customer c: c.id isin
             {"id_" + s | s in Set{"a","b","c"}};
```

Listing 7: Set Comprehension in OCL

We can define an equivalent SMT-assertion without a set-datatype. The expression in listing 8 uses the fact that the set is only an intermediate value. The expression is of type `Boolean` which is well-supported by SMT.

```
(assert (forall ((c S_Customer))
  (exists ((s String))
                ; c.id isin {"id_" + s | [...]}
       (and (= (Customer_id c) (str.++ "id_" s))
                ; s in Set{"a","b","c"}
             (or (= s "a") (= s "b") (= s "c"))))))
```

Listing 8: Translation of Set Comprehension into SMT

Set-operations such as union, intersection, minus or equality can be translated into SMT in a similar manner. However, the limitations of this approach become apparent when we try to translate the size operator for sets. Our translation can handle simple uses of size like `(1 <= |S1|)` which is equivalent to `(exists s: s isin S1)`. But for more complicated size-expression an error is thrown. For example, our translation into SMT fails on the following OCL-expression: `(|S1| <= |S2|)`. Luckily, some other SMT solver like CVC4 [3, 12] support a finite-set theory [2] and the associated operations. We leave full support of the size-operator on sets as future work.

6.2 Operation Constraint

Operation Constraints specify the allowed pairs of system states for before and after an operation is executed via pre- and postconditions. Objects, attributes, and links in the precondition always refer to the time before execution. By default, values in the postcondition refer to the time after execution, but OCL does offer a `@pre`-operator which can be used to refer to pre-execution values in the postcondition.

Lines 10-12 in listing 3 define an operation constraint on the method `withdraw` in the class `Account`. The precondition requires the balance of the account to be greater or equal to the amount to withdraw and the postcondition ensures that the balance of the account is correct after the transaction. The value of balance in the precondition is different from the value of balance in the postcondition. In order to consider this change, we modify the structure of the class diagram before translating it to SMT. Each class now has an additional _pre version of each attribute and each association. Pre-attributes and links hold the values of the object before executing the operation and the remaining attributes and association have post-execution values. Figure 2 displays the transformation of the class Account. The occurrence of the attribute balance in the precondition refers to balance_pre.

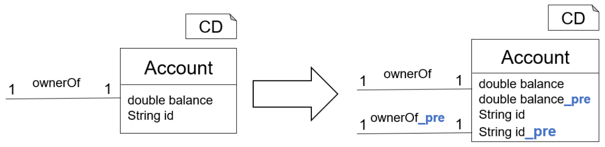

Figure 2: Transformation of CD for Operation Constraints

When an attribute does not occur in the operation constraint, the modeler assumes that the value was not changed. For example, since the account-id does not occur in lines 10-12 of listing 3, it is assumed that its value was not changed by the withdraw operation. But this is not encoded in the OCL-formula and it is possible to fulfill the postcondition with a modified account-id. It would be possible to add such a constraint as a further assertion to Z3, but doing so would change the semantics of the operation constraint. Instead, this could be translated as a soft-constraint (assert-soft) to Z3. This way Z3 would prefer solutions where the values do not change, but can also find solutions where the values change.
To correctly handle all cases where objects are created or deleted by an operation, additional information needs to be encoded into the CD. Both points are left as future work.

7 Semantic Difference of Two Models

To analyse the differences between two versions of CD and OCL models, we first use CDDiff, an existing semantic differencing operator for CDs [33], as a preprocessing step to identify structural differences in the input CDs that cause a semantic difference. Association cardinalities are extracted and removed from the CDs before calling the operator in order to account for OCL constraints that further restrict the number of links between certain objects. When the two data-models are not in a refinement-relation, CDDiff returns an OD as a diff-witness and the diff-operation terminates.

Otherwise, the newer version of the CD as well as the extracted cardinalities of both CDs are translated to SMT. From there, we can compute the semantic difference and derive several model properties.

The benefit of reusing CDDiff is that we do not have to translate and negate the structural constraints defined by a second CD on the set of permitted object-structures to SMT, a task that is not as simple as it might first appear and would require a significant amount of additional effort. However, there is a trade-off. The extraction of cardinalities does not completely eliminate the possibility of false positives, *i.e.,* semantic differences between the two CDs that disappear when considering the OCL constraints. These edge cases only occur when OCL constraints prevent the instantiation of model-elements, rendering them redundant in the CD. We therefore assume that they are rare in practice.

7.1 Refinement Checking

The first property that can be checked is the refinement property: We verify whether all object structures that satisfy the new OCL specifications also satisfy the previous specifications. Let Φ_i be a new constraint with $i \in I$ and Ψ_k an old constraint with $k \in K$. The new OCL constraints are a refinement of the old OCL constraints, precisely if the old constraints follow from the new constraints. This is written as:

$$\bigwedge_{i\in I} \Phi_i \implies \bigwedge_{k\in K} \Psi_k$$

This is equivalent to

$$\forall k \in K : (\bigwedge_{i\in I} \Phi_i) \implies \Psi_k$$

Which in turn is equivalent to:

$$\forall k \in K : (\bigwedge_{i\in I} \Phi_i \land \neg\Psi_k) \texttt{ is unsatisfiable}$$

Thus, a refinement check can be divided into multiple satisfiability checks. Running multiple smaller checks instead of one large check enables more finegrained control. For example, runs can be parallelized across different machines. Moreover, smaller problems are easier to solve and a timeout is less likely. Finally, smaller checks give us more information on the semantic relation of invariants, as every check corresponds to exactly one old constraint. Consequently, the output can detail which old constraints have not been refined by the new constraints.

7.2 Requirement Tracing

When the solver produces UNSAT on a check, a refinement was detected. There is however the possibility that the new constraints are contradicting, and thus no valid object structures exist. This always constitutes a refinement. We check for contradiction in a separate operation and assume in the following that the new constraints are satisfiable. Hence, the negated old constraint must be part of the UNSAT-CORE. Furthermore, the conjunction of all other assertions in the UNSAT-CORE implies the old constraint. Consequently, we can form a trace relationship between the other assertions in UNSAT-CORE and the old constraint.

The trace is returned as an instance of the CD in fig. 3. Each artefact has a file-path and the line-number inside the file. Concrete elements are OCLInv, which corresponds to an OCL invariant, and Assoc, which corresponds to the cardinality-restrictions of an assocation.

For the example in section 4 the trace from new constraints to old constraints is presented in fig. 4. Filepaths and line-numbers are omitted from the figure. The trace shows that the old invariant CustomerID_unique is implied by the three constraints on the right side. The other old invariant Balance_positive has no incoming trace-link, hence it does not hold in the new version. With the trace

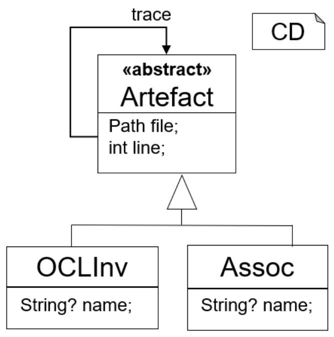

Figure 3: CD used for Traces

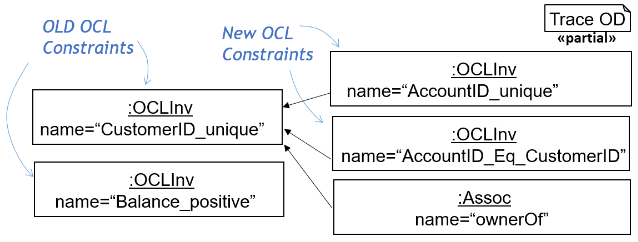
OLD OCL Constraints
New OCL Constraints
Trace OD «partial»
:OCLInv name="AccountID_unique"
:OCLInv name="CustomerID_unique"
:OCLInv name="AccountID_Eq_CustomerID"
:OCLInv name="Balance_positive"
:Assoc name="ownerOf"

Figure 4: Trace between New and Old Version

OD the developer can quickly see how constraints from different versions are related.

7.3 Witness Object Diagram

When the SMT-solver returns SAT on a check, a semantic difference was detected. The SMT-solver then produces an SMT-model with an assignment that fulfills all constraints. Afterwards, a diff-witness is constructed by translating this SMT-model to an OD. This translation happens automatically. Figure 5 shows one possible diff-witness for the example from section 4. Here, the ids of `Customer` and `Account` are different, thus violating the constraint `AccountID_Eq_CustomerID`.

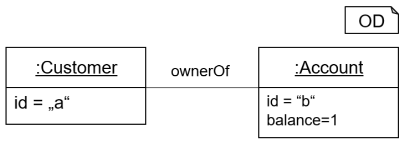

Figure 5: Witness OD

For operation constraints a single witness-OD is not sufficient. Consider for example a modified postcondition. Here the diff-witness must consist of a pair of ODs. One OD describing the data-state before the execution of the operation, which fulfills both the old and the new precondition, and another OD for the data-state after the execution, which fulfills only one of the postcondition. This pair of ODs is derived from a (counter-)example produced by Z3 which contains instances of classes with both pre- and post-versions of attributes and associations.

7.4 Tool Implementation

The implementation of the verification tool described in this paper is referred to as `OCLDiff`. It is Java-based and uses the corresponding Java-API of Z3. `OCLDiff` takes as input `.cd`- and `.ocl`-files containing models in a textual syntax based on UML/P. More specifically, we use the same syntax as defined by `CD4Analysis`[1] and `OCL/P`[2]. The tool is publicly available at:

https://github.com/MontiCore/ocl

The jar can be executed with the following CLI command:

```
java -jar MCOCL.jar --diff                              \
           -cd  Version1.cd  -ocl v1a.ocl v1b.ocl     \
           -ncd Version2.cd -nocl v2.ocl
```

This command computes the semantic difference between the two versions and the resulting diff-witness ODs and trace-OD are pretty-printed to the console.
The tool also provides additional analyses:

- *Consistency Check.* Given a CD and a set of OCL constraints it computes a witness Object Diagram that is a legal instance of the CD and satisfies the OCL Constraint.
- *Operation Witness.* Given a CD and a method name it provides two system states before and after applying the operation of an object.

8 Evaluation

```
classdiagram User {
 class Session {
   String name;
 }
 class User {
   String name;
 }
 class Role {
   String name;
 }
 class Permission {
   String name;
 }

 class UserRole extends Role;
 class AdministrativeRole extends Role;
 class UserPermission extends Permission;
 class AdministrativePermission extends Permission;

 // association [1..*] Session -> (roles) Role [1..*];
 association [1] User -> (sessions) Session [1..*];
 association [*] User (users) -> (roles) Role [*];
 association [*] Role -> Permission [*];
}
```

Listing 9: CD for role-based access control in `CD4Analysis` syntax

In order to evaluate our tool's ability to determine refinement between two OCL model versions in a MDD context, we constructed a case study based on the role-based access control system modeled by Ahn and Shin [1]. We start with the given CD and corresponding OCL constraints and consider several changes that could occur during further development. Our semantic differencing operator is then employed to determine whether these changes constitute successful refinement steps. Note that we focus exclusively on

[1]https://github.com/MontiCore/cd4analysis
[2]https://github.com/MontiCore/OCL

changes to OCL conditions and not the CD as the capabilities of CDDiff have already been evaluated in previous works [32, 40].

We first translate the CD into the textual syntax of CD4Analysis. Because of performance issues, we were forced to exclude one of the associations. Our current assumption is that association-cycles in our encoding might cause the SMT-solver some difficulty. By removing this association performance increases drastically. Luckily, we can adjust corresponding OCL constraints by using chained field-access calls. The CD used for the evaluation is displayed in Listing 9 with the association in question commented out.

Similarly, we translate the initial OCL constraints containing 8 invariants into the syntax of OCL/P. Two of these constraints (Example 7 and Example 8) use the size operator, which we currently do not support. As such, we refactor them into equivalent constraints that do not use this operator (cf. Listing 10).

```
  // Example 7: There can be only one chairman.
context Role r inv ExactlyOneChairman:
    r.name == "chairman" implies
    exists User u in r.users:
      (forall User u1 in r.users: u == u1);

  // Example 8: The session limit is two.
context User u inv SessionLimitation:
    !(exists Session s1, Session s2,Session s3:
        u.sessions.containsAll(Set{s1, s2, s3})
        && s1 != s2 && s3 != s2 && s1 != s3);
```

Listing 10: Refactored OCL constraints: size operator removed.

Step 1. In the first step, additional requirements are added to the project: permission and role names must be unique. These are modeled via the two OCL invariants displayed in Listing 11, which are then added to the project for the second version of the OCL model.

```
context Role r1, Role r2 inv RoleNameUnique:
     (r1.name == r2.name) <=> (r1 == r2);

context Permission p1, Permission p2 inv PermNameUnique:
     (p1.name == p2.name)  <=> (p1 == p2);
```

Listing 11: Added specifications to make permissions and role names unique.

When executing OCLDiff to compare this version to its predecessor, no diff-witnesses are found and instead, we are informed that our changes constitute a refinement. This is expected, as we have only added further specifications without making any changes to the initial constraints, thus simply restricting the set of permitted object-structures.

```
 //some roles have exactly one user
 context Role r inv ExactlyOneChairman:
  !(r.name == "chairman") ||
  exists User u in r.users:
    (forall User u1 in r.users: u == u1);
```

Listing 12: Refactored OCL constraints: size operator removed.

Step 2. In the next step, our goal is to perform a refactoring of OCL constraints, in order to utilize our code generation Syntactic changes are necessary, as the code generator in question does not support all features of OCL. Additionally, invariants must have a specific syntax if we want to obtain code structured in the manner we desire. For example, implications are transformed into equivalent formulas using disjunction and negation, as shown for the invariant *ExactlyOneChairman* in Listing 12.

When comparing this version of constraints to the previous one, OCLDiff finds a diff-witness, indicating that the semantics was not preserved and our attempt at refactoring was faulty. Upon reviewing the OCL specification, we can note the following errors:

(1) A typo was made in a role name in the first invariant.
(2) An invariant was forgotten.
(3) Negation was omitted somewhere when removing implications.

The other invariants were successfully refined / refactored, and a corresponding tracing was also produced.

Step 3. In a third step, the bugs of the previous versions are fixed, and a fourth and final version is produced. With the help OCLDiff, we are then able to verify that this version refines both the first and second version of constraints.

We conclude that OCLDiff is indeed capable of detecting semantic differences and determining refinement between OCL models versions of moderate size in a MDD process in a reasonable time. All experiments were performed on an 11th Gen Intel Core i7-1185G7 CPU, 3.0 GHz, with 32 GB RAM, running Windows 10. Note that the non-deterministic nature of Z3 might make it difficult to reproduce our measurements.

9 Conclusion and Future Works

In this paper, we presented a fully automatic tool that can detect the semantic difference between two compositions of CD and OCL models. Compared to previous works, where the focus was the verification of a single model, our method allows developers to check for refinement across model versions in an MDD context. When a newer model does not refine the previous version, the tool produces an OD as diff-witness. The tool utilizes an existing semantic differencing operator for CDs as well as a translation to SMT and the solver Z3 to compute these witnesses. Additionally, it provides a tracing of the specifications from the previous version of the model to the newer version.

In the future, we will define additional strategies for classes, associations, and inheritance relations and compare them. Further testing of the tool is underway, as well. The translation of CDs to SMT and SMT to ODs is tested by a separate tool that automatically verifies whether an OD represents a valid instance of a CD. This tool was also previously used to evaluate CDDiff [40]. For evaluating the translation of OCL to SMT, valid instances in the form of ODs can be translated to SMT and checked against the translated OCL constraints. We also plan to extend the tool such that ODs can be used as input to describe valid or invalid data patterns, in general. Moreover, we intend to integrate our tool into a larger toolchain where the diff-ODs can be used as test-input, and the trace-OD is used for artefact analysis. Lastly, we want to further evaluate the tool on examples from industry. One research question would concern the prevalence of false positives produced by using CDDiff as a subroutine of our approach. We currently consider these to be rare edge cases, however, if they occur frequently for industry

Table 1: Supported OCL features

Feature	**Supported and example**	**Transformation in SMT**
Datatypes	Supported are `int`, `Double`, `String`, and `Date`. The remaining types are not yet supported.	`int`, `Double`, and `String` are supported by SMT, and `Date` is translated to `int` using the Unix time format.
String Operations:	`replace()`, `contains()`, `startsWith()`, `endsWith()`,`concat()` are supported.	supported by SMT
Quantifiers:	`forall`, `exists`	supported by SMT
Conditional Expression:	`if then else`	supported by SMT
Common Expressions:	`+, -, *, /, >, <, <=, %, and, or, xor, etc`	`supported by SMT`
Context declaration:	`context Auction a : true;`	`(forall ((Auction a)) :(true))`

Field Access Expressions:	`context Auction a  inv :`
	`    a.name == "Auct1";`
		`(forall ((a Auction))`
		`        (= (attr_name a) "Auct1"))`

Set Comprehension:	`context Auction a inv:`
	`  a.name isin Set{"Auct1", "Auct2"};`
		`(forall ((a Auction))`
		`  (or (= "Auct2" (attr_name a))`
		`      (= "Auct1" (attr_name a))))`

Set Operations:	`union`, `intersection`, `difference` are supported and `size()` is not supported. e.g:
	`context Auction a inv:`
	`  a.name isin Set{"Auct1", "Auct2"}`
	`   intersect Set{"Auct3"};`
		`(forall ((a Auction))`
		`  (and (or (= "Auct1" (attr_name a))`
		`           (= "Auct2" (attr_name a)))`
		`      (= "Auct3" (attr_name a))))`

Table 2: Supported class diagram features

Feature	**Supported**	**Not supported**
Types	class, abstract class, interface, enum	
Inheritance	multiple Inheritance of interfaces, simple inheritance of classes	multiple inheritance of classes
Attribute datatypes	enumeration types, `String, boolean, int, double, Date`	the remaining types
Association	association/composition with roles and cardinalities	
Association cardinalities	optional(`[0..1]`), one(`[1]`) , atLeastOne(`[1..0]`), multiple(`[*]`)	the remaining cardinalities

models, we might need to address it by making corresponding adjustments to our approach.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 250902306

References

[1] Gail-Joon Ahn and Michael E Shin. 2001. Role-based authorization constraints specification using object constraint language. In *Proceedings Tenth IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises. WET ICE 2001*. IEEE, 157–162.

[2] Kshitij Bansal, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. 2016. A new decision procedure for finite sets and cardinality constraints in SMT. In *Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27–July 2, 2016, Proceedings*. Springer, 82–98.

[3] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification*. Springer, 171–177.

[4] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, Vol. 13. 14.

[5] Nikolaj Bjørner and Karthick Jayaraman. 2015. Checking cloud contracts in Microsoft Azure. In *Distributed Computing and Internet Technology: 11th International Conference, ICDCIT 2015, Bhubaneswar, India, February 5-8, 2015. Proceedings 11*. Springer, 21–32.

[6] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)* (Gothenburg). IEEE, 145–154. http://www.se-rwth.de/publications/Semantic-Differencing-for-Message-Driven-Component-and-Connector-Architectures.pdf

[7] Jordi Cabot, Robert Claris´o, and Daniel Riera. 2008. Verification of UML/OCL Class Diagrams using Constraint Programming. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*. 73–80. https://doi.org/10.1109/ICSTW.2008.54

[8] J. Cabot, R. Clarisó, and D. Riera. 2014. On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software* 93 (2014), 1–23. https://doi.org/10.1016/j.jss.2014.03.023

[9] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. 2008. *System Model Semantics of Class Diagrams*. Informatik-Bericht 2008-05. TU Braunschweig, Germany. http://www.se-rwth.de/staff/rumpe/publications20042008/System-Model-Semantics-of-Class-Diagrams.pdf

[10] Stephen A Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*. 151–158.

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[12] Morgan Deters, Andrew Reynolds, Tim King, Clark Barrett, and Cesare Tinelli. 2014. A tour of CVC4: how it works, and how to use it. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 7–7.

[13] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. 2019. Semantic Differencing of Statecharts for Object-oriented Systems. In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD’19)* (Prague), Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić (Eds.). SciTePress, 274–282. http://www.se-rwth.de/publications/Semantic-Differencing-of-Statecharts-for-Object-oriented-Systems.pdf

[14] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. 2019. Semantic Evolution Analysis of Feature Models. In *International Systems and Software Product Line Conference (SPLC’19)* (Paris), Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi (Eds.). ACM, 245–255. http://www.se-rwth.de/publications/Semantic-Evolution-Analysis-of-Feature-Models.pdf

[15] Uli Fahrenberg, Mathieu Acher, Axel Legay, and Andrzej Wąsowski. 2014. Sound Merging and Differencing for Class Diagrams. In *Fundamental Approaches to Software Engineering*, Stefania Gnesi and Arend Rensink (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 63–78.

[16] Erich Gamma, Ralph Johnson, Richard Helm, Ralph E Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

[17] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing: SAGE has had a remarkable impact at Microsoft. *Queue* 10, 1 (2012), 20–27.

[18] Martin Gogolla, Fabian Büttner, and Mark Richters. 2007. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming* 69, 1 (2007), 27–34. https://doi.org/10.1016/j.scico.2007.01.013 Special issue on Experimental Software and Toolkits.

[19] Martin Gogolla and Frank Hilken. 2016. Model validation and verification options in a contemporary UML and OCL analysis tool. In *Modellierung 2016*. Gesellschaft für Informatik e.V., Bonn, 205–220.

[20] Object Management Group. 2017. OMG Unified Modeling Language (OMG UML). (2017).

[21] ANN M. Hickey and Alan M. Davis. 2004. A Unified Model of Requirements Elicitation. *Journal of Management Information Systems* 20, 4 (2004), 65–84. https://doi.org/10.1080/07421222.2004.11045786 arXiv:https://doi.org/10.1080/07421222.2004.11045786

[22] Ethan Jackson and Wolfram Schulte. 2013. FORMULA 2.0: A Language for Formal Specifications. In *Unifying Theories of Programming and Formal Engineering Methods*. Springer Berlin Heidelberg, 156–206. https://www.microsoft.com/en-us/research/publication/formula-2-0-language-formal-specifications/

[23] Jackson, Daniel. 2006. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

[24] Oliver Kautz. 2021. *Model Analyses Based on Semantic Differencing and Automatic Model Repair*. Shaker Verlag. http://www.se-rwth.de/phdtheses/Diss-Kautz-Model-Analyses-Based-on-Semantic-Differencing-and-Automatic-Model-Repair.pdf

[25] Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2017. *CD2Alloy: A Translation of Class Diagrams to Alloy*. Technical Report AIB-2017-06. RWTH Aachen University. http://www.se-rwth.de/publications/CD2Alloy-A-Translation-of-Class-Diagrams-to-Alloy.pdf

[26] Oliver Kautz and Bernhard Rumpe. 2018. Semantic Differencing of Activity Diagrams by a Translation into Finite Automata. In *Proceedings of MODELS 2018. Workshop ME* (Copenhagen). http://www.se-rwth.de/publications/Semantic-Differencing-of-Activity-Diagrams-by-a-Translation-into-Finite-Automata.pdf

[27] Stuart Kent, Andy Evans, and Bernhard Rumpe. 1999. UML Semantics FAQ. In *Object-Oriented Technology, ECOOP’99 Workshop Reader (LNCS 1743)*, A. Moreira and S. Demeyer (Eds.). Springer Verlag, Berlin.

[28] Mirco Kuhlmann and Martin Gogolla. 2012. From UML and OCL to Relational Logic and Back. In *Model Driven Engineering Languages and Systems*, Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 415–431.

[29] Achim Lindt, Bernhard Rumpe, Max Stachon, and Sebastian Stüber. 2023. CDMerge: Semantically Sound Merging of Class Diagrams for Software Component Integration. *Journal of Object Technology* 22, 2 (July 2023), 2:1–14. https://doi.org/10.5381/jot.2023.22.2.a1

[30] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2010. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME’10) (LNCS 6627)*. Springer, 194–203. http://www.se-rwth.de/publications/A-Manifesto-for-Semantic-Model-Differencing.pdf

[31] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE ’11)*. ACM, 179–189. http://www.se-rwth.de/publications/ADDiff-Semantic-Differencing-for-Activity-Diagrams.pdf

[32] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. *An Operational Semantics for Activity Diagrams using SMV*. Technical Report AIB-2011-07. RWTH Aachen University, Aachen, Germany. http://www.se-rwth.de/publications/An-Operational-Semantics-for-Activity-Diagrams-using-SMV.pdf

[33] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CDDiff: Semantic Differencing for Class Diagrams. In *ECOOP 2011 - Object-Oriented Programming*, Mira Mezini (Ed.). Springer Berlin Heidelberg, 230–254. https://se-rwth.de/publications/CDDiff-Semantic-Differencing-for-Class-Diagrams.pdf

[34] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. 2015. Hardware verification using software analyzers. In *2015 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 7–12.

[35] Imke Nachmann, Bernhard Rumpe, Max Stachon, and Sebastian Stüber. 2022. Open-World Loose Semantics of Class Diagrams as Basis for Semantic Differences. In *Modellierung 2022*. Gesellschaft für Informatik, 111–127. https://doi.org/10.18420/modellierung2022-016

[36] Jaideep Nijjar and Tevfik Bultan. 2012. Unbounded data model verification using SMT solvers. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 210–219. https://doi.org/10.1145/2351676.2351706

[37] Beatriz Pérez and Ivan Porres. 2019. Reasoning about UML/OCL class diagrams using constraint logic programming and formula. *Information Systems* 81 (2019), 152–177.

[38] Mark Richters and Martin Gogolla. 1998. On Formalizing the UML Object Constraint Language OCL. In *Conceptual Modeling – ER ’98*, Tok-Wang Ling, Sudha Ram, and Mong Li Lee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 449–464.

[39] Mark Richters and Martin Gogolla. 2000. Validating UML Models and OCL Constraints. In *«UML »2000 — The Unified Modeling Language*, Andy Evans, Stuart Kent, and Bran Selic (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–277.

[40] Jan Oliver Ringert, Bernhard Rumpe, and Max Stachon. 2023. On Implementing Open World Semantic Differencing for Class Diagrams. *Journal of Object Technology* 22, 2 (July 2023), 2:1–14. https://doi.org/10.5381/jot.2023.22.2.a11

[41] Bernhard Rumpe. 2011. *Modellierung mit UML, 2te Auflage*. Springer Berlin. https://mbse.se-rwth.de/

[42] Bernhard Rumpe. 2016. *Modeling with UML: Language, Concepts, Methods*. Springer International. https://mbse.se-rwth.de/

[43] Bernhard Rumpe. 2017. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International. https://mbse.se-rwth.de/

[44] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. 2010. Verifying UML/OCL models using Boolean satisfiability. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. 1341–1344. https://doi.org/10.1109/DATE.2010.5457017

[45] EV Sunitha and Philip Samuel. 2018. Object constraint language for code generation from activity models. *Information and Software Technology* 103 (2018), 92–111.

[46] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–white box test generation for. net. In *Tests and Proofs: Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings 2*. Springer, 134–153.

[47] Emina Torlak and Daniel Jackson. 2007. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems: 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24-April 1, 2007. Proceedings 13*. Springer, 632–647.

[48] Hao Wu. 2023. QMaxUSE: A new tool for verifying UML class diagrams and OCL invariants. *Science of Computer Programming* 228 (2023), 102955.

[49] Hao Wu and Marie Farrell. 2021. A formal approach to finding inconsistencies in a metamodel. *Software and Systems Modeling* 20, 4 (2021), 1271–1298.