Innovations in Systems and Software Engineering (2026) 22:14
https://doi.org/10.1007/s11334-026-00630-y

[RSSV26] B. Rumpe, M. Stachon, S. Stüber, V. Voufo:
Semantic difference analysis for model compositions of class diagrams, OCL constraints, and object diagrams.
In: Innovations in Systems and Software Engineering, Volume 22(1), pp. 14, DOI 10.1007/s11334-026-00630-y,
Springer, May 2026.

ORIGINAL ARTICLE

Semantic difference analysis for model compositions of class diagrams, OCL constraints, and object diagrams

Bernhard Rumpe[1] · Max Stachon[1] · Sebastian Stüber[1] · Valdes Voufo[1]

Received: 15 January 2025 / Accepted: 12 April 2026

Abstract

Models are the primary source-artifacts in model-driven development 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 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, and the refinement of invariants is traced across models. In addition to invariants, we also consider semantic differencing of operation constraints in OCL, as well as the supplementary translation of ODs to SMT for further restricting or completing instances. The implementation of this tool is publicly available.

Keywords OCL · Class Diagrams · Analysis · Model Semantics · Differencing · Tracing · UML · Model-Driven

1 Introduction

Effective change management is crucial for successful software development. In Model Driven Development (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 [1]. 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 [2, 3]. 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 [4, 5].

We focus in this paper on CDs and OCL of the Unified Modeling Language (UML) [6], more specifically the UML/P variant described in [7]. 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.

Our primary research question that we wish to address in this paper is:

(RQ1) Can semantic differences between CDs extended by OCL be automatically detected.

Bernhard Rumpe, Max Stachon, Sebastian Stüber, and Valdes Voufo contributed equally to this work.

✉ Max Stachon
stachon@se-rwth.de

Bernhard Rumpe
rumpe@se-rwth.de

Sebastian Stüber
stueber@se-rwth.de

Valdes Voufo
valdes.voufo@rwth-aachen.de

[1] Chair of Software Engineering, RWTH Aachen University, Aachen 52074, Germany

Published online: 25 May 2026

There also exists a follow-up questions that we will not be able to fully address in this paper, but that is nonetheless of interest to us, and will be considered for future work on this subject:

(RQ2) Can corresponding tooling be used in the context of MDD to determine or exclude refinement relations between model versions.

Our main contribution for this paper is a semantic differencing operator for compositions of CDs and OCL models that utilizes a translation to SMT [8] and the solver `Z3` [9]. The semantic difference is reported via a diff-witness in the form of an object diagram (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.

This paper extends the previous publication [10] by introducing several new contributions to the field. We present and discuss alternative translation approaches from CDs to SMT and consider their application in our case-study. Additionally, we introduce a novel method for translating object diagrams into SMT, which is useful for completing object diagrams in conjunction with OCL constraints, thereby facilitating the rapid creation of test inputs, as well as defining and forbidding anti-patterns. We also elaborate our discussion on semantic differencing of operation constraints and include a further evaluation of our tool's scalability using constructed models of different sizes. In general, we offer an extended description, discussion, and evaluation of our approach.

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 the translation of CD and OCL constraints to SMT. In section 8, we detail our approach for semantic differencing. The tool is then evaluated regarding its ability to determine refinement between model versions in section 9. A conclusion and outlook on future work is given in section 10.

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 [11], a precise semantics is needed in order to perform automatic verification of CDs and OCL constraints and [12] 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 [13], 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 [14]. Similarly, [15] 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` [16] 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 [17]. 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` [9].

In [18] and [19], 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, [20] presents a framework for reasoning about CDs annotated by OCL constraints based on constraint logic programming that utilizes the formal specification language FORMULA [21] and the SMT solver `Z3`[9]. 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 [22], 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 [23] in order to detect inconsistencies in domain / meta-models and determine a maximum set of consistent features based on a ranking of model elements. The tool `MaxUSE` is built on top of the modeling tool `USE` [24] which offers a variety of verification

options for CDs with OCL constraints [25]. 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 [26] 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 [27, 28] 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 [29]. To address this issue, `CDDiff` was later extended to be able to operate under an open-world assumption [5], i.e., consider CDs as underspecified and thus permit additional objects, links, and attribute instances within object structures.

[30] 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.

[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. [33] 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 [34] and [35] as well as Time-Synchronous Port-Automata (TSPA) in [36].

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 [37]. 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 latter 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 [29, 38–40].

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 [2, 7].

3.3 Satisfiability modulo theorem

SMT-solvers determine whether a given mathematical formula is satisfiable. While the satisfiability problem is, in general, NP-hard [41], in practice, SMT-solvers are often able to find a solution quickly. The syntax of SMT inputs is

standardized in SMTLib2 [8]. We opted to use Z3[9], one of the most prominent SMT-solvers that is utilized in many applications [42–45]. 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.

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

Listing 1 SMT example with UNSAT-CORE

4 Motivating example

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.

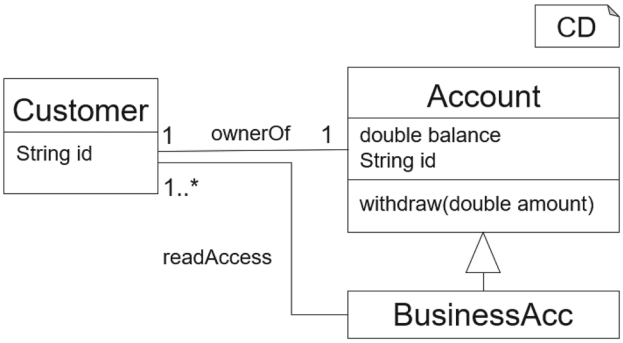

Fig. 1 Class Diagram of a Bank Management System

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).

```
1  inv CustomerID_unique:
2    forall Customer c1, c2:
3      c1 != c2 implies c1.id != c2.id;
4  inv Balance_positive:
5    forall Account a:
6      a.balance >= 0;
7
8  context Account.withdraw(double amount)
9  pre: balance >= amount;
10 // 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).

```
inv AccountID_unique:
  forall Account a1, a2:
    a1 != a2 implies a1.id != a2.id;
inv AccountID_Eq_CustomerID:
  forall Account a:
    a.id == a.customer.id;

// Balance_positive missing

context Account.withdraw(double amount)
pre:  balance >= amount;
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 [2].

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 [46] and decompose the transformation of the CD into smaller strategies. An overview of the supported CD features is provided in table 2.

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 discuss some of the implemented strategies.

5.1 Class strategies

We have implemented three distinct *class strategies* for encoding classes and interfaces, as well as their attributes to SMT: *distinct sort*, *single sort*, and *bounded datatype*.
Class Strategy: Distinct Sort
The fist *class strategy* introduces a new SMT sort symbol S_a for each class `A`. Additionally, 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, and

thus require no dedicated encoding efforts. Listing 4 shows how the class Customer from fig. 1 is translated into SMT.

```
1 (declare-sort S_Customer)
2 (declare-fun Customer_id (S_Customer)
3                          (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, i.e., with this encoding every class must be instantiated at least once.

This can be an issue in some cases, so we developed an alternative *class strategy* that defines a single sort for all objects of all types.

Class strategy: single sort

Our second *class strategy* defines a common sort Object for all instances of all CD types, as seen in listing 5, line 3. To distinguish between different types of objects, we declare the datatype CDType which encodes all classes and interfaces declared in the CD as its constructors. Next, we declare the function has_type with the purpose of mapping each instance of Object to its corresponding CDType. Moreover, we require a function for each attribute declaration in the CD that maps an instance of Object to the attribute's value for that instance, i.e., either another instance of Object or an instance of a predefined type. The problem here is of course redundancy, i.e., each object is assigned a value for every attribute of every class, even if the corresponding types of the object do not contain this attribute. This issue also translates to the encoding of the associations discussed in section 5.2.

The encoding of the class Customer from fig. 1 now looks as depicted in listing 5: The common sort Object and the datatype CDType are declared in line 2 and line 5, respectively. The constructors for CDType corresponding to the 3 classes Customer, Account, and BusinessAccount declared in the CD. In line 8, the function has_type is declared, which should return true precisely if the specified Object is of the specified CDType. The attribute id of the class Customer is encoded using the function Customer_attr_id declared in line 11, which intends to map each specified Object to a String corresponding to the attribute's value for a Customer instance. Additional constraints on the declared functions are required in order to complete the encoding, such as the constraint defined in lines 15-21 which ensures that each instance of Object has precisely 1 type.

```
1  ; common sort for all objects of all types
2  (declare-sort Object 0)
3
4  ; data-type encoding all classes and
       interfaces from the CD
5  (declare-datatypes ((CDType 0))
6    (((Customer) (Account) (BuisinessAcc))))
7
8  ; function for mapping objects to their
       respective types
9  (declare-fun has_type (Object CDType) Bool)
10
11 ; function for Customer.id that maps each
       Object to a String value
12 (declare-fun Customer_attr_id (Object)
       String)
13
14 ; constraint that ensures each Object has
       precisely 1 type
15 (assert (forall ((obj Object) (t1 CDType) (
       t2 CDType))
16   (and
17         (=> (and (has_type obj t1)
18                  (has_type obj t2))
19             (= t1 t2))
20         (exists ((t CDType))
21             (has_type obj t)))
22   ))
```

Listing 5 Translation of Customer class to SMT

Class strategy: bounded datatype

Another encoding that we experimented with defined an SMT datatype for each class and interface declared in the CD with each constructor representing a specific instance of that type. This results in a finite preset quantity of objects for each type, which can be checked iteratively to achieve a bounded model checking approach. An example of a corresponding SMT encoding can be seen in listing 6. We generate such an encoding to consider instances of the CD from fig. 1 with precisely 2 accounts, 1 business account, and 2 customers.

```
1  ; Declaration of bounded datatypes:
2
3  ; 2 Account instances
4  (declare-datatypes ((Account 0))
5      (((Acc_0)(Acc_1))))
6
7  ; 1 BusinessAcc instance
8  (declare-datatypes ((BusinessAcc 0))
9      (((BAcc_0))))
10
11 ; 2 Customer instances
12 (declare-datatypes ((Customer_obj 0))
13     (((Cust_0)(Cust_1))))
```

Listing 6 Transformation of Classes in to Bounded Datatypes in SMT

One of the major benefits of this encoding strategy is that the use of quantifiers is significantly reduced, as constraints can now be specified for each individual object. This drastically improves the performance of the SMT solver. In turn, however, the encoding now has to be computed multiple times in an iterative manner in order to get the desired result. With regards to the semantic difference analysis, this approach limits the completeness of the result, i.e., if no diff

witness is discovered, the only guarantee is that none exist within in the specified bounds. Additionally, scalability also becomes an issue: if no semantic difference exists all configurations have to be checked until the bound is reached.

5.2 Association strategies

For associations, we have implemented a *default strategy*, as well as a slightly modified version that concerns the translation of *one-to-one associations*.

Default 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 7.

```
(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 7 Translation of `readAccess` relation into SMT

One-to-one association strategy

In order to optimize the SMT solving operation, we developed an additional encoding for one-to-one associations. This new encoding translates each association to a function that maps an object of one associated type to an object of the other associated type. The direction of the function is based on the lexicographic order of the associated types. For the association `ownerOf` between `Customer` and `Account` in fig. 1, the SMT translation is shown in listing 8.

```
(declare-fun customeraccount
                 (Account_obj) Customer_obj)
```

Listing 8 Translation of `owenerOf` relation into SMT

5.3 Inheritance strategies

The *inheritance strategies*, which encode the inheritance hierarchy of the CD in SMT, are dependent on the encoding of classes and interfaces performed by the *class strategies*. To accommodate the implemented *class strategies*, we utilize two `inheritance strategies`: one utilizing the type system of SMT sorts in combination with *virtual super-/sub objects*, and another which circumvents the existing SMT type system entirely with a `custom type system` using SMT functions.

Inheritance strategy: virtual super-/sub-objects

First, we consider an encoding developed to work in tandem with the *distinct sort class strategy* but is also adaptable for the *single sort* and *bounded datatype* approaches.

Since SMT has no concept of inheritance for sorts, the translation is not as straightforward as in the previous strategies. 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 9). 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 9).

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 9).

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 9).

```
; Cast to super class
(declare-fun super_BusinessAcc
                  (S_BusinessAcc) (S_Account)
                     )
; Possible subtypes of Account
(declare-datatypes ((Account_sub 0))
              (((T_BusinessAcc) (T_Account)))
                  )
; Concrete type of Account
(declare-fun Account_type
                    (S_Account) (Account_sub)
                       )
; After casting to superclass, type is
    correct
(assert (forall ((ba S_BusinessAcc))
     (= (Account_type (super_BusinessAcc ba)
         )
         T_BusinessAcc)))
; [...] more asserts omitted
```

Listing 9 Translation of inheritance between `BusinessAcc` and `Account` into SMT

Inheritance strategy: custom type system

When using the *single sort class strategy*, we effectively circumvent the type system of SMTLib using the datatype CDType. Therefore, to encode the type hierarchy, we can relate the instances of CDType via a subtype function to reflect the inheritance hierarchy in the CD (cf. line 1 in listing 10), as well as an instance_of function that determines if a specified intance of Object is of the specified type (cf. line 2 in listing 10). For each sub-type-relation in the CD we add a constraint to the subtype function (cf. line 13-23 in listing 10). Additionally, we add a constraint based on the function subtype ensuring that instances of Object related to a type are also related to all of its super-types (cf. lines 5-10 in listing 10).

```
1  (declare-fun subtype (CDType CDType) Bool)
2  (declare-fun instance_of (Object CDType)
       Bool)
3
4  ;Assertion defining the instance_of function
5  (assert
6     (forall ((obj Object) (type CDType))
7        (= (instance_of obj type)
8           (or (has_type obj type)
9              (exists ((type1 CDType))
10               (and (subtype type1 type)
11                    (has_type obj type1))))))
                      )
12
13 ;Assertion defining the subtype function
14 (assert
15    (and true
16      (subtype Customer Customer)
17      (not (subtype Customer Account))
18      (not (subtype Customer BusinessAcc))
19      (not (subtype Account Customer))
20      (subtype Account Account)
21      (not (subtype Account BusinessAcc))
22      (not (subtype BusinessAcc Customer))
23      (subtype BusinessAcc Account)
24      (subtype BusinessAcc BusinessAcc)))
```

Listing 10 Encoding the inheritance hierarchy of the CD in SMT when the single sort class strategy is used

Finally, we need one more constraint that prevents any instances of Object that are solely related to abstract or interface types by the instance_of function. Our CD from the motivating does not declare such types, but suppose there was an interface AccountInterface, we would then generate the corresponding assertion seen in listing 11.

```
1 (assert (forall ((obj Object))
2     (not (has_type obj AccountInterface))))
```

Listing 11 Assertion to prevent direct instances of AccountInterface

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 and extensibility, we again make use of the strategy pattern [46]. An overview of the supported OCL features is provided in table 3.

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 12 which restricts the id of a Customer to the values {"id_a", "id_b", "id_c"}.

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

Listing 12 Set Comprehension in OCL

We can define an equivalent SMT-assertion without a set-datatype. The expression in listing 13 uses the fact that the set is only an intermediate value. The expression is of type Boolean which is well-supported by SMT.

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

Listing 13 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 [47, 48] support a finite-set theory [49] 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

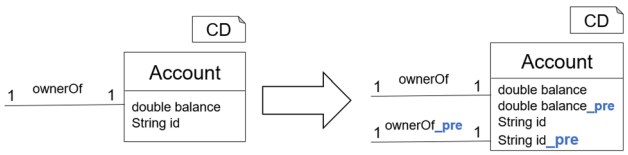

Fig. 2 Transformation of CD for Operation Constraints with `@pre` and `@post` attributes

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 than or equal to the amount to withdraw, while the postcondition ensures that the balance of the account is correct after the transaction. Specifically, when `withdraw` is called, if the precondition does not hold (i.e., if the balance is less than the amount requested), then the behavior of the operation is not specified. In this scenario, it is uncertain whether the postcondition will hold or not; thus, any changes to the state of the account remain ambiguous. It is crucial to note that regardless of whether preconditions are satisfied, normal invariants from listing 3 must always hold for both pre- and post-states. This ensures that essential properties of objects are maintained throughout their lifecycle, providing a consistent framework for reasoning about operations within our model.

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`.

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.

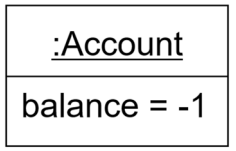

Fig. 3 Partial OD as input for completion. It only contains the relevant information for the test

To correctly handle all cases where objects are created or deleted by an operation, additional information needs to be encoded into the CD. The transformation implicitly assumes that the object exists at both the beginning of the method, in the precondition, and in the post-condition. Alternatives such as additional boolean attributes `existsPre` and `existsPost` or a loop association like listing 14 have been be considered.

```
1 association Account [0..1] (post)
2       <-> (pre) [0..1] Account;
```

Listing 14 Alternative pre/post translation to fig. 2

However, both alternatives further complicate the SMT problem and reduce the effectiveness of Z3. Therefore, we focus on the main case and view creation and deletion as corner cases for future work.

7 From object diagram to SMT

The translation of UML Object Diagrams (ODs) into SMT queries presents a direct and efficient approach for leveraging the structural information encapsulated in ODs. Unlike an intermediary translation into OCL, which would introduce additional existential quantifiers and potentially degrade the performance of SMT solvers, we perform a direct conversion. This method not only preserves the integrity of the original diagram but also enhances computational efficiency.

Use-case 1: completing partial ODs

One significant application of this translation is the completion of partial ODs. In scenarios where modelers focus on specific objects and their attributes for testing purposes, the SMT solver can generate valid values for all other attributes, ensuring that the resulting OD adheres to all defined constraints. This capability allows for effective testing while maintaining a valid object structure.

Figure 3 shows a partial OD of the running example. This OD should be used to test the behaviour of `whitdraw` with a negative balance. Using the "Distinct Sort" strategy from section 5 the OD from fig. 3 is translated into the following SMT-Code:

```
1 (declare-const x1 (S_Account))
2 (assert (= (Account_balance x1) -1))
```

Listing 15 Translation of partial OD from fig. 3 into SMT

Use-case 2: forbidding antipatterns

Another compelling use case involves the identification and prohibition of antipatterns within ODs. By negating specific

conditions represented in an OD, we can enforce constraints such as "there cannot be an account associated with two customers." This modeling technique effectively captures design flaws that may compromise system integrity. While not a conventional application of ODs, this approach aligns with UML's `complete` stereotypes and suggests that more expressive ODs–potentially incorporating variables at attributes instead of fixed values– could further enhance our ability to describe various antipatterns comprehensively. This is discussed in [50].

8 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 [26], 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 datamodels 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 by translating both sets of OCL constraints to SMT, and checking the satisfiability of each negated old constraint against the set of new constraints.

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.

Note that we will not evaluate `CDDiff` in this paper, as it was already evaluated in previous literature [5, 26]. We will also not present any examples of diff-witnesses produced by `CDDiff`, as these can be found in the literature, as well. Instead, for the remainder of the paper, we will only consider diff-witnesses produced by our SMT-based operator for semantic differencing of OCL constraints and the model properties that can be checked using it.

8.1 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:

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

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.

8.2 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. 4. 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.

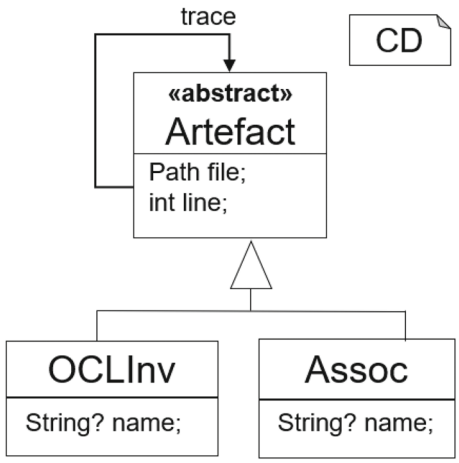

Fig. 4 CD used for Traces

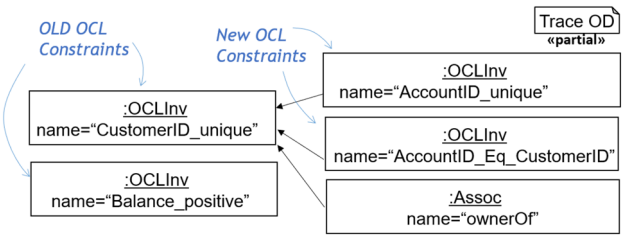

Fig. 5 Trace between New and Old Version

For the example in section 4 the trace from new constraints to old constraints is presented in fig. 5. 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 OD the developer can quickly see how constraints from different versions are related.

8.3 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 6 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`.

Of course diff-witnesses produces by our method are not always minimal and their size may vary due to `Z3`'s non-deterministic behavior. Depending on the strategy a diff-witness may contain at least one object per class even if unnecessary. However, each witness corresponds to precisely

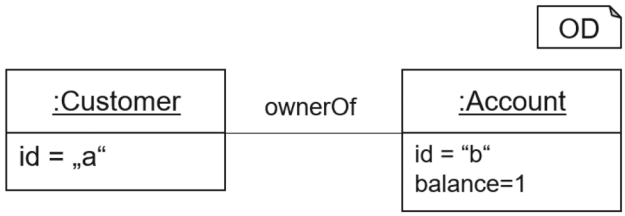

Fig. 6 Witness OD

one of the old constraints that was not refined by the new constraints, which makes finding the actual diff considerably easier, thus drastically improving understandability.

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.

8.4 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`, `.ocl`, and `.od`-files containing models in a textual syntax based on UML/P. More specifically, we use the same syntax as defined by `CD4Analysis`[1], `OCL/P`[2], and `OD`[3]. The tool is publicly available at:

https://github.com/MontiCore/ocl

The jar can be executed with the following CLI command:

[basicstyle=]

```
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 features:

[1] https://github.com/MontiCore/cd4analysis
[2] https://github.com/MontiCore/OCL
[3] https://github.com/MontiCore/object-diagram

- *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.
- *Object Diagram Completion:* Complete a partial object diagram, e.g. useful for test-case generation.

8.5 Example diff witness

If we compute the semantic difference between the new OCL specification of our *Band Management System* from section 4 to the previous specification using our tool, we will get an OD containing at least one `Account` or `BusinessAcc` object with negative `balance`. An example diff-witness `Balance_positive` can be seen in listing 16. Note that the OD's name corresponds to the missing invariant from the original set of OCL specifications.

```
objectdiagram Balance_positive {
  customer_2:Customer {
    String id="";
  };
  businessAcc_0:BusinessAcc {
    double balance=-1.0;
    String id="C";
  };
  account_0:Account {
    double balance=-1.0;
    String id="A";
  };
  customer_0:Customer {
    String id="A";
  };
  customer_1:Customer {
    String id="C";
  };
  account_1:Account {
    double balance=-1.0;
    String id="";
  };
  link customer_0(customer)--(account)
      account_0;
  link customer_1(customer)--(businessAcc)
      businessAcc_0;
  link customer_1(customer)--(account)
      businessAcc_0;
  link customer_2(customer)--(account)
      account_1;
}
```

Listing 16 Witness OD in textual notation

9 Evaluation

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 [51]. 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.

```
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 17 CD for role-based access control in `CD4Analysis` syntax

Note that we focus exclusively on changes to OCL conditions and not the CD as the capabilities of `CDDiff` have already been evaluated in previous works [5, 32].

9.1 Case study

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 17 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 18).

```
// 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 18 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 19, which are then added to the project for the second version of the OCL model.

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.

```
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 19 Added specifications to make permissions and role names unique.

```
//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 20 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 20.

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 of `OCLDiff`, we are then able to verify that this version refines both the first and second version of constraints.

9.2 Configurations and parameters

All experiments were performed on an 11th Gen Intel Core i7-1185G7 CPU, 3.0 GHz, with 32 GB RAM, running Windows 10. We tested 3 configurations of encoding strategies in order to perform the evaluation, 2 of these configurations proved successful while the remaining one did not terminate in reasonable time.

Note that the non-deterministic nature of `Z3` might make it difficult to reproduce our measurements.

First Configuration (4min 12 sec)
The first successful configuration employed the *single sort class strategy*, the *custom type system inheritance strategy*, as well as the *default* and *one-to-one association strategies*. With this configuration, we were able to perform the case study in 4 minutes and 12 seconds. This time includes the entire process of parsing, conversion to SMT, and the SMT-solving operation. Note, however, that the parsing and conversion is negligible compared to the actual solver operation.

Second Configuration (aborted)
For our second configuration, we changed the *inheritance strategy* from *custom type system* to *virtual super-/sub-objects*. Otherwise this configuration is the same as the first, i.e., we used the *single sort class strategy*, as well as the *default* and *one-to-one association strategies*. We found that this approach did not terminate in reasonable time, and thus we had to abort the execution.

Third Configuration (1min 1sec)
The third configuration utilizes a two-step process: First, we use the *distinct sort class strategy*, the *virtual super-/sub-objects inheritance strategy*, as well as the *default* and *one-to-one association strategies* with a timeout after 10 seconds. Second, whenever the timeout occurs, the analysis is again performed using the *bounded datatype class strategy* in an iterative manner with the maximum number of objects set to 1000. With this configuration, we were able to perform the case study in 1 minute and 1 second.

An interesting observation that let to the construction of the third configuration: We noticed that, in most cases, `Z3` was able to quickly determine the unsatisfiability of the encoded problem using the *distinct sort class strategy*. On the other hand, the *bounded datatype strategy* proved particularly useful in cases where the problem was satisfiable, i.e., a semantic difference was present. Combining the two approaches with a timeout of 10 seconds allowed us to achieve the best performance result out of all 3 configurations.

However, one issue that remained was that in cases where the unsatisfiability could not be determined by the first approach in time, the second approach would have to go through all configurations of objects until the upper bound was reached. Not only is this time-consuming, it would also not guarantee model refinement due to the bounded search space.

Through some testing we determined a timeout after 10 seconds to be appropriate, as from our experience, the analysis using the *distinct sort class strategy* would not terminate in reasonable time if it took longer than that.

Size and Understandability of Diff Witnesses
We observed that the size of diff-witnesses produced during this case study ranged from 6 objects and 2 links to 26 objects and 11 links. Each corresponded precisely to one of the constraints, we failed to properly refactor in Step 2, making manual review and understanding the diff straightforward.

9.3 Scalability

In order to evaluate the scalability of our approach, we considered multiple constructed models of varying sizes. We performed 10 differencing operations, each involving one CD and two sets of OCL constraints that were compared regarding their semantic differences. The third configuration of translation strategies was chosen as it performed the best in our case study.

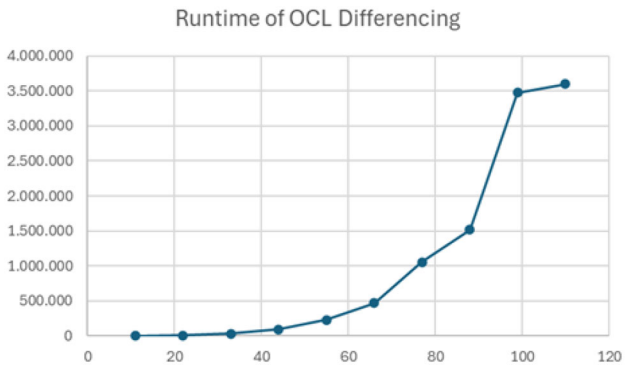

Fig. 7 Runtimes of the OCL differencing operations in ms (x-axis) compared to the size of constructed CDs (y-axis)

The CDs and sets of OCL constraints were constructed by an algorithm. Each CD consisted of a chain of *stars*, with each star having a central class connected to 10 other classes via associations. The central class of each star was also connected to the central class of the next star via association. We started with a single star and added one more with each subsequent difference operation.

Each class contained two attributes `int num` and `String text`. For each class, we constructed an OCL constraint that required `num` to be greater than 5 and `text` to not be the empty string. Finally, a second set of OCL constraint was added that contained precisely one difference to the previous set: one of the constraints required that `num` be smaller than 5.

In each iteration our operator was able to correctly identify the semantic difference and produce a corresponding diff-witness. As the *distinct sort class strategy* requires the instantiation of every class, each diff-witness contained precisely one object for each class in the class diagram. Exactly

Table 1 Runtimes of the OCL differencing operations on constructed CDs

Model Size	Runtime
11 classes	1 914 ms
22 classes	7 698 ms
33 classes	30 769 ms
44 classes	91 948 ms
55 classes	228 444 ms
66 classes	468 185 ms
77 classes	1 052 730 ms
88 classes	1 516 053 ms
99 classes	3 473 036 ms
110 classes	3 595 805 ms

one diff-witness was produced per iteration, corresponding to the modified constraint.

We measured the runtime of the diff-operation in milliseconds. The results are displayed in table 1 and fig. 7. As can be seen there, the runtime appears to grow exponentially with the model size. However, this increase in runtime can theoretically be offset by parallelizing the refinement checks for the constraints of the old model. This might be a necessary optimization for dealing with larger models in the future work.

10 Conclusion

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 this extended version of the paper, we additionally gave a comprehensive overview on alternative SMT-encoding strategies for CDs, and considered their application in our case-study. Furthermore, we also elaborated our discussion on semantic differencing of operation constraints, and extended our tool such that ODs can be used as input to describe valid or invalid data patterns. We also evaluated the scalability of our approach using constructed models of varying size.

10.1 Discussion of research results

With respect to RQ1 we have sufficiently demonstrated that `OCLDiff` is indeed capable of automatically detecting semantic differences and determining refinement between OCL model versions of small to moderate size (i.e., around 100 classes / associations/ constraints). To deal with larger models, further optimization of the operator is needed due to the exponential runtime growth. However, the issue might be addressed by simply paralellizing the refinement checks for individual constraints.

Concerning RQ2, our case study at least provides a proof-of-concept that our tool can be applied in a MDD context for refinement checking. However, further optimization and evaluation is necessary to determine if an SMT-based semantic differencing operator can be of use in practical MDD scenarios. Ideally, we envision the operator being integrated into a CI/CD pipeline to ensure refinement between model versions post-commit or alternatively as an IDE plugin for pre-commit checks. The diff-ODs could then also be used to automatically create test-cases, while the trace-ODs are processed to identify and highlight missing refinements of constraints across models.

10.2 Future work

For the future, we seek to define additional strategies for classes, associations, and inheritance relations and compare them to the existing strategies. 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` [5]. 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.

Moreover, we intend to integrate our tool into a larger toolchain where model versions are automatically checked for refinement post-commit in a CI/CD pipeline, diff-ODs are used as test-input, and trace-ODs for artifact 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 models, we might need to address it by making corresponding adjustments to our approach.

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

Table 3 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))))`

In order to check for semantic differences in larger models, we intend to parallelize the refinement checks for individual constraints.

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

Author contributions The SMT-based semantic differencing approach was developed by Max Stachon, Sebastian Stüber, and Valdes Voufo in regular meetings with feedback from Bernhard Rumpe. The initial draft of the manuscript was written by Valdes Voufo, who also implemented the tool utilized in this study. Max Stachon and Sebastian Stüber were responsible for significant revisions across all sections of the paper. Funding and support were provided by Bernhard Rumpe. All authors participated in reviewing the final version of the manuscript. The authors are listed in alphabetical order.

Funding Open Access funding enabled and organized by Projekt DEAL. Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 250902306

Data availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

Ethics approval and consent to participate Not applicable

Consent for publication Present

Materials availability Available on Github, see section 8.4

Code availability Available on Github, see section 8.4

References

1. Hickey AM, Davis AM (2004) A unified model of requirements elicitation. J Manag Inf Syst 20:65–84. https://doi.org/10.1080/07421222.2004.11045786
2. Rumpe B (2017) Agile modeling with UML: code generation. Testing, Refactoring (Springer International
3. Sunitha E, Samuel P (2018) Object constraint language for code generation from activity models. Inf Softw Technol 103:92–111
4. Maoz S, Ringert JO, Rumpe B (2010) A Manifesto for Semantic Model Differencing. Proceedings Int. Workshop on Models and Evolution (ME'10), LNCS 6627, Springer, pp 194–203
5. Ringert JO, Rumpe B, Stachon M (2023) On implementing open world semantic differencing for class diagrams. J Object Technol (JOT) 22(2):1–14
6. Group OM (2017) Omg unified modeling language (omg uml)
7. Rumpe B (2016) Modeling with UML: language. Concepts, Methods (Springer International
8. Barrett C, Stump A, Tinelli C et al (2010) The smt-lib standard: version 2.0. Proceedings of the 8th international workshop on satisfiability modulo theories Vol 13, Edinburgh, UK, pp 14
9. De Moura L, Bjørner N (2008) Z3: an efficient smt solver. International conference on tools and algorithms for the construction and analysis of systems, Springer, pp 337–340
10. Rumpe B, Stachon M, Stüber S, Voufo V (2024) Semantic difference analysis with invariant tracing for class diagrams extended by OCL. Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa), MODELS Companion '24, ACM, pp 1066–1075
11. Kent S, Evans A, Rumpe B, Moreira A, Demeyer S (1999) UML Semantics FAQ. In: Moreira A, Demeyer S (eds) Object-Oriented Technology, ECOOP'99 Workshop Reader, LNCS 1743 Springer Verlag, Berlin
12. Richters M, Gogolla M, Evans A, Kent S, Selic B (2000) Validating uml models and ocl constraints. In: Evans A, Kent S, Selic B (eds) UML 2000 — The Unified Modeling Language, Springer, Berlin, Heidelberg, pp 265–277
13. Richters M, Gogolla M, Ling T-W, Ram S, Li Lee M (1998) On formalizing the uml object constraint language ocl. In: Ling T-W, Ram S, Li Lee M (eds) Conceptual Modeling – ER '98, Springer, Heidelberg, Berlin, pp 449–464
14. Soeken M, Wille R, Kuhlmann M, Gogolla M, Drechsler R (2010) Verifying uml/ocl models using boolean satisfiability. 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010), pp 1341–1344
15. Kuhlmann M, Gogolla M, France RB, Kazmeier J, Breu R, Atkinson C (2012) From uml and ocl to relational logic and back. In: France RB, Kazmeier J, Breu R, Atkinson C (eds) Model driven engineering languages and systems, Springer, Berlin, Heidelberg, pp 415–431
16. Torlak E, Jackson D (2007) Kodkod: a relational model finder. 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, pp 632–647
17. Nijjar J, Bultan T (2012) Unbounded data model verification using smt solvers. 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp 210–219
18. Cabot J, Clarisó R, Riera D (2008) Verification of uml/ocl class diagrams using constraint programming. 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, pp 73–80
19. Cabot J, Clarisó R, Riera D (2014) On the verification of uml/ocl class diagrams using constraint programming. J Syst Softw 93:1–23
20. Pérez B, Porres I (2019) Reasoning about uml/ocl class diagrams using constraint logic programming and formula. Inf Syst 81:152–177
21. Jackson E, Schulte W (2013) Formula 2.0: a language for formal specifications. Unifying Theories of Programming and Formal Engineering Methods, Springer, Berlin Heidelberg, pp 156–206. https://www.microsoft.com/en-us/research/publication/formula-2-0-language-formal-specifications/
22. Wu H (2023) Qmaxuse: a new tool for verifying uml class diagrams and ocl invariants. Sci Comput Program 228:102955
23. Wu H, Farrell M (2021) A formal approach to finding inconsistencies in a metamodel. Softw Syst Model 20:1271–1298
24. Gogolla M, Büttner F, Richters M (2007) Use: a uml-based specification environment for validating uml and ocl. Sci Comput Program 69:27–34
25. Gogolla M, Hilken F (2016) In Model validation and verification options in a contemporary uml and ocl analysis tool Modellierung 2016, Gesellschaft für Informatik e.V, Bonn, pp 205–220
26. Maoz S, Ringert JO, Rumpe B, Mezini M (2011) CDDiff: semantic differencing for class diagrams. In: Mezini M (ed) ECOOP 2011 - Object-Oriented Programming, Springer, Berlin Heidelberg, pp 230–254
27. Jackson D (2006) Software abstractions: logic, language, and analysis, The MIT Press
28. Kautz O, Maoz S, Ringert JO, Rumpe B (2017) CD2Alloy: a translation of class diagrams to alloy. Technical Report AIB-2017-06, RWTH Aachen University
29. Nachmann I, Rumpe B, Stachon M, Stüber S (2022) Open-World loose semantics of class diagrams as basis for semantic differences. Modellierung 2022, Gesellschaft für Informatik e.V., pp 111–127
30. Drave I, Kautz O, Michael J, Rumpe B, Berger T et al (2019) Semantic evolution analysis of feature models. In: Berger T. et al (eds) International systems and software product line conference (SPLC'19), ACM, pp 245–255
31. Maoz S, Ringert JO, Rumpe B (2011) ADDiff: semantic differencing for activity diagrams. Conference on foundations of software engineering (ESEC/FSE '11), ACM, pp 179–189
32. Maoz S, Ringert JO, Rumpe B. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany (2011)
33. Kautz O, Rumpe B (2018) Semantic Differencing of Activity Diagrams by a Translation into Finite Automata. Proceedings of MODELS 2018. Workshop ME
34. Drave I, Eikermann R, Kautz O, Rumpe B, Hammoudi S, Pires LF, Selić B (2019) Semantic differencing of statecharts for object-oriented systems. In: Hammoudi S, Pires LF, Selić B (eds) In: Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19), SciTePress, pp 274–282
35. Kautz O (2021) Model analyses based on semantic differencing and automatic model repair aachener informatik-berichte, Software Engineering, Band 46, Shaker Verlag
36. Butting A, Kautz O, Rumpe B, Wortmann A (2017) Semantic differencing for message-driven component & connector architectures. International conference on software architecture (ICSA'17), IEEE, pp 145–154
37. Rumpe B (2011) Modellierung mit UML, 2te edn. Springer, Berlin

38. Cengarle MV, Grönniger H, Rumpe B (2008) System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany
39. Fahrenberg U, Acher M, Legay A, Wasowski A Gnesi S, Rensink A (2014) Sound merging and differencing for class diagrams. In: Gnesi S, Rensink A (eds) Fundamental approaches to software engineering, Springer, Berlin, Heidelberg, pp 63–78
40. Lindt A, Rumpe B, Stachon M, Stüber S (2023) CDMerge: semantically sound merging of class diagrams for software component integration. J Object Tech (JOT) 22(2):1–14
41. Cook SA (1971) The complexity of theorem-proving procedures. Proceedings of the third annual ACM symposium on Theory of computing, pp 151–158
42. Mukherjee R, Kroening D, Melham T (2015) Hardware verification using software analyzers. 2015 IEEE Computer Society Annual Symposium on VLSI, IEEE, pp 7–12
43. Godefroid P, Levin MY, Molnar D (2012) Sage: whitebox fuzzing for security testing: sage has had a remarkable impact at microsoft. Queue 10:20–27
44. Tillmann N, De Halleux J (2008) Pex–white box test generation for. net. Tests and Proofs: Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings 2, Springer, pp 134–153
45. Bjørner N, Jayaraman K (2015) Checking cloud contracts in microsoft azure. Distributed Computing and Internet Technology: 11th International Conference, ICDCIT 2015, Bhubaneswar, India, February 5-8, 2015. Proceedings 11, Springer, pp 21–32
46. Gamma E, Johnson R, Helm R, Johnson RE, Vlissides J (1995) Design patterns: elements of reusable object-oriented software, Pearson Deutschland, GmbH
47. Barrett C, et al (2011) Cvc4. International conference on computer aided verification, Springer, pp 171–177
48. Deters M, Reynolds A, King T, Barrett C, Tinelli C (2014) A tour of cvc4: how it works, and how to use it. 2014 Formal Methods in Computer-Aided Design (FMCAD), IEEE, pp 7–7
49. Bansal K, Reynolds A, Barrett C, Tinelli C (2016) A new decision procedure for finite sets and cardinality constraints in smt. Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27–July 2, 2016, Proceedings, Springer, pp 82–98
50. Rumpe B (2012) Agile Modellierung mit UML: codegenerierung, Testfälle, Refactoring, 2nd edn. Springer, Berlin
51. Ahn G-J, Shin ME (2001) Role-based authorization constraints specification using object constraint language. Proceedings Tenth IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises. WET ICE 2001, IEEE, pp 157–162