Software and Systems Modeling (2021) 20:1271–1298
https://doi.org/10.1007/s10270-020-00849-8

REGULAR PAPER

A formal approach to finding inconsistencies in a metamodel

Hao Wu[1] · **Marie Farrell**[2]

Received: 21 December 2018 / Revised: 12 November 2020 / Accepted: 24 November 2020 / Published online: 29 January 2021

Abstract
Checking the consistency of a metamodel involves finding a valid metamodel instance that provably meets the set of constraints that are defined over the metamodel. These constraints are often specified in Object Constraint Language. Often, a metamodel is inconsistent due to conflicts among the constraints. Existing approaches and tools are typically incapable of pinpointing the conflicting constraints, and this makes it difficult for users to debug and fix their metamodels. In this paper, we present a formal approach for locating conflicting constraints in inconsistent metamodels. Our approach has four distinct features: (1) users can rank individual metamodel features using their own domain-specific knowledge, (2) we transform these ranked features to a weighted maximum satisfiability modulo theories problem and solve it to compute the set of maximum achievable features, (3) we pinpoint the conflicting constraints by solving the set cover problem using a novel algorithm, and (4) we have implemented our approach into a fully automated tool called MaxUSE. Our evaluation results, using our assembled set of benchmarks, demonstrate the scalability of our work and that it is capable of efficiently finding conflicting constraints.

Keywords Metamodel · Conflicts · SMT

1 Introduction

Metamodelling plays a key role in model-driven engineering (MDE), it paves the way for many other MDE approaches including model transformation, language engineering and business process modelling [8,47,84]. A metamodel captures the syntax for a set of *models* and allows users to construct a system design at a higher level of abstraction. A valid model, or an *instance* of a metamodel, is one that conforms to all of the constraints imposed by its features. These constraints vary according to the structural features of the metamodel such as multiplicities for an association or class invariants written in Object Constraint Language (OCL). Then the task for checking the consistency of a metamodel becomes finding a valid instance. This is challenging because an instance must meet all of the constraints that are defined over that metamodel. Recent studies have shown that this task can be tackled using well-engineered constraint solvers [51,76,83].

In practice, many metamodels are not consistent. This is caused by conflicting constraints that are imposed by different features such as the multiplicities of an association or class invariants. These conflicts can be a result of user error or of features being over-constrained in the design. Conflicts are more likely to occur when a metamodel defines a large number of constraints interleaving over different features. In this case, current modelling tools and approaches are unable to assist users in identifying the constraints that cause inconsistencies. These tools usually terminate and report inconsistent metamodels, or are unable to generate a valid instance. Without automated tool support, it is extremely difficult for users to pinpoint the conflicts in the constraints. It is often helpful for users to know how many metamodel features can be fulfilled in their current design so that they might use this information to further refine their metamodels. For exam-

We dedicate this paper to the memory of our colleague and mentor, James F. Power.

Communicated by Oystein Haugen.

This work is partially supported by Grant EP/R026092 (FAIR-SPACE Hub) through UKRI under the Industry Strategic Challenge Fund (ISCF) for Robotics and AI Hubs in Extreme and Hazardous Environments.

✉ Hao Wu
haowu@cs.nuim.ie

Marie Farrell
marie.farrell@manchester.ac.uk

[1] Computer Science Department, Maynooth University, Maynooth, Ireland

[2] Computer Science Department, University of Manchester, Manchester, UK

ple, a user may be interested in maximising the number of features that can be satisfied in their metamodel and remove those that cannot be satisfied. In realistic scenarios, users may employ their domain-specific knowledge to rank individual features and search for a model that fulfils as many of the higher ranked features as possible.

In this paper, we present an approach to dealing with metamodel inconsistency caused by conflicting constraints. Our approach aims to provide the user with two distinct pieces of information: (1) the set of achievable metamodel features and (2) the set of structural constraints or class invariants that cause conflicts. By identifying the former, users can either compute a model that contains as many achievable features as possible or find a model that conforms to the most desirable features based on their rankings. We distinguish our approach from the literature on checking consistencies of a metamodel in that we focus on the information that should be presented to the user when a metamodel is inconsistent.

To be precise, we require every (non-abstract) class to be instantiated at least once in the same configuration. If there exists one (non-abstract) or more classes that cannot be instantiated, while preserving all of the imposed constraints, we then consider a metamodel to be *inconsistent* or to have a *conflict*. Throughout the remainder of this paper, we strictly adhere to this notion of conflict.

We believe that providing the set of achievable features and conflicting constraints is useful and will help users to further refine their metamodels by locating and understanding the cause of the constraint conflicts. We compute both of these pieces of information using an SMT solver. The use of an SMT solver has several advantages. First, we can perform *fast* satisfiability checks on not only pure Boolean constraints but also complex structures with a number of numeric constraints. Second, it does not introduce a substantial implementation overhead since an SMT solver is treated as a *black-box* engine. Third, with recent advances in SMT solving, SMT solvers have been proven to be widely adopted in different software engineering research projects such as program synthesis/analysis, test case generation and in the education domain, to name but a few [30,39,40,46,68,73].

This paper extends and builds upon our previous work where we introduced an algorithm for locating the conflicts among different constraints [79]. This paper expands this work by presenting our complete techniques in greater detail, including refined examples, full formalisations, proofs of the correctness of our formulas and algorithms, and the implementation of our MaxUSE tool. We have also performed a new evaluation of our automated tool, MaxUSE, with the latest SMT-solving techniques. We present our new findings and discuss the strengths and limitations of our techniques.

Overall, the contributions and organisation of this paper are as follows:

1. We introduce a set of annotations, in Sect. 4, that allows users to rank individual metamodel features, including OCL constraints, based on their own domain-specific knowledge. These annotations categorise metamodel features into *soft* and *hard* features that are essential to our algorithm for maximising the number of metamodel features based on different ranks.
2. In Sect. 5, we present a set of SMT encodings for different metamodel features including classes, associations, and OCL invariants. With these encodings, we can now encode individual ranked metamodel features.
3. We present an algorithm for computing the maximised number of metamodel features by solving ranked OCL constraints in Sect. 6. This algorithm reduces ranked OCL constraints to a weighted maximum satisfiability modulo theories (MaxSMT) problem and solves it using a binary-search-based technique. Moreover, we provide a proof of correctness of this algorithm.
4. Based on computed weighted MaxSMT solutions, we present an algorithm for finding constraint conflicts by solving the set cover problem in Sect. 7. We show that our reduction from the set cover problem to SMT is correct.
5. We introduce our fully automated prototype tool, MaxUSE, in Sect. 8 by discussing its architecture and core parts with a detailed, illustrative example.
6. We propose a benchmark that can be used for evaluating scalability and performance. We evaluate our MaxUSE tool (with the latest version of the SMT solver integrated) against this proposed benchmark in Sect. 9. Furthermore, we discuss our findings and MaxUSE's capabilities in terms of usability, scalability and performance in Sect. 10.

In order to frame our subsequent discussions, we begin by introducing a motivating example in Sect. 2 and present an overview of our approach in Sect. 3.

2 A motivating example

In this section, we provide a small example that will be used throughout this paper to illustrate and motivate our approach at a high level. In this example, we use a UML class diagram to depict our metamodel for the purpose of discussing a real world example. Specifically, we use UML as a notation to capture the metamodel which describes the grammar for the structures in question.

This example is illustrated in Fig. 1, and represents a metamodel that models a real world example of the multiple relationships among the **University**, **Department**, **Student**, **Child** and **Module** entities. For example, a **Child** does not attend a **University**. However, the **Student**s can attend a **University** and choose multiple **Module**s to study. On the other

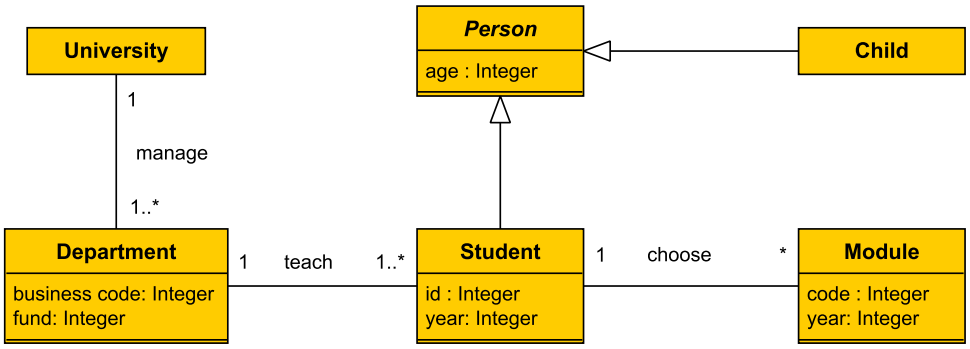


Fig. 1 An example of a metamodel describing the relationships among the **University**, **Department**, **Student**, **Child** and **Module** entities. Here, the **Person** class is abstract, and a **University** contains multiple **Departments** which, in turn, contain many **Students** that take many **Modules**

Table 1 The descriptions for the 8 class invariants defined for the metamodel shown in Fig. 1 with their corresponding enumerative surrogates

Description	Reference
Every person's age is between 0 and 18	$inv1$
The minimum student age is 18	$inv2$
Year range is between 1 and 6	$inv3$
Student numbers are unique	$inv4$
Students can only select modules which correspond to their year	$inv5$
There are both research and non-research students	$inv6$
All students must take modules	$inv7$
Modules are for students between years 1 and 5	$inv8$

hand, a **Department** can teach many **Student**s. Besides the associations among different classes capturing multiple relationships, this metamodel is also enriched with 8 different class invariants. Throughout the remainder of this paper, we refer to these invariants by number in order to keep the text clear and concise. However, we provide Table 1, which contains a brief description of each of these invariants, that can be used as a reference point for the reader. Each class invariant expresses a constraint over this metamodel. For example, $inv4$ in Fig. 2a states that each **Student** must have a unique `id` number. The fifth invariant, $inv5$, indicates that every **Student** can only choose **Module**s that are in their year. In this example, we use numbers 1 to 6 to distinguish a **Student**'s year, and **Student**s that are in year 6 are considered as research **Student**s. Thus, invariant $inv6$ specifies that there must exist some research students and non-research students.

Unfortunately, this metamodel is inconsistent because not every class invariant that is defined here can be achieved. Thus, we cannot generate valid instances from this metamodel. In order to fix this inconsistent metamodel, a user may wish to know two pieces of information. First, the *maximum* number of class invariants that can be achieved in the current design. This gives the user a clear idea about how robust their current design is. The second thing that the user may want to know is the *exact* class invariant(s) that cause the

```
context Person
inv1: Person.allInstances()->forAll(p|p.age>0 and p.age<18)

context Student
inv2: self.age>18
inv3: self.year>=1 and self.year<=6
inv4: Student.allInstances()->forAll
(s1,s2:Student|s1<>s2 implies s1.id <> s2.id)
inv5: Student.allInstances()->forAll(s|s.modules->
forAll(m|s.year=m.year))
inv6: Student.allInstances()->exists(s|s.year=6) and
Student.allInstances()->exists(s|s.year<6)
inv7: Student.allInstances()->forAll(s|s.modules->notEmpty())

context Module
inv8: self.year>=1 and self.year<=5
```

(a) The 8 class invariants for the metamodel illustrated in Figure 1. These class invariants impose constraints on **Student**s choosing **Module**s.

```
context Person
@Rank=4
inv1: Person.allInstances()->forAll(p|p.age>0 and p.age<18)

@StudentRank{Rank=5}
context Student
inv2: self.age>18
inv3: self.year>=1 and self.year<=6
inv4: Student.allInstances()->forAll
(s1,s2:Student|s1<>s2 implies s1.id <> s2.id)
@Rank=6
inv5: Student.allInstances()->forAll(s|s.modules->
forAll(m|s.year=m.year))
inv6: Student.allInstances()->exists(s|s.year=6) and
Student.allInstances()->exists(s|s.year<6)
inv7: Student.allInstances()->forAll(s|s.modules->notEmpty())

context Module
inv8: self.year>=1 and self.year<=5
```

(b) The 8 class invariants are annotated with the ranks (shaded area) for the metamodel that is illustrated in Figure 1. The higher the rank of a class invariant, the more important it is.

Fig. 2 An example of two types of OCL constraints (unranked on the left and ranked on the right) that are used in the metamodel shown in Fig. 1. The conflicts computed from our approach are: ($inv1$, $inv2$) and ($inv5$, $inv6$, $inv7$ and $inv8$)

```
context Person
inv1: Person.allInstances()->forAll(p|p.age>0 and p.age<18)

context Student
inv2: self.age>18
inv3: self.year>=1 and self.year<=6
inv4: Student.allInstances()->forAll
(s1,s2:Student|s1<>s2 implies s1.id <> s2.id)
inv5: Student.allInstances()->forAll(s|s.modules->
forAll(m|s.year=m.year))
inv6: Student.allInstances()->exists(s|s.year=6) and
Student.allInstances()->exists(s|s.year<6)
inv7: Student.allInstances()->forAll(s|s.modules->notEmpty())

context Module
inv8: self.year>=1 and self.year<=5
```

(a) Sample Solution 1: $inv2$ and $inv5$ cannot be satisfied. This is because $inv1$ and $inv2$ conflict, and $inv5$ conflicts with $inv6$, $inv7$ and $inv8$.

```
context Person
inv1: Person.allInstances()->forAll(p|p.age>0 and p.age<18)

context Student
inv2: self.age>18
inv3: self.year>=1 and self.year<=6
inv4: Student.allInstances()->forAll
(s1,s2:Student|s1<>s2 implies s1.id <> s2.id)
inv5: Student.allInstances()->forAll(s|s.modules->
forAll(m|s.year=m.year))
inv6: Student.allInstances()->exists(s|s.year=6) and
Student.allInstances()->exists(s|s.year<6)
inv7: Student.allInstances()->forAll(s|s.modules->notEmpty())

context Module
inv8: self.year>=1 and self.year<=5
```

(b) Sample Solution 2: $inv1$ and $inv7$ cannot be satisfied. This is because $inv1$ conflicts with $inv2$, and $inv7$ conflicts with $inv5$, $inv6$ and $inv8$.

Fig. 3 Two sample solutions (computed by our MaxUSE tool) of achieving a maximum number of 6 class invariants defined in Fig. 2a. In each solution, the invariants in the red boxes are those that cannot be achieved

inconsistencies. This helps users to narrow down their search and investigate the reasons behind the inconsistencies.

Our approach allows us to compute these two pieces of information. In fact, we are able to achieve a maximum number of 6 class invariants for this metamodel. Figure 3 shows two possible ways of achieving 6 class invariants. For example, Fig. 3a shows that invariants $inv1$, $inv3$, $inv4$, $inv6$, $inv7$ and $inv8$ can be achievable. There are a total of 8 different ways (solutions) of achieving a maximum number of 6 class invariants as shown in Table 2. This is due to the two conflicts among the invariants in Fig. 2a. Let us first consider $inv1$ as defined in the **Person** abstract class and $inv2$ in the **Student** class. For example, when creating a student (student person only) whose age is greater than 18. Since the **Person** class is abstract and a student is an instance of the **Person** class, it inherits the invariant $inv1$ (it applies to both **Student** and **Child** class). However, $inv1$ states that for every single **Person** whose `age` is less than 18, while $inv2$ specifies that every **Student**'s `age` must be greater than 18. Thus, a conflict occurs between $inv1$ and $inv2$.[1] This conflict is easy to spot once the user identify that $inv1$ is accidentally defined under $Person$ class instead of $Child$ class.

The second conflict is not so easy to identify, even for experienced users. This conflict is caused by the invariants that there must exist some research and non-research **Student**s ($inv6$) choosing some modules ($inv7$) in their corresponding `year` ($inv5$). But, according to $inv8$, **Module**s are only available for non-research **Student**s ($inv8$: between `year` 1 and 5).

Table 2 A total of 8 different solutions (computed by our MaxUSE tool) of achieving a maximum number of 6 class invariants shown in Fig. 2a

Solutions	inv1	inv2	inv3	inv4	inv5	inv6	inv7	inv8
(1)	✓	✗	✓	✓	✗	✓	✓	✓
(2)	✓	✗	✓	✓	✓	✗	✓	✓
(3)	✓	✗	✓	✓	✓	✓	✗	✓
(4)	✓	✗	✓	✓	✓	✓	✓	✗
(5)	✗	✓	✓	✓	✗	✓	✓	✓
(6)	✗	✓	✓	✓	✓	✗	✓	✓
(7)	✗	✓	✓	✓	✓	✓	✗	✓
(8)	✗	✓	✓	✓	✓	✓	✓	✗

Here, we use a ✓ to indicate that an invariant is selected and a ✗ to denote a class invariant that cannot be met

In the real world, all constraints are not treated equally. In particular, some may be considered to be more important than others. Thus, we allow users to freely rank individual invariants using their own domain-specific knowledge. Then, our approach searches for all possible ways of maximising these ranks.

Consider the class invariants defined in Fig. 2a, one can rank them as in the example presented in Fig. 2b. The rank for each class invariant is highlighted in the shaded region. For example, $inv1$ is ranked with an integer value of 4 and $inv8$ is not ranked at all.[2] All other invariants ($inv2$, $inv3$, $inv4$, $inv6$ and $inv7$) defined under the **Student** class are ranked with an integer value of 5, except for $inv5$ which is ranked

[1] Based on our notion of conflicts, here we require both Student and Child to be instantiated in the same configuration.

[2] If an invariant is not ranked then it must be included in the solution. This will be explained further in Sect. 4.1.

```
context Person
@Rank=4
inv1: Person.allInstances()->forAll(p|p.age>0 and p.age<18)

@StudentRank{Rank=5}
context Student
inv2: self.age>18
inv3: self.year>=1 and self.year<=6
inv4: Student.allInstances()->forAll
(s1,s2:Student|s1<>s2 implies s1.id <> s2.id)
@Rank=6
inv5: Student.allInstances()->forAll(s|s.modules->
forAll(m|s.year=m.year))
inv6: Student.allInstances()->exists(s|s.year=6) and
Student.allInstances()->exists(s|s.year<6)
inv7: Student.allInstances()->forAll(s|s.modules->notEmpty())

context Module
inv8: self.year>=1 and self.year<=5
```

(a) Here, $inv2$, $inv3$, $inv4$, and $inv6$ can be achieved and each of them contributes to a rank of 5. $inv8$ is achieved but with no rank defined while $inv5$ is also achieved and contributes a rank of 6. In this case $inv1$ and $inv7$ cannot be achieved.

```
context Person
@Rank=4
inv1: Person.allInstances()->forAll(p|p.age>0 and p.age<18)

@StudentRank{Rank=5}
context Student
inv2: self.age>18
inv3: self.year>=1 and self.year<=6
inv4: Student.allInstances()->forAll
(s1,s2:Student|s1<>s2 implies s1.id <> s2.id)
@Rank=6
inv5: Student.allInstances()->forAll(s|s.modules->
forAll(m|s.year=m.year))
inv6: Student.allInstances()->exists(s|s.year=6) and
Student.allInstances()->exists(s|s.year<6)
inv7: Student.allInstances()->forAll(s|s.modules->notEmpty())

context Module
inv8: self.year>=1 and self.year<=5
```

(b) Here, $inv2$, $inv3$, $inv4$, and $inv7$ can be achieved and each of them contributes a rank of 5. $inv8$ is achieved but with no rank defined while $inv5$ is also achieved and contributes rank of 6. In this case $inv1$ and $inv6$ cannot be achieved.

Fig. 4 There are two ways of achieving a maximum total rank of 26 out of 35 (computed by our MaxUSE tool). The invariants in the blue dashed box are these that can be achieved and each of them contributes to the maximum total rank. The invariant in the green dashed box means that it can be achieved but with no rank defined. The invariants that are not in the dashed box are those that cannot be achieved

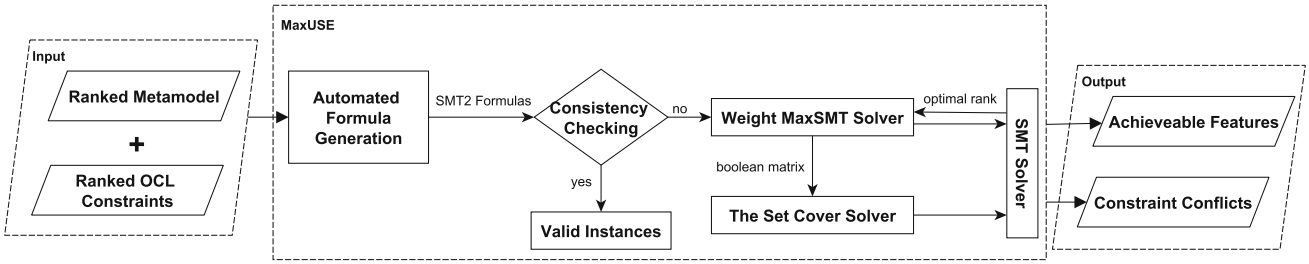


Fig. 5 Our approach reduces a metamodel along with ranked OCL constraints into an SMT problem. This is achieved by our automated formula generation engine. Core to our approach are two customised built-in solvers (Weighted MaxSMT and set cover solver) for solving two challenging problems: computing the maximum number of achievable features and finding minimum conflicts for OCL constraints

with a value of 6. This means that a **University** may consider that a registration procedure for **Student**s choosing **Module**s in their corresponding `year` ($inv5$) is more important than other constraints such as choosing some **Module**s ($inv7$) or having some non-research and research **Student**s ($inv6$). With these ranked invariants, we now have a total rank of 35 for this metamodel. Since not every invariant can be achieved due to the two conflicts: ($inv1$, $inv2$) and ($inv5$, $inv6$, $inv7$, $inv8$), it is impossible to achieve a total of rank 35. However, we can achieve a total of rank of 26 out of 35. In fact, there are two ways of achieving this rank of 26 as shown in Fig. 4. In particular, Fig. 4a shows one way of achieving a total rank of 26 by choosing $inv2$ over $inv1$ and $inv5$ over $inv7$, while Fig. 4b shows another way of achieving this rank by choosing $inv2$ over $inv1$ and $inv5$ over $inv6$.

3 Our approach

Figure 5 provides an overview of our approach to maximising the number of achievable features based on their ranks and locating conflicts among different constraints. For our approach to be successful, we require that users provide a well-formed metamodel along with syntactically correct OCL constraints. First, users use a set of pre-defined annotations to rank their metamodel features and the corresponding OCL constraints. The formula generation engine then automatically translates the metamodel and OCL constraints into a set of SMT2 formulas, and checks the satisfiability of these formulas. In other words, we perform consistency checking on both metamodel structural and OCL constraints here. If these formulas are satisfiable (**SAT**), then our approach

directly reports that the metamodel is consistent and that there are no conflicts among the defined constraints. Thus, we are able to generate valid instances conforming to the constraints defined for the metamodel.

If these formulas are unsatisfiable (**UNSAT**), then the metamodel is inconsistent. This means that there exists at least one conflict among the structural and OCL constraints. As mentioned in Sect. 1:

*a **conflict** occurs in a metamodel if at least one of the non-abstract classes cannot be instantiated while preserving all of the associated OCL constraints.*[3]

In this way, our approach tries to instantiate every *non-abstract* class in the same configuration. If there exists at least one (non-abstract) class that cannot be instantiated, while preserving all of the imposed constraints, then this metamodel has a conflict. For example, we try to instantiate every non-abstract class in Fig. 1. These classes are: **University**, **Student**, **Child**, **Module** and **Department**. However, it is impossible to instantiate the **Student** class due to the conflict between $inv1$ defined in the **Person** abstract class and $inv2$ defined in the **Student** class.

To compute the set of maximum number of achievable features, we first compute a maximum rank that can be achieved by casting it to the weighted maximum satisfiability problem (MaxSMT). The key step here is that our approach traverses every element and constraint in a metamodel and uses a weighted MaxSMT solver to find an optimal value for the ranks. This step employs a binary-search-based algorithm that iteratively asks an SMT solver for an optimal value. The returned solution here is a set containing all possible ways of maximising the ranks over a metamodel (namely, weighted MaxSMT solutions).

In order to pinpoint the exact conflicts among the metamodel features, our approach treats all features (classes, assertions and invariants) equally and casts them into the set cover problem [2,21,32]. This work is inspired by the work in [55] on computing minimal unsatisfiable subsets of constraints. Therefore, all features are treated equally here because the relationship between weighted MaxSMT solutions and constraint conflicts is captured by the set cover problem [55]. We form a Boolean matrix representing the set cover problem and then use a customised solver (namely, the set cover solver) to solve the set cover problem. This solver uses a novel algorithm that reduces the set cover problem to an SMT problem and solves it using an efficient SMT solver. The returned solution here is a set containing those constraints that cause conflicts.

[3] However, users can verify a metamodel in a specific configuration by issuing a specific query. This is a new feature that we will discuss in our next article.

$$
\begin{array}{lll}
AnnotationSpec & ::= & (BlockAnnotation \mid LineAnnotation)\ ?\ Feature \\
BlockAnnotation & ::= & @\mathbf{IDEN}\{AnnotationTag\} \\
LineAnnotation & ::= & @AnnotationTag \\
AnnotationTag & ::= & \mathbf{Rank} = Tag \\
Tag & ::= & \mathbf{automatic} \mid INT
\end{array}
$$

Fig. 6 The syntax of our annotations for ranking different metamodel features. Here, $Feature$ denotes a single metamodel feature that includes a class, an association and invariant. Here, **'IDEN'** denotes an identifier. **'Rank'** and **'automatic'** are two keywords

In the following sections, we describe the details about our formulations and the algorithms (Sects. 5, 6 and 7) used for computing these two kinds of information. We begin by introducing our annotations for ranking individual metamodel features in Sect. 4.

4 Annotation for ranking

In this section, we present our ranking annotations by first describing their syntax. Then we discuss ranking criteria and introduce an automated ranking process.

4.1 Annotation syntax

We provide a simple set of annotations that enable users to freely rank distinct metamodel features. The syntax of these annotations is outlined in Fig. 6. In general, a rank can be applied to each metamodel feature depending on different purposes. There are two distinct types of annotations: the line annotation and the block annotation. Each annotation denotes a specific rank for a metamodel feature. This includes classes, associations and OCL invariants. The rank for each feature must be a non-negative integer (i.e. $rank \in \mathbb{Z} \wedge 0 \leq rank$).

A line annotation is used for ranking one single metamodel feature. For example, in Fig. 2b, $inv1$ is given a rank of 4. A block annotation allows a user to rank a set of features with the same value. For example, the annotation used in Fig. 2b for the **Student** class ranks every class invariant with an integer value of 5. If a user wishes to overwrite a rank in a block annotation then a line annotation can be inserted. For example, the rank for $inv5$, in Fig. 2b, is overwritten with an integer value of 6.

We consider all ranked metamodel features as *soft features*. A soft feature with a higher rank is more favourable than a feature with a lower rank during the search. For example, $inv5$ in Fig. 2b is more likely to be chosen over $inv7$ (see Fig. 4a). On the other hand, if a feature is *not* ranked, then we say that it is a *hard feature*. This means that it *must not* be ignored during the search. For example, $inv8$ in Fig. 2b must hold in all situations. Thus, these annotations enable a user to specify a set of soft and hard features over a metamodel.

4.2 Ranking criteria

A metamodel feature can be ranked in two ways: (1) users rank an individual metamodel feature as a soft feature based on their domain-specific knowledge, or (2) in situations where users wish the program to automatically handle a particular feature for them, an automatic ranking criteria is provided.

When users rank a metamodel, there are 3 different scenarios that our approach computes and these are summarised below.

1. A metamodel is partially ranked. This means that it contains a mixture of soft and hard features such as the one presented in Fig. 2b. In this case, our approach computes the maximum rank for all of the soft features defined and satisfies the hard features.
2. A metamodel is totally ranked. This means that every single metamodel feature is ranked with some integer value. Our approach tries to find a maximum total rank for all achievable metamodel features. If the metamodel is consistent, then the maximum rank can be achieved and is equal to the sum of the ranks of all of the metamodel features.
3. A metamodel is not ranked at all. This means that every single metamodel feature must be considered during the search. In this scenario, our approach only performs the consistency checking for the metamodel, since no features can be ignored during the search.

In order to pinpoint the set of exact conflicts, we require that the user rank every single metamodel feature with the same value. In other words, every feature is treated equally and no feature is more important than another.

4.3 Automatic ranking

We now present our automatic ranking that provides a quick and simple ranking scheme for users who do not have domain-specific knowledge of the system being modelled to begin with. This automatic ranking should then be overridden once the user's domain-specific knowledge increases. In this way, our automatic ranking provides an initial ranking that could then be further refined.

By default all metamodel features are initially treated as hard features. However, users may override the default settings using the **'automatic'** keyword. All features annotated with **'automatic'** are assigned a specific value internally and automatically calculated using the following set of rules.

- For a non-abstract class, we compute its rank based on counting the number of attributes and operations (including those inherited from an abstract class) defined within. We take this view because a class (non-abstract) that contains more attributes and operations typically describes more information about a system than a class with fewer attributes and operations.
- For an association, the rank is calculated by adding up the rank defined on each association end. Currently, we require that each association end is owned by a class.
- For a class invariant, we calculate the size of its abstract syntax tree (AST) by counting the number of nodes. The larger the size of an invariant's AST, the more likely it is that a complex constraint will be imposed on a metamodel. Though an invariant could be written in multiple ways, we assume that users write all class invariants in a consistent manner. For example, using $self$ to constrain attributes and $allInstances()$ for quantifiers and navigations. Thus, an invariant with a large AST will have a higher rank than an invariant with a small AST. For example, the invariant in Fig. 7 is automatically ranked. In Fig. 8, we can see that its AST has 12 nodes in total. Thus, this invariant is ranked with a value of 12.

```
@Rank = automatic
inv5: Student.allInstances()->forAll(s|s.modules->
forAll(m|s.year=m.year))
```

Fig. 7 An example of using the **'automatic'** ranking keyword to rank a class invariant

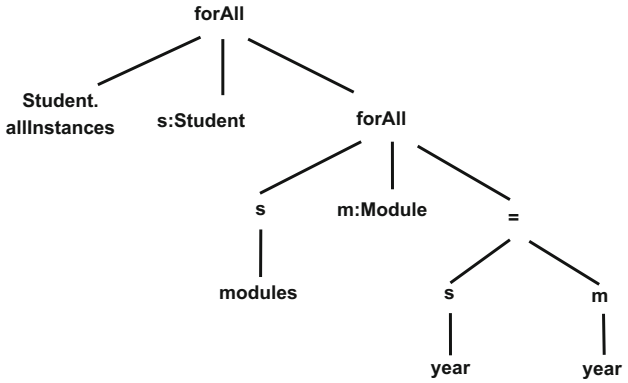


Fig. 8 The abstract syntax tree for the invariant in Fig. 7 has a total of 12 nodes

In order to compute the precise size of an OCL abstract syntax tree, we implement an algorithm that iteratively records the node seen during the traversal. This algorithm uses a visitor pattern to traverse each OCL expression's AST and calculate its size.

5 Generating SMT formulas from a metamodel

In this section, we describe how we reduce all of the ranked metamodel features to a weighted MaxSMT problem. First we encode different types of metamodel features into SMT formulas. We then decompose an SMT formula into two parts. The first part is an SMT encoding of a specific metamodel feature including classes, attributes, associations and class invariants. The second part is an SMT formula that represents the corresponding rank for each metamodel feature so that this can be successfully solved using an SMT solver. We begin by introducing the first part of our SMT encoding in this section. This includes our SMT encoding for classes, associations and class invariants.

5.1 Classes and attributes

Each class that is defined in a metamodel has its own attributes and can be instantiated to create a specific object. In order to capture a set of objects that have the same type, we must show our corresponding SMT encodings. To encode an object we use an object function, O_{id}, and a type function per class, T_{class}, to represent an object's unique id and its type, respectively. These two functions are defined as follows:

$$\begin{aligned} O_{id} &: INT \rightarrow INT \\ T_{class} &: INT \rightarrow BOOL \end{aligned}$$

The intuition here is that we use an integer to represent the memory address of an object/instance, o. The functions O_{id} and T_{class} are then designed for dereferencing o and ascertaining its type, respectively. For example, the following axiom checks whether an object p_1 is of type $Person$ and that the object p_2 is *not* a $Person$.

$$T_{Person}(O_{id}(p_1)) \wedge \neg T_{Person}(O_{id}(p_2))$$

With functions O_{id} and T_{class}, we now define a function per attribute, F_{attr}, of the following form to access an attribute of an object.

$$F_{attr} : INT \rightarrow T_{attr}$$

Here the input argument is a unique object id (captured by the function O_{id}) and T_{attr} represents the attribute's type. The intuition here is that, given an object id, the function F is able to return its object's attribute. For example, given an object p of type $Person$ in Fig. 1, the property which specifies that the age attribute corresponding to p is equal to 18 is formalised as:

$$\big(F_{age}(O_{id}(p)) = 18\big) \wedge \big(T_{Person}(O_{id}(p))\big)$$

Thus we have presented our SMT encodings for classes and attributes. Next, we show how to encode the relationships among different classes.

5.2 Relationships: inheritance and associations

Each class that is defined in a metamodel may be related to other classes. These relationships are typically depicted using inheritance and associations among classes. Hence, in order to capture such relationships, we show our SMT encodings.

Since we can use the type function, T_{class}, that we have defined above to determine an object's type, we can build a general form for inheritance. Generally, if a class B inherits from A (denoted by $B \prec A$), then every attribute (with protected and public modifiers) in A is also in B. In other words, an instance of B is also an instance of A. Thus, we use Formula 1 as follows to capture this semantics.

$$\bigwedge_{i=1}^{|G|} T_{B_i}(O_{id}(b)) \wedge T_A(O_{id}(b)) \tag{1}$$

where $B_i \in G, i \in \mathbb{N}$ and $B_i \prec A$. Here, G is a generalisation set[4] that contains a set of subclasses of A. For each subclass B_i and its instance b, the type of b is B_i but also is A. For a disjoint relation, we simply require that no instances (of a subclass) can be of any other specific subclasses. This is done by adding one additional axiom.

To encode an association, we introduce a relational function, Rel, to relate two objects.

$$Rel : INT \times INT \rightarrow BOOL$$

In fact, we only consider binary associations since n-ary associations can be decomposed into multiple binary associations. The rule (encoding) here is that if two classes A and B are associated with each other, then their instances might also be linked. Since a binary association relates two objects, we use Formula 2 to add an additional axiom stating that if an instance a of class A is associated with an instance b of class B, then b is linked with a.

$$\bigwedge_{i=1}^{|S|} Rel_{c_i}(O_{id}(a), O_{id}(b)) = Rel_{c_i}(O_{id}(b), O_{id}(a)) \tag{2}$$

Here S denotes the set of binary associations and $c_i \in S$. A and B are two classes at two association ends. Then a and b are the instances of A and B respectively. Note that Rel here does not necessarily encode the direction of a binary association. However, this directional information can be reflected

[4] Note that we only consider the scenario where G is incomplete and disjoint.

Fig. 9 A general form of a binary association. Here, R denotes the association name. Then $l1 \ldots u1$ and $l2 \ldots u2$ denote the lower and upper bounds at two association ends

by the information used in a metamodel. For example, users may define $Rel_{parent}(a, b)$ as a is a parent of b, or conversely, as b is a parent of a.

In order to capture the semantics of multiplicities, we introduce a cardinality function, $Card$, to express the lower and upper bound of a class that can be associated with another. Given a binary association, R, as shown in Fig. 9, we construct Formula 3:

$$\begin{aligned} &Card(X_A) \geq R.l_1 \ \wedge \ Card(X_A) \leq R.u_2 \\ &\wedge \ Card(X_B) \geq R.l_2 \ \wedge \ Card(X_B) \leq R.u_2 \end{aligned} \tag{3}$$

Since cardinality constraints are numerical, we directly use two integers X_A and X_B to encode two collections of objects from both ends of an association. X_A here represents the set of objects connecting to the instances of class B. Similarly, X_B represents the set of objects connecting to the instances of class A. Thus, $Card$ is directly applied to a collection rather than using a quantified formula as was the case in previous formulas. This is because our cardinality function returns the number of elements in a collection. In fact, we could use linear inequalities to reason about cardinality constraints and this process can be carried out separately from the relations used for associations [5,70]. Differently, we also consider OCL invariants here such as: the $size$ operator over a collection. However, the encoding presented here does not reflect the detailed links at object level. For readers who are interested in the detailed SMT encodings at object level, we refer to [78,83]. Next, we show how class invariants can be encoded.

5.3 Class invariants

For class invariants that are written in OCL, we support a wide range of constructs including navigation, nested quantifiers and operations on generic collection data types such as $include$. These encodings are generally in first-order form. Currently, we do not support $string$ operations.

Since each class invariant specifies a constraint that all instances of that class must conform to, we introduce a $\forall$ quantifier in our SMT encoding. In general, each class invariant is encoded in the following form:

$$\forall P_1 : INT, \ldots, P_n : INT \cdot \bigwedge_{i=1}^{n} T_{class}(O_{id}(P_i)) \Rightarrow Expr \tag{4}$$

The list of objects $(P_1, \ldots, P_n)$ are bounded by the $\forall$ quantifier ($allInstances()$). The implication simply means that if an object is an instance (type) of class, then the corresponding SMT formula ($Expr$) is implied.

For example, the invariant ($inv3$) for the **Student** class in Fig. 2a is encoded as follows:

$$\begin{aligned} &\forall P : INT \cdot \ T_{Person}(O_{id}(P)) \wedge T_{Student}(O_{id}(P)) \Rightarrow \\ &F_{year}(O_{id}(P)) \geq 1 \wedge F_{year}(O_{id}(P)) \leq 6 \end{aligned}$$

The conjunction here is used to express that P is also an instance of both the **Person** and **Student** classes. Note that this was captured by our use of the `self` keyword in $inv3$. For operations such as $include()$ and $notEmpty()$ over a collection data types, our encoding is similar to that used in [24]. We use either a quantified predicate or function to encode an operation depending on the specific receiver of that operation. For example, we use an existential quantifier ($\exists$) for the $notEmpty()$ operation.

6 Reducing to weighted MaxSMT

Each of our SMT formulas consists of two parts, the first is an SMT formula for a specific metamodel feature such as those described in the previous section. The second part of the formula is central to our approach. Using the formulas generated for this part, we are able to apply a rank for a specific metamodel feature when the constraint imposed by that feature is achievable. In other words, the second part of our formula enables us to transform the SMT formulas (described in the previous section) into a weighted MaxSMT problem that we can solve using an efficient SMT solver. Our reduction to SMT is a procedure that traverses the set of soft features defined on a metamodel and automatically generates a set of SMT formulas.

6.1 Forming weighted MaxSMT

The key idea of reducing ranked metamodel features into a weighted MaxSMT problem is that we introduce an integer type auxiliary variable for each SMT formula that encodes a feature. To be precise, given a total of $k \in \mathbb{N}$ soft features and letting F_i be an SMT formula that encodes the ith soft feature in a metamodel. We use an integer type auxiliary variable, Aux_i, whose range is $\{0, 1\}$. We then construct Formula 5.

$$\left(\bigwedge_{i=1}^{k} F_i \vee \underbrace{\left(Aux_i = 1 \right)}_{part\ a} \right) \wedge \underbrace{\left(\left(\sum_{i=1}^{k} Aux_i \right) = 0 \right)}_{part\ b} \tag{5}$$

The key insight here is that we associate each F_i with an auxiliary variable so that it is *equisatisfiable* to the original F_i. This is ensured by *part a* and *part b* of Formula 5 since both parts cannot be satisfied simultaneously. Therefore, we can check whether a feature encoded by Formula F_i is achievable via testing the satisfiability of Formula 5. To show that Formula 5 is equisatisfiable to F_i, we prove Lemma 1.

Lemma 1 *Testing the satisfiability of Formula 5 is equivalent to testing the satisfiability of F_i.*

Proof ($\Rightarrow_{sat}$): Suppose that Formula 5 is satisfiable, then the two subformulas at both ends of the conjunction must hold. Here, *part b* of Formula 5 constrains the summation of auxiliary variables to be equal to 0. Hence, it is impossible for *part a* of Formula 5 to be satisfied. In order to make Formula 5 satisfiable, F_i must be satisfiable because of the disjunction in the left subformula.
($\Leftarrow_{sat}$): Suppose that F_i is satisfiable, then in order to make Formula 5 satisfiable, the SMT solver must find an assignment that satisfies the two subformulas at both ends of the conjunction. The only way to achieve this is to assign zeros to all of the defined auxiliary variables (*part b*). This is because either *part a* or *part b* holds, but not both. Therefore, Formula 5 is satisfiable because F_i is satisfiable and *part b* is satisfiable.
($\Rightarrow_{unsat}$): Suppose that Formula 5 is unsatisfiable then there are 3 scenarios: (1) both the left and right subformulas of the conjunction are unsatisfiable including F_i, (2) if the right subformula (*part b*) of the conjunction is unsatisfiable, then the left subformula including *part a* could be satisfiable. This allows F_i to be satisfiable. However, by $\Leftarrow_{sat}$ this is impossible. That is, if F_i is satisfiable, then there exists a way of making Formula 5 satisfiable. Thus, F_i must be unsatisfiable and, (3) if the left subformula is unsatisfiable and the right subformula is satisfiable, then F_i must be unsatisfiable due to the disjunction in the left subformula.
($\Leftarrow_{unsat}$): Suppose that F_i is unsatisfiable. Since either *part a* or *part b* can be satisfied but not both, no matter which case here then Formula 5 must be unsatisfiable. □

We let V^{W_i} be an SMT encoding for a user-specified rank W_i of the ith soft feature. Note that $V^{W_i} \geq 0$ (no negative value is allowed). We now generate Formula 6.

$$\bigwedge_{i=1}^{k} \Big(\big((Aux_i = 0) \Rightarrow (V^{W_i} = c_i) \big) \wedge \big((Aux_i = 1) \Rightarrow (V^{W_i} = 0) \big) \Big) \tag{6}$$

where $c_i > 0$.

The implication of this formula is built on Formula 5. If Formula 5 is satisfiable, then each $Aux_i = 0$ and F_i must also be satisfiable. This means that the constraint imposed by the ith soft feature can be achieved. Thus, we assign an integer constant, c_i, to V^{W_i} to indicate that the corresponding rank is achieved. Otherwise, there must exist some F_is that are not satisfiable. In this case, we simply disable the corresponding rank by setting V^{W_i} to 0.

Finally, we form a weighted MaxSMT problem by generating Formula 7.

$$\underbrace{\Big(\Big(\sum_{i=1}^{k} Aux_i \Big) = m \Big)}_{part\ c} \wedge \underbrace{\Big(\Big(\sum_{i=1}^{k} V^{W_i} \Big) = c \Big)}_{part\ d}, \tag{7}$$

where $1 \leq m \leq k$ and $1 \leq c \leq \sum_{i=1}^{k} W_i$.

We generate this formula only when Formula 5 is not satisfiable. This is because if Formula 5 is satisfiable, then the metamodel is consistent. Intuitively, we know that some F_is are not satisfiable, and both *part a* and *part b* of Formula 5 cannot be satisfiable at the same time. Now to make Formula 5 become satisfiable, we remove *part b* (Formula 5) and rewrite it as *part c* (Formula 7). This forces some of the auxiliary variables (Aux_i in Formula 5) to be evaluated to 1.

In other words, we fix some number m and if there are some features that cannot be met, then the associated auxiliary variables (Aux_i) must be evaluated to 1 in order to be satisfiable. In this way we can work out m number of constraints imposed by the metamodel features that cannot be fulfilled. In the meantime, we also check whether it is possible to achieve a total rank of c based on the remaining number of metamodel features (*part d* of Formula 7). If c is the maximum number that we can find to make Formula 7 satisfiable, then c is a solution to our weighted MaxSMT problem.

Now, we have formed a weighted MaxSMT problem from a ranked metamodel, the goal here is to find a maximum total rank from all ranked metamodel features, namely a weighted MaxSMT solution.

6.2 Solving weighted MaxSMT problem

Each weighted MaxSMT solution is a set that represents a way of forming a set of achievable metamodel features. We use Algorithm 1 to find these sets. This algorithm first checks whether a metamodel is consistent or not (line 4). In order to achieve this, we add an extra axiom stating that there must exist some instances for every non-abstract class. This axiom is shown in Formula 8 and conjoined with the SMT encodings of different metamodel features (line 1). To be precise, this

formula states that given n non-abstract classes ($n \in \mathbb{N}$), then every class must be instantiated at least once.[5]

$$\bigwedge_{i=1}^{n} \exists P_i : INT \cdot T_{class}(O_{id}(P_i)) \tag{8}$$

If a metamodel is *not* consistent (line 4), then the algorithm uses a customised binary search algorithm to locate an achievable maximum rank from a total rank ($\sum_{i=1}^{k} W_i$) of all soft features (F_s). This binary-search algorithm iteratively checks the constant c in *part d* of Formula 7 with a new possible optimal value and asks an SMT solver to determine whether this is the maximum value that can be achieved (line 6). If it is, then the algorithm finds a way of maximising the total rank, namely a weighted MaxSMT solution.

It then enumerates all other possible ways of achieving this value (r_{opt} on line 6) by blocking all previous successful assignments (line 8) until no more weighted MaxSMT solutions can be found. Note that a metamodel could contain soft features that have the same rank. In other words, each feature is equally weighted. In this case, the algorithm enumerates all possible ways of achieving as many features as possible (maximising the number of achievable features). For example, there are 8 ways (Table 2) of achieving a maximum of 6 invariants (in Fig. 2) for the metamodel in Fig. 1.

Algorithm 1: This algorithm uses a customised binary search method to find the set of all achievable features for inconsistent metamodels.

Input : A set of SMT formulas ϕ encoding soft (F_s) and hard features (F_h), and a total rank $\sum_{i=1}^{k} W_i$ from k metamodel soft features.
Output: A set s containing all sets of achievable metamodel features.

1 $\phi \leftarrow \phi \wedge Formula\ 8$
2 $Solver.add(\phi \wedge Formula\ 5 \wedge Formula\ 6)$
3 $s \leftarrow \emptyset$
4 **if** $SMTSolve(\phi) =$ **UNSAT then**
5 // a metamodel is *not* consistent
6 $r_{opt} \leftarrow BinarySearch(Formula\ 7, F_s)$
7 **while** $SMTSolve\left(\left(\sum_{i=1}^{k} V^{W_i}\right)=r_{opt}\right) =$ **SAT do**
8 $Solver.add($**BlockingFormula**$)$
9 $s \leftarrow s \cup Interpret(Solver.model())$
10 **end**
11 **else**
12 // a metamodel is consistent
13 $s \leftarrow F_s \cup F_h$
14 **end**
15 **return** s

[5] In the case of inheritance: $B \prec A$ (B is subtype of A). We also require that the successful instantiation of B and creation of an instance of A alone that is not of type B.

Theorem 1 *Algorithm 1 finds all possible sets of achievable metamodel features that maximise the total ranks.*

Proof Let ϕ be the set of SMT formulas that capture soft and hard features. To prevent generating empty instances, Formula 8 is added to enforce class instantiation. By Lemma 1, we know that Formula 5 is equisatisfiable to some formulas ϕ_s that capture soft features. Formula 6 only adds extra constraints on the consequent assignments of auxiliary variables. Hence, the conjunction of three formulas on line 2 maintains the satisfiability of ϕ.
Case 1 When a metamodel is consistent, the set s simply returns all metamodel features (line 13). The maximum rank found is $\sum_{i=1}^{k} W_i$.
Case 2 When a metamodel is inconsistent, a binary search is employed to explore the search space. The lower bound and upper bound here are 1 and $\sum_{i=1}^{k} W_i - 1$, respectively. In other words, there exists $m \in \mathbb{N}$ constraints that cannot be satisfied. Each step in the binary search is a call to an SMT solver to test the satisfiability of ϕ with m number of constraints deactivated. Since the binary search guarantees to deterministically find the optimal value (r_{opt}), the following properties must hold for all possible total ranks, r:

- $\forall r \cdot r \geq 1 \wedge r > r_{opt} \Rightarrow \phi$ is unsatisfiable.
- $\forall r \cdot r \geq 1 \wedge r \leq r_{opt} \Rightarrow \phi$ is satisfiable.

Thus, the enumeration on the optimal value, r_{opt}, by rewriting *part d* of Formula 7 finds all possible other ways. □

7 Finding metamodel inconsistencies

It has been determined that the set of conflicts among SAT formulas can be captured by the set cover problem [55]. Inspired by this work, we find constraint conflicts of metamodel features by further solving the set cover problem using an SMT solver. A conflict among a set of metamodel features is essentially a *minimal unsat core* [55]. This core is a set of unsatisfiable SMT formulas and all *proper* subsets of this core are satisfiable. Although only a few SMT solvers provide unsat core extraction, such extraction is not guaranteed to find *all minimal* unsat cores [27]. In particular, the Z3 SMT solver only finds one conflict ($inv1$, $inv2$) for the example in Fig. 2a.

7.1 Mapping to the set cover problem

Formally, a set cover problem can be defined as follows: given a finite universe, $U = \{S_1, S_2, ..., S_n\}$, and a collection of subsets, $I_1, I_2, ..., I_k \subseteq U$, find a sub-collection (set) of I_is, $i \subseteq \{1, 2, ..., k\}$ such that $\bigcup I_i = U$. The sub-collection

Table 3 An example that illustrates how the set cover problem captures the conflicts for the metamodel presented in Fig. 1 including 8 class invariants in Fig. 2a

MaxSMT solutions	$inv1$	$inv2$	$inv3$	$inv4$	$inv5$	$inv6$	$inv7$	$inv8$
$S_1 = \{inv2, inv5\}$	0	**1**	0	0	**1**	0	0	0
$S_2 = \{inv2, inv6\}$	0	**1**	0	0	0	**1**	0	0
$S_3 = \{inv2, inv7\}$	0	**1**	0	0	0	0	**1**	0
$S_4 = \{inv2, inv8\}$	0	**1**	0	0	0	0	0	**1**
$S_5 = \{inv1, inv5\}$	**1**	0	0	0	**1**	0	0	0
$S_6 = \{inv1, inv6\}$	**1**	0	0	0	0	**1**	0	0
$S_7 = \{inv1, inv7\}$	**1**	0	0	0	0	0	**1**	0
$S_8 = \{inv1, inv8\}$	**1**	0	0	0	0	0	0	**1**

For example, a conflict between $inv1$ and $inv2$ in Fig. 2a can be identified here, since $inv1$ covers $\{S_5, S_6, S_7, S_8\}$ and $inv2$ covers $\{S_1, S_2, S_3, S_4\}$

is minimum if it uses the least number of I_is to cover U and such a collection is called *a minimum set*.

To illustrate that the conflicts among the set of metamodel features can be mapped to the set cover problem we use our motivating example from Fig. 1. This example has 8 class invariants as shown in Fig. 2a that we then solve to derive a total of 8 different solutions (S_1, S_2, ..., S_8), as outlined in Table 3. Each individual solution describes a way of maximising the number of class invariants in Fig. 2a, in particular, each of these are MaxSMT solutions. We then construct a matrix with each row describing one solution and each column denoting a class invariant from Fig. 2a. For example, in Table 3, row $S_1 = \{inv2, inv5\}$ denotes a way of achieving 6 of the invariants by deactivating 2 invariants ($inv2$ and $inv5$ in Fig. 2a). In the first row, we use a **1** to denote these two invariants that can *not* be achieved, and **0** to denote the remaining invariants that can be achieved.

There are two conflicts in Table 3. In order to find these two conflicts, consider this table in two dimensions: row and column. We define the union of each row (S_i) as a set to be covered using the column inv_i as a collection of subsets. Each column covers only those rows marked with a 1 in that column. Now, we say that S_i is covered if and only if at least one of the elements is covered. For example, column $inv1$ covers row S_5, S_6, S_7, and S_8, while column $inv3$ covers no rows. A conflict can now be identified by finding a sub-collection (set) of inv_is such that the union of the inv_is covers all rows (S_1 to S_8) in Table 3. Such a set is a minimal unsat core. It is minimal in the sense that the removal of any element from the set results in at least one of the rows becoming uncovered. For example, we can form a set $A = \{inv1, inv2\}$. In fact, this set is a minimal unsat core and thus $inv1$ and $inv2$ (from Fig. 2a) conflict with one another. The other conflict can be identified by forming the second set $B = \{inv5, inv6, inv7, inv8\}$. This is because each element of B uniquely covers two rows, i.e. $inv5$ covers S_1 and S_5 while $inv6$ covers S_2 and S_6.

7.2 Solving the set cover problem

In general, finding one solution to the set cover problem is NP-complete, and finding a minimum set is NP-hard [48]. To tackle this problem, we present a novel technique that allows us to find all metamodel constraint conflicts via SMT solving. Our technique first computes a set of achievable metamodel features (MaxSMT solutions) and populates an $m \times n$ matrix, M, that is similar to the one in Table 3. Then it automatically generates a set of SMT formulas that capture the set cover problem and uses an SMT solver to find metamodel constraint conflicts.

The basis of this technique is to reformulate the set cover problem into a set of numeric constraints so that we can utilise SMT solvers' well-engineered arithmetic reasoning engine to quickly explore the search space. To form such constraints, we first define the $m \times n$ matrix, M, as follows:

$$M = \begin{array}{c} \\ S_1 \\ S_2 \\ S_3 \\ \vdots \\ S_m \end{array} \begin{array}{c} \begin{array}{cccccc} I_1 & I_2 & I_3 & \ldots & I_n \end{array} \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \ldots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \ldots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \ldots & a_{mn} \end{bmatrix} \end{array}$$

M represents the set cover problem. The union of S_is is a finite universe to be covered and each I_i represents every subset that can be used. In fact, this matrix is structured as follows:

- each entry $a_{ij} \in \{0, 1\}$ is an element from a set (S_i or I_j), and 1 denotes that $a_{ij} \in S_i \wedge a_{ij} \in I_j$, otherwise the entry is not in both S_i and I_j.
- each S_i denotes a set of metamodel features that *cannot* be achieved.
- each I_j denotes a subset of the S_is in the jth column, depending on whether $a_{ij} = 1$.

Let the mappings $S_i \mapsto V^{S_i}$, $I_j \mapsto V^{I_j}$ and $a_{ij} \mapsto V^{a_{ij}}$ be SMT encodings of S_i, I_j and each entry a_{ij} of M respectively, where V^{S_i}, V^{I_j} and $V^{a_{ij}}$ are SMT integer variables whose range is $\{0, 1\}$. Next, we generate a set of SMT formulas which captures the set cover problem. The range value 1 denotes that an element or a set is selected (covered) while 0 indicates that it is unselected (not covered).

We first generate Formula 9 which states that S_i is selected (covered) if one of the a_{ij}s in the ith row is selected. Otherwise if all of the a_{ij}s (in the ith row) are not chosen, then S_i cannot be covered. For example, in Table 3, we say that S_1 can be covered by either the entry in the 1st row and 2nd column (a_{12}) or the entry in the 1st row and 5th column (a_{15}), as both of them are set to 1 ($S_1 = \{a_{12}, a_{15}\}$).

$$\bigwedge_{i=1}^{m}\Bigg(\Bigg(\Big(\bigvee_{\substack{j=1\\ a_{ij}\in S_i}}^{n} V^{a_{ij}} = 1\Big) \Rightarrow \Big(V^{S_i} = 1\Big)\Bigg) \wedge \wedge\Bigg(\Big(\bigwedge_{\substack{j=1\\ a_{ij}\in S_i}}^{n} V^{a_{ij}} = 0\Big) \Rightarrow \Big(V^{S_i} = 0\Big)\Bigg)\Bigg) \tag{9}$$

Intuitively, Formula 10 encodes a constraint indicating that if the subset I_j is selected, then all of its elements must be selected as well. Otherwise no elements in I_j can be selected. This formula guarantees that either I_j is chosen or it is not chosen at all. This rules out the possibility of a partial selection of I_j's elements. This is because when a subset is not chosen (used), then none of its elements should be selected. This condition is enforced by the use of a conjunction to connect all elements in I_j to make sure that none of its elements are selected. For example, if the subset I_5 in Table 3 is not chosen, then its two elements at the $5th$ column, marked as 1 (a_{15} and a_{55}) are also not selected ($I_5 = \{a_{15}, a_{55}\}$).

$$\bigwedge_{j=1}^{n}\Bigg(\Bigg(\Big(V^{I_j} = 1\Big) \Rightarrow \Big(\bigwedge_{\substack{i=1\\ a_{ij}\in I_j}}^{m} V^{a_{ij}} = 1\Big)\Bigg) \wedge\Bigg(\Big(V^{I_j} = 0\Big) \Rightarrow \Big(\bigwedge_{\substack{i=1\\ a_{ij}\in I_j}}^{m} V^{a_{ij}} = 0\Big)\Bigg)\Bigg) \tag{10}$$

Finally, we generate an integer equality as shown in Formula 11 that describes the restriction that every S_i must be covered ($part\ a$) by some subsets I_js ($part\ b$). To find all possible combinations of subsets, (I_j), that cover S_is, we use Algorithm 2 (presented overleaf) to iteratively ask an SMT solver to find an answer for $part\ b$, starting from 1 subset to n subsets. If this equality is satisfiable (line 5), we then have a solution to the set cover problem with k subsets covering all S_is. Otherwise, there is no solution to the set cover problem with k subsets. Finally, we interpret those V^{I_j}s assigned with 1 as the chosen subsets (line 6) and find the next solution by blocking all previous solutions (line 7).

$$\Bigg(\underbrace{\Big(\sum_{i=1}^{m} V^{S_i}\Big) = m}_{part\ a}\Bigg) \wedge \Bigg(\underbrace{\Big(\sum_{j=1}^{n} V^{I_j}\Big) = k}_{part\ b}\Bigg) \tag{11}$$

where $1 \leq k \leq n$.

Lemma 2 *The reduction (Formulas 9, 10 and 11) from the set cover problem to SMT is correct.*

Proof We decompose our proof into three components as follows:
Proof of the correctness of Formula 9 by contradiction: Suppose that an element $a_{ij} \in S_i$ from matrix M is selected, but the corresponding S_i is *not* covered. This is, in fact, impossible because the disjunction in Formula 9 guarantees that if at least one of the $a_{ij} \in S_i$ is selected, then S_i must be covered ($V^{S_i} = 1$). Suppose that $a_{ij} \in S_i$ is *not* selected and the corresponding S_i is covered. This is also not possible because the second conjunction in Formula 9 guarantees that when there are no a_{ij}s selected, then S_i is not covered either ($V^{S_i} = 0$). Thus, Formula 9 captures the constraints for the set ($S_i s$) to be covered in M.
Proof of the correctness of Formula 10 by contradiction: Let the subset I_j be selected and there exists some $a_{ij} \in I_j$s that are *not* selected. However, the second conjunction in Formula 10 prevents this from happening. As long as I_j is selected, then every single element ($a_{ij} \in I_j$) in I_j must also be selected. Thus, this is not possible. Similarly, let us assume that V_j is *not* selected but some $a_{ij} \in V_j$ are selected. Again, this is not possible since the third conjunction in Formula 11 rules out this scenario. Thus, Formula 10 captures the constraints for the subsets to be used (I_i) in M.
Proof of the correctness of Formula 11: $part\ a$ in Formula 11 enforces that all S_is must be covered and by Formula 9 we know that at least one a_{ij} must be selected in the corresponding ith row. Furthermore, $part\ b$ in Formula 11 enforces that we must use k subsets of I_js and by Formula 10 we know that once an I_j is selected, then all of its elements are also selected and this leads to some S_is being covered.
Therefore, by the correctness of Formula 9, 10 and 11, this is a correct reduction from the set cover problem to SMT. □

We now construct Algorithm 2 which iteratively calls an SMT solver to return all solutions to the set cover problem.

Theorem 2 *Algorithm 2 finds all solutions to the set cover problem.*

Proof By Lemma 2, we know that Formula 9, Formula 10 and Formula 11 correctly capture the set cover problem. Each iteration (outer loop on lines 4–9) in Algorithm 2 issues a call

Algorithm 2: This algorithm iteratively calls an SMT solver and returns all solutions to the set cover problem. The first set of solutions found by this algorithm must be the set containing all minimum sets since k starts from 1. This algorithm works by iteratively using an SMT solver as an oracle for solutions that uses k subsets.

Input : A matrix M representing metamodel constraint conflicts as the set cover problem.
Output: A set s containing all solutions to the set cover problem including all minimum sets.

1 $k \leftarrow 1$
2 $s \leftarrow \emptyset$
3 $Solver.add(Formula\ 9 \wedge Formula\ 10 \wedge Formula\ 11_{part\ a})$
4 **while** $k \leq n$ **do**
5 | **while** $SMTSolve\Big(\big(\sum_{j=1}^{n} V^{I_j}\big) = k\Big) =$ **SAT do**
6 | | $s \leftarrow s \cup Interpret(V^{I_j})$
7 | | $Solver.add($**BlockingFormula**$)$
8 | **end**
9 | $k \leftarrow k + 1$
10 **end**
11 **return** s

to an SMT solver to test the satisfiability of ϕ with k subsets. If ϕ is satisfiable, then there must exist a way that uses k subsets to cover the set. The algorithm then enumerates (inner loop on lines 5–8) all other possible ways (using k subsets). Since the value of k starts from 1 to n, s (line 11) must contain all solutions (using k subsets) to the set cover problem. □

In this section, we have presented our reduction and proofs of the correctness of our algorithm for solving the set cover problem to derive the set of minimum conflicts.

8 Tool implementation: MaxUSE

In this section, we introduce our automated tool called MaxUSE by describing its architecture and core components. Further to this, we show how MaxUSE solves an example by revealing its generated SMT2 formulas along with relevant screen-shots of the final output.

8.1 Implementation

We have built a prototype tool called MaxUSE that implements our approach.

Available at https://github.com/classicwuhao/maxuse

MaxUSE is a fully automated tool that builds on top of the existing USE modelling tool [37]. We chose USE mainly because it is a widely used modelling tool that has its own specification language that we can alter for our requirements. We modified its grammar and abstract syntax trees so that it now takes a metamodel that is either fully or partially ranked as input. MaxUSE works by traversing a metamodel and automatically generates a set of SMT2 formulas [7]. Currently, MaxUSE uses Z3 as its solving engine [27]. It incrementally solves generated formulas and interprets each successful assignment as a solution. The implementation of MaxUSE consists of approximately 10,000 lines of Java code, with approximately 3000 of these lines dedicated to our core algorithm.

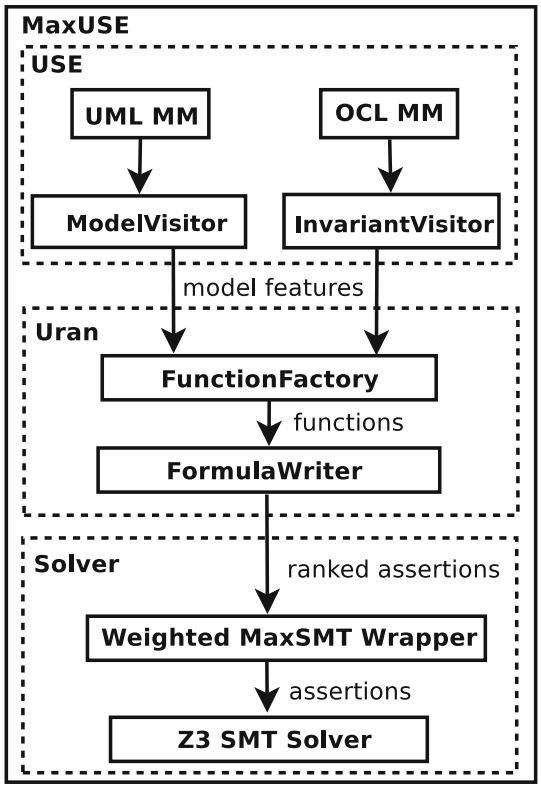


Fig. 10 The architecture of MaxUSE integrates three layers: USE, Uran and Solver

8.1.1 Overall architecture

MaxUSE is composed of three layers: USE, Uran and Solver [80]. MaxUSE exploits USE's front-end (OCL engine) to read in a UML class diagram that is annotated with OCL constraints and automatically generates SMT assertions that can be solved by an SMT solver. The overall architecture of MaxUSE is illustrated in Fig. 10. We discuss each of these three layers in detail as follows:
USE is an open-source modeling tool that allows users to construct UML class diagrams in its own specification language [37]. It also supports constraints that are written in OCL. USE provides a set of commands that enable users to construct object diagrams (instances) and to check whether an object diagram (instance) conforms to its class diagram's structural and OCL constraints. To support ranked constraints,

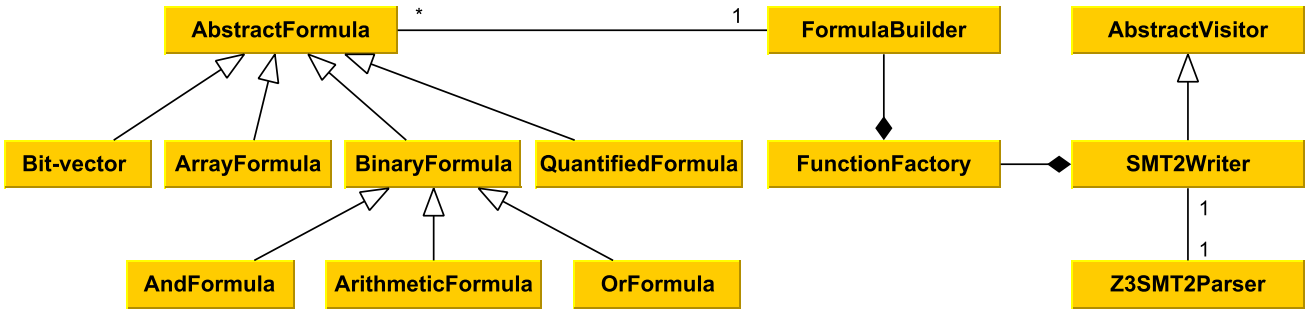


Fig. 11 The core classes of our intermediate layer Uran. Uran uses a factory design pattern ($FunctionFactory$) to build different kinds of formulas and uses a visitor pattern ($SMT2Writer$) to output standard SMT2 formulas

we amend USE's front-end by modifying its grammars, UML and OCL metamodels (abstract syntax trees). We then implement two visitors that traverse and store each model feature. In particular, classes, associations and class invariants are stored into a temporal memory location that can be used by our intermediate layer, Uran.

Uran is an open-source project that aims to provide users with an engine for constructing and evaluating standard (ranked or unranked) SMT2 assertions through well-defined APIs.

Available at: https://github.com/classicwuhao/uran.

Uran provides an intermediate layer between a specification and a constraint solver so that formula generation functionalities are decoupled from the modules that are designed for other purposes. More importantly, Uran outputs all assertions into standard SMT2 format so that they can be easily debugged.[6] This design allows users to freely modify and upgrade formula generation for specific purposes without affecting other modules. Currently, Uran communicates with the Z3 SMT solver.

The core part of Uran is illustrated in Fig. 11 and central to it are the $FunctionFactory$ and $SMT2Writer$ classes. To construct a formula or a constraint, users must instantiate the $AbstractFormula$ class via its children classes. Currently, Uran supports the creation of a variety of formulas including Boolean, integer arithmetic, array, and bit-vector through the factory design pattern. Uran uses a visitor pattern to traverse the formulas that are created by $FunctionFactory$ and writes them to an SMT2 file that is later parsed using the Z3 APIs. Once Z3 determines the satisfiability of the formulas, Uran maps each successful assignment into a solution back to the problem domain. It then generates blocking formulas in order to find the next solution until the formulas are unsatisfiable.

Solver we have implemented Algorithms 1 and 2 into a wrapper. This wrapper iteratively calls the Z3 SMT solver and performs constraint solving until no more solutions are found. Currently, this wrapper only provides APIs to work with the Z3 SMT solver and standard SAT solvers such as SAT4J and minisat [10,28]. In the future, we plan to extend this wrapper for other SMT solvers such as CVC4 and MATHSAT [6,23].

The above, three-layer architecture is modular thus making it easily extensible and we illustrate its use via a detailed example in the next subsection.

8.2 A Detailed Example

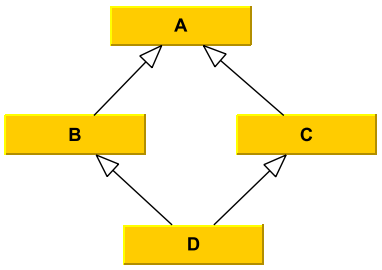
In this section, we use a small but detailed example from [36] to illustrate how the techniques that we have described in the previous sections are implemented in MaxUSE. Figure 12a illustrates this example that models multiple inheritance relationships with one class invariant. This metamodel is inconsistent. In fact, it is impossible to instantiate class D because of the conflict between the diamond shaped inheritance relationship and the $disjointBC$ constraint. To demonstrate our MaxUSE tool, we specify this metamodel in the USE specification language. Figure 12b contains the USE specification corresponding to this example. However, the current version of MaxUSE does not support the syntax of variable declaration of the form:

$$context\ v : V$$

Thus, we rewrite the $disjointBC$ into a semantically equivalent class invariant in Fig. 12b. We then rank each class with an integer value of 2 and the invariant with a value of 8. Hence, the total rank for this metamodel is now 16. Since there is a conflict between the diamond inheritance relationship and the class invariant, MaxUSE finds a maximum rank of 14 achievable constraints. This is accomplished by deactivating class D.

The detailed SMT encodings for this example are shown in Fig. 13 and we summarise them as follows

[6] The output from current Z3 Java APIs includes other information or may be reduced into one long assertion which is not ideal for debugging purposes.




```
context b: B inv disjointBC:
C.allInstances()->forAll(c | b<>c)
```

(a) A metamodel that has multiple inheritance relationships along with one OCL class invariant.

```
model DisjointSubclasses
@Rank = 2
class A end

@Rank = 2
class B < A end

@Rank = 2
class C < A end

@Rank = 2
class D < B, C end

constraints
context B
@Rank = 8
inv: B.allInstances()->forAll(b|C.allInstances()->forAll(c|b<>c))
```

(b) The corresponding ranked USE specification for the metamodel in Figure 13(a).

Fig. 12 A metamodel depicts a diamond inheritance relationship among four different classes. The USE specification for this inheritance relationship is also shown

$$
\begin{aligned}
1 \quad & \Big(\forall b : INT \cdot T_B(O_{id}(b)) \wedge T_A(O_{id}(b)) \Rightarrow \\
2 \quad & \Big(\forall c : INT \cdot T_C(O_{id}(c)) \wedge T_A(O_{id}(c)) \Rightarrow \\
3 \quad & O_{id}(b) \neq O_{id}(c)\Big)\Big) \vee (Aux_1 = 1) \wedge \\
4 \quad & \big(\exists p : INT \cdot T_A(O_{id}(p))\big) \vee \big(Aux_2 = 1\big) \wedge \\
5 \quad & \big(\exists p : INT \cdot T_B(O_{id}(p))\big) \vee \big(Aux_3 = 1\big) \wedge \\
6 \quad & \big(\exists p : INT \cdot T_C(O_{id}(p))\big) \vee \big(Aux_4 = 1\big) \wedge \\
7 \quad & \big(\exists p : INT \cdot T_D(O_{id}(p))\big) \vee \big(Aux_5 = 1\big) \wedge \\
8 \quad & (Aux_1 = 1) \Rightarrow (W_1 = 0) \wedge \\
9 \quad & (Aux_1 = 0) \Rightarrow (W_1 = 8) \wedge \\
10 \quad & (Aux_2 = 1) \Rightarrow (W_2 = 0) \wedge \\
11 \quad & (Aux_2 = 0) \Rightarrow (W_2 = 2) \wedge \\
12 \quad & (Aux_3 = 1) \Rightarrow (W_3 = 0) \wedge \\
13 \quad & (Aux_3 = 0) \Rightarrow (W_3 = 2) \wedge \\
14 \quad & (Aux_4 = 1) \Rightarrow (W_4 = 0) \wedge \\
15 \quad & (Aux_4 = 0) \Rightarrow (W_4 = 2) \wedge \\
16 \quad & (Aux_5 = 1) \Rightarrow (W_5 = 0) \wedge \\
17 \quad & Aux_5 = 0) \Rightarrow (W_5 = 2) \wedge \\
18 \quad & \sum_{i=1}^{5} Aux_i = 0 \wedge \\
19 \quad & \sum_{i=1}^{5} W_i = C
\end{aligned}
$$

Fig. 13 The detailed SMT encodings for the example shown in Fig. 12. Lines 1–7 show the encodings of the four classes and one class invariant in Fig. 12a. Lines 8–17 show the encodings of corresponding ranks. Line 18 encodes the condition for checking consistency and line 19 encodes the condition for maximising defined ranks

Lines 1–3: show the formula that encodes the class invariant (this corresponds to Formula 4 in Sect. 5). The formula also captures the inheritance structural constraints referred to by the class invariant. For example, an instance (b) of a class B is also a type of a class A since B inherits from A. This is captured by $T_B(O_{id}(b)) \wedge T_A(O_{id}(b))$.

Lines 4–7: contain the additional axioms stating that there must exist at least one object for each of the four classes (A, B, C and D) (this corresponds to Formula 8 in Sect. 6.2).

Lines 8–17: Since this metamodel is ranked, an integer type auxiliary (Aux_1 - Aux_5) variable is introduced for the formulas that encode each different metamodel feature. Note that we constrain each auxiliary variable with either 0 or 1. Hence, these formulas have two consequences:

1. If an Aux_i is successfully assigned to a 0, then a corresponding rank is applied.
2. Otherwise no ranks should be applied ($W_i = 0$).

Line 18: To check the consistency of this metamodel, we construct the formula on line 18 to constrain

Weighted Solution 1:

Name	Type	Status	Weight
inv1	INVAR	ON	8
A	CLASS	ON	2
B	CLASS	ON	2
C	CLASS	ON	2
D	CLASS	OFF	NA
			Total: 14

Fig. 14 This report shows that the solution found by MaxUSE for ranked USE specification in Fig. 12b

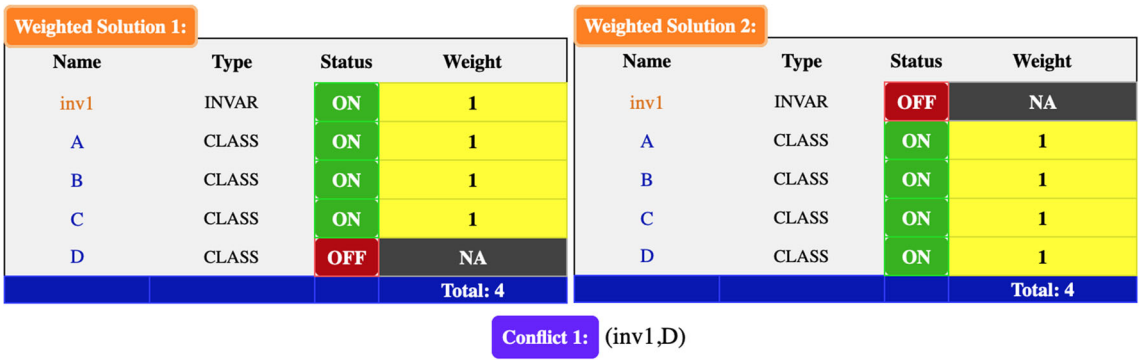


Fig. 15 The report shows that the two solutions with one conflict found by MaxUSE for an equally ranked metamodel in Fig. 12a. OFF in two screen-shots here indicate that a particular feature is deactivated, and ON means a feature is activated

the summation of all auxiliary variables to 0. This forces the SMT solver to try and satisfy the SMT encodings of each class and invariant. If the SMT solver could not find an assignment for the formula on line 18, then this metamodel is *not* consistent.

Line 19: If the metamodel is inconsistent, we add this additional formula and use Algorithm 1 to maximise the constant C. This allows us to compute the set of achievable features based on their individual ranks.

Figure 14 contains a screenshot of the report that is generated by MaxUSE showing a maximum total rank of 14.

In order to pinpoint the conflict, we then rank each feature in this example equally[7] and run MaxUSE again. This time MaxUSE returns two solutions indicating that there are two ways of achieving the maximum number of 4 achievable features. Figure 15 presents a screenshot of a MaxUSE generated report containing these two solutions.

9 Evaluation

In the previous section, we described the internal design of our tool and how it can be used to find conflicting OCL constraints via a detailed example. In this section, we present systematic evaluation results of MaxUSE and discuss its advantages and our findings. Furthermore, we contribute a benchmark that we have used to evaluate our tool and we believe that it can be used for evaluating the usability, scalability and performance of other OCL-based verification tools.

[7] In this case, we rank each class and the invariant with 1.

9.1 Generating a benchmark

In order to extensively evaluate the capabilities of our tool, MaxUSE, we initially collected a group of metamodels from [36]. However, each of these metamodels was quite small and contained very few class invariants. Hence, using these metamodels was insufficient for us to determine the capabilities of MaxUSE. In fact, the majority of the literature in this area uses very few examples for evaluation and some of them only use a single, bespoke example [51,63,69,76]. Therefore, one of the contributions of this paper is our formation of a robust benchmark that can be used for evaluating OCL-based verification tools.

To form a benchmark that is suitable for fully and systematically evaluating our approach, we use the metamodels from [36] as candidate metamodels. Based on these candidate metamodels, we calculate a configuration in terms of the number of associations (different multiplicities), quantifiers, logic/arithmetic operators, invariants, quantifiers, and breadth/depth of inheritance trees. We then follow the work presented in [81] to develop a generator for USE specifications based on different sized configurations. This generator currently employs a tree generation algorithm that generates a distinct shape of AST based on the given size. We use this generator to generate an additional four groups (Group B, C, D and E in Table 4) of metamodels using the configurations calculated from the candidate metamodels. For each metamodel listed in each group, we generate two sets of rankings: one for mixed rankings so that we can evaluate finding optimal values and one for equal rankings so that we can evaluate finding conflicts. For mixed rankings, we randomly generate a set of different rankings (including the automatic rankings as outlined in Sect. 4.3) for each individual feature including classes, associations and class invariants.

The full benchmark is listed in Table 4 where each group (B,C,D and E) contains small, medium, large and extreme numbers of class invariants. The detailed overall structure of the defined invariants are calculated in terms

Table 4 The benchmark for evaluating MaxUSE

	Name	Metamodel structure			Invariant structure		
		Classes	Assocs	Invs	Nodes	Quantifiers	Operators
Group A	CS	3	1	6	30	3	9
	WR	2	2	7	52	8	1
	DS	4	0	1	7	2	1
	SM	5	3	8	73	7	18
Group B	B1	13	5	27	150	10	30
	B2	24	9	45	266	13	57
	B3	33	14	68	430	9	111
	B4	46	15	90	599	23	152
	B5	57	19	136	925	44	228
Group C	C1	13	5	29	201	24	33
	C2	24	11	43	279	28	51
	C3	35	17	66	413	42	82
	C4	46	15	98	698	69	137
	C5	57	15	156	1008	100	184
Group D	D1	13	2	22	174	23	36
	D2	26	9	47	286	29	68
	D3	33	3	61	324	23	72
	D4	46	9	101	753	102	163
	D5	56	18	166	1143	131	225
Group E	E1	10	6	31	294	12	86
	E2	15	12	39	452	18	135
	E3	30	18	37	403	31	102
	E4	18	18	105	985	56	246
	E5	18	18	167	1134	68	325

The columns of metamodel structure list the overall size of each metamodel. The columns of invariant structure list the overall size of class invariants of each metamodel

of the AST size (*Node* in Table 4), number of quantifiers and logic/arithmetic/collection operators. The benchmark in Table 4 covers a variety of class invariants and imposes a great challenge for current techniques in terms of automation, verification and constraint solving. Hence, we believe that our generated benchmark can be used not only for measuring the strengths of existing OCL-based verification techniques but also their limitations. Currently, our prototype tool, MaxUSE, fully supports the OCL constraints that are used in this benchmark, except for the OAI metamodel (in Group A) due to its recursive structures. Thus, we replace this OAI metamodel with our running example that was presented in Sect. 2.

9.2 Performance evaluation

We have evaluated MaxUSE on an Intel(R) Xeon(R) machine that has eight 3.2GHz cores with 16G of memory. Nevertheless, our current implementation uses only one core. MaxUSE's underlying solver is the Z3 SMT solver (version 4.8.3).[8] Though we have performed an evaluation in [79], we have not covered MaxUSE's capabilities for solving ranked constraints and enumerating all possible solutions. In our new evaluation, we evaluate the performance of MaxUSE in two different experiments. The first experiment focuses on measuring MaxUSE's performance in two scenarios: equally ranked and mix ranked. In the first scenario, we rank each metamodel feature equally. In the second scenario, we use a mixture of rankings including randomly generated ranks (soft and hard features) and the automatic ranking that was described in Sect. 4. Our second experiment measures the effectiveness and efficiency of MaxUSE when it is used to find conflicts among class invariants in each of the metamodels in Table 4. We discuss our findings from these two experiments in what follows.

In order to carry out these experiments, MaxUSE first determines whether a metamodel is consistent or not. MaxUSE can generate SMT2 formulas and determine the consistency of each metamodel in under *1 s*. Figure 16a, b show the average time that MaxUSE spent on generating

[8] https://github.com/Z3Prover/z3.

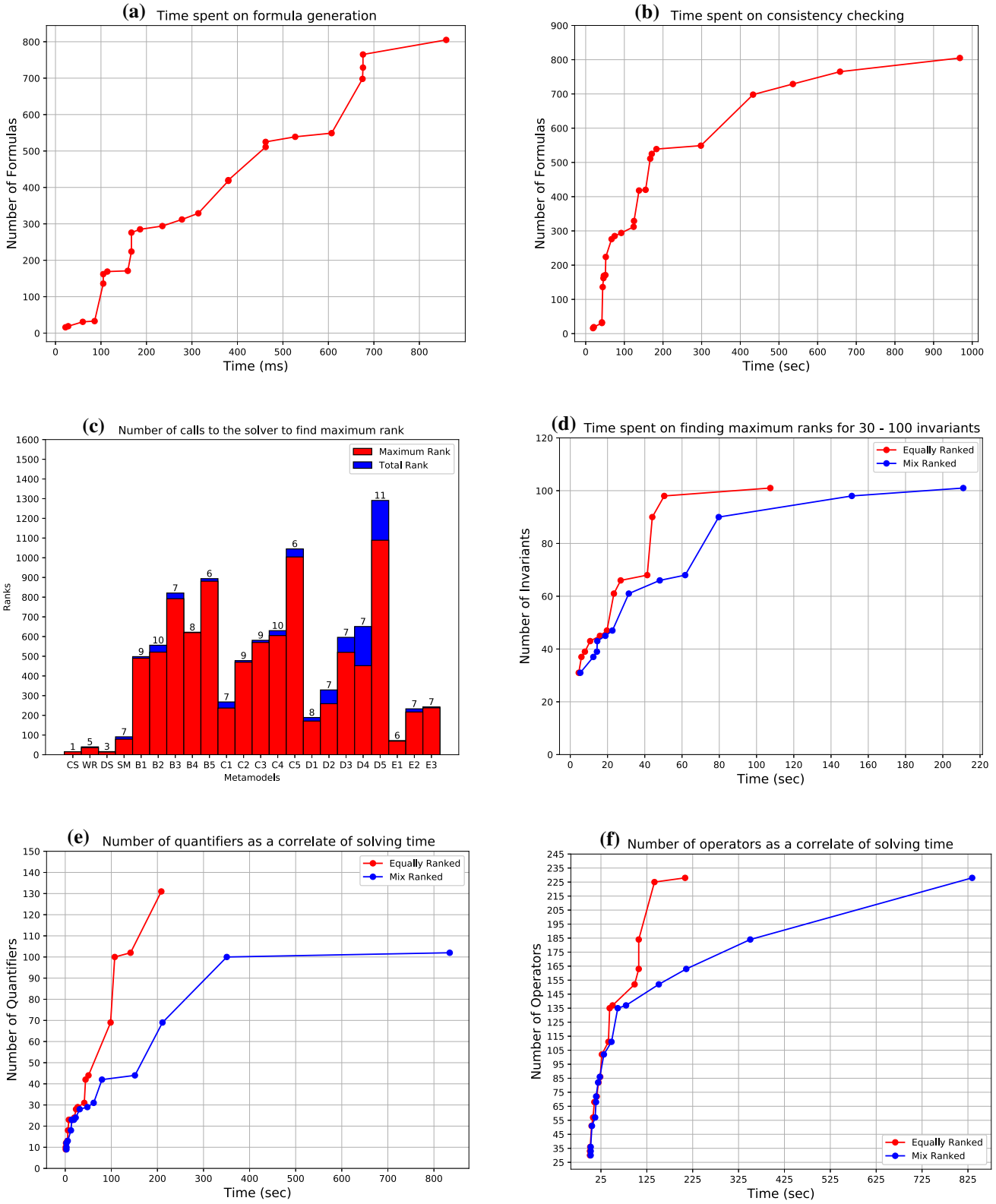


Fig. 16 Performance evaluation: **a** formula generation, **b** consistency checking, **c** number of calls to find maximum rank, **d** computing achievable features for 30–100 invariants, **e** number of quantifiers affect solving time, and **f** number of operators affect solving time

Table 5 The results of the first two experiments

	Name	Formulas	Consis (s)	Rank		Optimal value (s)		All solutions (#/s)	
				Max	Total	Eq ranked	Mix ranked	Eq ranked	Mix ranked
Group A	CS	19	0.019	15	15	NA	NA	NA	NA
	WR	33	0.044	36	40	0.450	4.02	2/0.132	2/12.57
	DS	16	0.047	14	16	0.450	0.74	2/0.134	1/0.129
	SM	44	0.021	79	91	0.183	0.292	8/0.670	1/0.36
Group B	B1	169	0.042	490	498	1.092	2.609	2/1.787	1/3.249
	B2	285	0.067	521	556	7.845	14.485	12/19.110	4/18.971
	B3 *	420	0.125	792	821	23.412	31.42	6/44.945	6/53.89
	B4	539	0.433	620	622	44.040	151.165	2/68.438	1/175.073
	B5	729	0.167	881	894	141.715	834.210	24/267.369	8/5978.879
Group C	C1	171	0.171	237	268	1.092	2.609	2/1.787	1/3.249
	C2	276	0.051	470	478	5.993	12.383	4/10.498	1/13.743
	C3	418	0.298	570	581	50.448	61.686	1/57.034	1/68.912
	C4 *	549	0.968	605	630	107.395	211.004	4/176.403	1/250.790
	C5	765	0.155	1004	1045	208.489	350.517	11/749.589	91/8246.351
Group D	D1	136	0.033	171	189	1.730	2.544	1/2.264	1/3.062
	D2	294	0.075	259	329	19.683	18.835	3/32.816	1/23.924
	D3	329	0.092	520	596	10.683	22.617	6/21.835	2/28.274
	D4	525	0.124	452	651	41.315	79.698	3/85.313	2/2733.066
	D5	805	0.183	NA	1291	98.376	TO	91/ 8246.351	TO
Group E	E1*	162	0.042	69	72	4.538	5.375	1/5.723	1/6.559
	E2*	224	0.052	217	233	15.920	48.034	2/25.027	1/55.289
	E3	312	0.138	238	243	27.057	14.303	1/35.153	1/23.652
	E4	511	0.536	NA	515	TO	TO	TO	TO
	E5	698	0.658	NA	415	TO	TO	TO	TO

Here, "Formulas" denotes the number of generated formulas, "Consis" denotes the time spent by MaxUSE on determining whether a metamodel is consistent or not. "Rank" denotes the achieved maximum rank ("Max") out of a total rank distributed ("Total"). "Optimal Values" measures the time MaxUSE took in order to find an optimal rank for both equally ranked and mix ranked scenarios. "All Solutions" denotes the time (in s) spent by MaxUSE on finding all possible solutions. Note, * denotes the metamodels that require several runs of MaxUSE in order to find solutions

SMT formulas and checking consistency. The performance here shows that MaxUSE is *very* efficient when checking metamodel's consistency. In other words, checking the consistency of a metamodel for MaxUSE is an easy task [50,51,72]. Compared to the research on consistency checking, MaxUSE takes one step further to finding achievable features (based on their ranks) and pinpointing conflicts. This is done by solving two much more challenging problems: weighted MaxSMT and the set cover problem.

9.2.1 Experiment 1: equally ranked and mix ranked

In this experiment, both scenarios (equally ranked and mix ranked) require MaxUSE to first find an optimal rank and then enumerate all other possible solutions. Hence, we measure the performance of finding such an optimal rank. The results are presented in the columns labelled "Optimal Value" in Table 5. In general, finding an optimal rank for the mix ranked scenario is more challenging than for the equally ranked scenario. This is because the mix ranked scenario could easily have a much larger total rank. In most cases, MaxUSE is able to find an optimal rank within a reasonable amount of time. However, in some of the extremely challenging cases, MaxUSE may not be able to find a solution. For example, MaxUSE could not progress, in one hour for, metamodels $E4$ and $E5$. This is because the underlying solver (Z3) is not capable of efficiently solving formed linear equalities using our Algorithm 1. Overall, the problem (weighted MaxSMT) that we are trying to solve here is extremely challenging for SMT solvers due to the nature of its computational complexity.

To precisely evaluate MaxUSE's capabilities for finding maximum ranks (optimal value) for mix ranked metamodels, we record the number of calls to the solver that MaxUSE makes. Figure 16c shows these numbers for each metamodel listed in Table 4. In the best case scenario, MaxUSE is able to find maximum ranks between one and three SMT calls such as CS and DS metamodels. However, most of the metamodels

Table 6 The detailed rankings for each class in the B1 metamodel from our benchmark listed in Table 4

Feature	Rank
Class 0	37
Class 1	3
Class 2	4
Class 3	1
Class 4	2
Class 5	5
Class 6	3
Class 7	2
Class 8	4
Class 9	6
Class 10	2
Class 11	1
Class 12	26

in our benchmark require quite a few SMT calls in order to find the maximum achievable ranks. As shown in Fig. 16c, each maximum rank found is very close to the total rank. This suggests that MaxUSE explores the worst case scenario for most of the metamodels to hit the optimal value.

As outlined above, our first experiment focuses on two scenarios: a metamodel that is equally ranked and a metamodel that is mix ranked. The columns labelled "All Solutions" show the performance for finding all possible solutions for both scenarios (Table 5). In general, solving equally ranked metamodels is slightly faster than mix ranked metamodels. This is because the ranks distributed to each metamodel feature are relatively smaller. On the other hand, mixed rank pose a great challenge to the underlying SMT solver. In most cases, MaxUSE can enumerate all possible solutions for mixed rank metamodels within a reasonable amount of time. For example, Table 6 illustrates our randomly generated rankings for 13 classes from the B1 metamodel in our benchmark. Detailed rankings for each of the metamodels in our banchmark are available in the MaxUSE git repository as `.rank` files.

In most cases where metamodels contain a medium to large (30–100) number of invariants, MaxUSE is able to find the maximum total rank for both equally and mix ranked scenarios within a reasonable amount of time (at most 4 min). This is shown in Fig. 16d. As can be seen, MaxUSE typically takes longer to find maximum ranks for mix ranked metamodels than for equally ranked ones. This further confirms our observation that mix ranked features impose greater challenges to the SMT solver than equally ranked ones. In extreme cases (over 100 invariants), the longest time taken by MaxUSE is approximately 2.5 h to get 91 solutions for the C5 metamodel in Table 5. This is mainly due to the Z3 SMT solver spending a significant amount of time solving a large number of formulas that contain deeply nested quantifiers and inequalities.

Both the number of quantifiers and the number of operators are proportional to the solving time. This relationship is shown in Fig. 16e, f. The more quantifiers a constraint contains, the more challenging it is for MaxUSE to solve this constraint. When a metamodel contains a large number of invariants with large rankings, MaxUSE typically spends a significant amount of time on finding optimal values. Similarly, the number of operators (including: logical, arithmetic, collection operators) could also have an impact on solving time. In particular, operators over collection data types such as include/exclude. This is because these operators are translated into quantifiers over collections.

9.2.2 Experiment 2: finding conflicts

When compared to solving equally ranked or mix ranked metamodel features, finding conflicting constraints is much more efficient. In general, MaxUSE finds all constraint conflicts in a very efficient manner. This is because once MaxUSE solves the (equally) weighted MaxSMT problem, it can utilise the MaxSMT solutions to solve the set cover problem much faster by forming a boolean matrix. In two cases, MaxUSE could not find solutions. For the E5 metamodel, MaxUSE was stuck with a particular value and could not progress to the next possible optimal value within 9 h. We extracted the formulas from MaxUSE's engine, manually checked them with Z3 and found that Z3 timed-out when deciding the satisfiability of the formulas for a particular value. In general, this is an extremely challenging task for any algorithm to find an optimal value for such a large number of complicated formulas. This is because the nature of this particular optimisation problem typically has a massive search space.

For the conflicts found in these metamodels using the benchmarks shown in Table 4, we compare them against actual injected conflicts in order to determine their accuracy. The injected conflicts cover a wide range of different metamodel features including multiplicities on association ends, different types of attributes and inheritance relationships among multiple classes. We classify our comparison results as either "accurate" (denoted as a), "near" (denoted as n) or "miss" (denoted as m) and present them in Table 7. Here, "accurate" means that MaxUSE finds conflicts that match with exact injected conflicts. In other words, each one (set) is minimal and the removal of any members can make the metamodel become consistent. Then "near" means that MaxUSE is able to identify all of the conflicts that are close enough to the injected ones. We label these as "near" because each reported conflict is a slightly larger set containing those injected ones as a subset. For example, MaxUSE may list the a class containing conflicted invariants as a part of the

Table 7 Our evaluation results on finding all constraint conflicts for each metamodel in Table 4

Group A			Group B			Group C			Group D			Group E		
Name	Conflicts		Name	Conflicts		Name	Conflicts		Name	Conflicts		Name	Conflicts	
	$\mathcal{A}$	#/s		$\mathcal{A}$	#/s		$\mathcal{A}$	#/s		$\mathcal{A}$	#/s		$\mathcal{A}$	#/s
CS	NA	0	B1	*a*	1/0.45	C1	*a*	11/0.68	D1	*a*	6/0.43	E1	*n*	1/0.44
WR	*a*	1/0.91	B2	*n*	1/0.21	C2	*a*	2/0.87	D2	*n*	13/1.06	E2	*n*	1/0.66
DS	*a*	1/0.06	B3	*a*	4/1.42	C3	*n*	2/1.35	D3	*n*	2/1.05	E3	*a*	1/0.93
OAI	NA	NA	B4	*a*	1/1.76	C4	*a*	2/1.96	D4	*n*	1/1.40	E4	TO	TO
SM	*a*	2/0.13	B5	*a*	6/3.83	C5	*n*	11/3.22	D5	*n*	114/165.81	E5	TO	TO

$\mathcal{A}$ here denotes the quality of computed conflicts. We use *a*, *n* and *m* to denote accurate, near and miss, respectively. TO here means timed out

returned conflicts. Thus, users could easily understand this information and use it at a later stage for debugging or fixing conflicts. Finally, "miss" indicates that MaxUSE returns at least one "conflict" that is not related to any of those injected conflicts. We suspect that this is probably caused by the heuristic algorithms that are used internally by Z3. Despite this inaccuracy, we believe that the results here show the potential of our approach to finding constraint conflicts for inconsistent metamodels.

10 Discussion

In this section, we discuss the advantages of MaxUSE in terms of its usability, scalability and performance. We also discuss some other interesting findings that we uncovered during our evaluation phase.

10.1 Usability, scalability and performance

We discuss the advantages of MaxUSE under three specific headings as follows.

Usability In general, MaxUSE is easy to use and does not require manual interactions since it is fully automatic. It produces HTML-based reports that contain the number of achievable features (based on the ranks) and the conflicting constraints without any human interaction. Since MaxUSE is built on top of the USE modelling tool, users who are familiar with USE would find the annotations that we have described in Sect. 4 easy to use. For users who have not used USE before, we provide detailed descriptions, including build instructions, on the MaxUSE website.[9] Though the automation here has been achieved, in some cases an interactive mode is necessary. This is because the underlying Z3 SMT solver might not be able to construct a successful assignment for complex formulas. For example, when Z3 could not solve formulas generated for the $E5$ metamodel within a specified time frame, we paused MaxUSE and manually chose a possible optimal value. MaxUSE was then able to resume the search. However, selecting such a value is tricky and requires that one has knowledge about the internal workings of the Z3 solver.

Scalability MaxUSE performs well on a large number of class invariants. From our evaluation results, it can be seen that MaxUSE is able to handle over 100 different types of class invariants ranging from nested quantified expressions to a combination of heavily used arithmetic and logic operators. In general, the solving time is proportional to the number of ranked features. The more ranked features found in a metamodel, the more time MaxUSE needs to find a solution. Though it is the user's choice to use different integers for ranking individual metamodel features, we suggest that one could gain better performance by normalising the ranks. Using relatively smaller ranks can ease the computational burden of an SMT solver.

Performance The reductions to the weighted MaxSMT problem and the set cover problem pose a great challenge for MaxUSE. This is because both problems are *not* easy in terms of computational complexity. However, the overall performance of computing achievable features and conflicting constraints mainly depends on the kinds of formulas that are generated by MaxUSE. For example, expressions with quantifier alternations are typically challenging for the underlying solver, while propositional formulas are easy to decide. We found that, overall, MaxUSE is efficient and effective in finding achievable features and conflicting constraints. However, finding one solution is quite different from enumerating all possible solutions. In general, computing all solutions such as conflicting constraints can be significantly more expensive than finding one solution since there could be an exponential number of them. In this case, it might be necessary to let the user decide when to stop MaxUSE during the enumeration of all constraint conflicts. This is because some constraint conflicts are not independent. Therefore, these dependent conflicts can be used to identify other conflicts without exhaustive enumeration. In the future, we plan to address this issue and enhance our algorithms to reduce the

[9] https://github.com/classicwuhao/maxuse.

number of calls to the solver for finding all possible constraint conflicts.

10.2 Other findings

During our evaluation phase, we encountered a number of other findings and we discuss these below.

1. The Z3 SMT solver is particularly sensitive to alternative quantifiers. This means that for an OCL constraint of the following form, where Q is a predicate or function over x and y, it is highly likely that the solver will return $unknown$.

   $$\forall x \exists y\, Q(x, y)$$

   The reason for this is that alternative quantifiers in general impose a great computational challenge for all SAT/SMT solvers. Though dedicated decision procedures can be integrated directly into the solver, existing SMT-based techniques explore large search space [66,67].
2. The first run of MaxUSE does not always return solutions.[10] In order to get solutions, multiple runs of MaxUSE are required. For example, MaxUSE is not able to find solutions for metamodels $B3$, $C4$, $E1$ and $E2$ in the first run. However, the second or third runs typically return solutions. We surmise that this is mainly due to the incomplete heuristic algorithms used in SMT solvers [65]. These algorithms may use random seeds for their fitting functions to choose values that are approximately to the best one. In the future, we plan to overcome this by integrating multiple SMT solvers (in a similar fashion to Why3 [33]) and allow users to switch among them for the best performance and accuracy.
3. MaxUSE was able to solve the $E4$ metamodel from [79] with version 4.4.0 of Z3. However, with the latest version of Z3, MaxUSE cannot solve $E4$, even for equally ranked features. In fact, MaxUSE cannot find a more optimised maximum rank than 128, and returns unknown. We surmise that the reason for this is the new linear integer arithmetic solver that has been integrated into the Z3 SMT solver since version 4.8.0. This new solver may have a direct impact on solving linear integer inequalities. Furthermore, we also notice that the latest version of the solver is slightly more accurate than the old version. This is reflected in multiple metamodels from our benchmark. For example, for metamodels $C1$ and $C2$ MaxUSE is able to find exact sets of conflicting features compare to our evaluation in [79].

[10] The time taken here varies and we only consider successful runs in our evaluation.

11 Limitations

In this section, we discuss MaxUSE's limitations.

11.1 Ranking

The automatic ranking calculation that we describe in Sect. 4.3 provides a simple way of calculating rankings for classes, associations and invariants. However, this current automatic ranking calculation may not precisely capture the importance of each individual feature. For example, a class with 2 attributes and 1 operation is weighted less than an invariant that has 12 AST nodes. During the search, the invariant with a higher rank is more likely to be chosen (by the solver) than its class if there is a conflict between the class and the invariant. This is counter-intuitive and one may argue that this invariant is not 4 times more important than the class. Thus, we conclude that our automatic ranking does not always precisely reflect which features are most important.

In fact, it is designed as a quick and simple ranking scheme for those who do not have domain-specific knowledge. In this way, we provide a simple ranking scheme that can be used as a base for more precise rankings at a later stage. To avoid an imprecise ranking, we recommend that users use the automatic ranking first and then refine the model with manual rankings as the design progresses to appropriately reflect the relative importance of the features. For example, a user may not have domain-specific knowledge at the beginning of the design. They may choose to use the automatic rankings as a default scheme. When they gain more knowledge about the system, they can then use manual rankings to override the previous automatic rankings.

Furthermore, our automatic ranking is designed for the purpose of evaluating our techniques. Currently, we are developing new ranking schemes that allow users to build much more customised rankings. For example, a user can build a personalised ranking scheme for a particular set of model features using their domain-specific knowledge. This includes introducing a different set of ranking schemes for classes and associations as well as measuring OCL constructs by taking into account more language features such as nested quantifiers and collection operations. We intend to address these issues and solutions as future work.

11.2 OCL

Though our benchmark covers a wide range of OCL constructs including: multiple arithmetic/logic operators, navigations, nested quantifiers and operations on collection data types, MaxUSE does not support the full range of OCL features such as string data types, closure operators, navigations (using the self keyword) and variable declarations. For example, the syntax of a variable declaration along with

the *context* keyword described in Sect. 8.2. Hence, there is a gap between the full range of OCL language constructs and current version of MaxUSE.

Currently, we are extending MaxUSE to support string data types by directly casting OCL string data types into SMT string theories. Typically, solving complex string constraints is a very challenging task. With the recent advances in string constraint solving techniques, it is now possible to cast complex OCL string operations such as concatenation and substring directly to a string solver [20,43,54]. This would allow us to utilise the latest SMT solving techniques combined with string decision procedures for solving complex OCL string-based constraints in a metamodel.

12 Related work

In this section, we review the literature that is related to our techniques that we present in this paper. Recently, this area has received quite a bit of attention with most of the research focused on determining the consistency of a metamodel/UML class diagram. Of course, finding inconsistencies is more challenging than deciding consistencies, however, it is still necessary to decide the consistencies of a metamodel in the first place as was mentioned in Sect. 9. Our work is mainly concerned with the scenario when a metamodel is not consistent. Therefore, the literature on consistency checking is relevant here. We categorise the set of relevant literature into five areas based on their underlying techniques. These include graph-based approaches, SAT/SMT (constraint solving), constraint programming, Alloy (kodkod) and others.

Graph-based approaches A metamodel or UML class diagram can be considered as a graph so graph-based approaches are naturally employed for reasoning about consistencies of a metamodel [5,29,41,42,57]. Among them, Ehrig et al. propose an instance-generating graph grammar for creating instances of a metamodel. In particular, they use an attributed type graph to capture metamodel structures, and the concept of layered graph grammars to order rule applications. However, this approach cannot handle OCL constraints. Winkelmann et al. present a method for translating a subset of OCL constraints into graph constraints [77]. The OCL constraints in this approach are restricted to equality, size and attribute operations. Others in this domain devise-specific algorithms that determine consistencies of a metamodel. For example, Balaban and Maraee propose a very specialised algorithm called FiniteSat for deciding (finite) satisfiability of class hierarchy and generalisation constraints that are defined over UML class diagrams [5]. The FiniteSat algorithm transforms a class diagram with multiplicity constraints into a linear inequality system. However, this algorithm does not support any OCL constraints.

SAT/SMT With recent advances in constraint solving [22,53,60], well-engineered SAT/SMT solvers have become popular in the verification of metamodels/UML class diagrams. In fact, a significant number of SAT/SMT-based techniques and approaches have emerged, ranging from consistency checking to model synthesis [13,26,71,72,76,78,83]. Büttner et al. [13] and Clavel et al. [24,26] directly map a metamodel and its OCL constraints into first-order logic (FOL) that can be handled by SMT solvers. Büttner et al. use the Z3 SMT solver to verify the correctness of the ATL transformation, while Clavel and Dania use Prover 9 and Z3 to check the satisfiability of OCL constraints. Przigoda et al. encode OCL operational contracts into bit-vectors and use an SMT solver to check concurrent behaviour of a model [59,62]. Each triggered internal system state is represented as a vector and verified via SMT solving. Similarly, Soeken et al. encode the OCL data collection data type into a set of bit-vector-based formulas which can be solved by SMT solvers [69]. Our previous work also focuses on reasoning about metamodel consistencies by generating different types of instances [78,83]. We used bounded typed attributed graphs as our intermediate representation and Z3 for solving constraints over these graphs to generate instances. In this work, we used an unbounded encoding to encode individual metamodel features into FOL and introduce ranked OCL constraints. This allows us to pinpoint conflicting constraints rather than only checking consistencies.

Constraint programming Calvanese represents UML class diagrams using description logic [19], and Cadoli et al. use this idea to implement a technique that can encode a UML class diagram into linear inequalities that can be solved by a constraint programming solver [17,18]. In general, constraint programming allows users to program a problem into a Constraint Satisfaction Problem (CSP). For example, Cabot et al. propose a detailed systematic procedure that automatically translates UML/OCL class diagrams into a CSP [14,16,38,44]. Their approach can check a variety of correctness properties including weak and strong satisfiability by generating a different number of instances for every class. In [15], they extended this approach to OCL operational contracts. The pre/postconditions along with class invariants are programmed into a CSP and solved by a constraint solver. In general, using constraint programming techniques is similar to SAT/SMT. Both techniques require a translation to a set of constraints that can be solved by a constraint solver. However, the main advantage is that CSP provides a high-level language so that a particular constraint problem is programmable while SAT/SMT approaches typically require a relatively low-level encoding (propositional or first-order logic).

Alloy (kodkod) Alloy as a model finder, is a popular tool that receives much attention in many areas including the Model Driven Engineering (MDE) community [45,75]. There has

been much work on using Alloy to test/verify specifications of both semi-formal models and formal specifications [35,58,61]. Since Alloy can be used to generate model instances, research with Alloy has been highly active [4,34,49–51]. Most of this literature uses Alloy as a back-end reasoning engine to check consistencies of a metamodel. Among them, Anastasakis et al. focus on a transformation between UML class diagrams and Alloy's relational specification language [3,4]. In [51], Kuhlmann et al. integrate kodkod (Alloy's reasoning engine) into the USE modeling tool and translate OCL collection data types into Alloy [50]. The main advantage of using Alloy is that it possesses a dedicated algorithm for finding minimal conflicts in the specification [74]. Hence, users are not required to have knowledge about SAT encoding details. However, Alloy currently does not support ranked constraints. Thus, Alloy cannot compute the maximum number of achievable features for a metamodel. Further, Alloy is not guaranteed to find all minimal conflicts. Therefore, approaches using Alloy as a basis for a constraint solving engines are restricted by this functionality [3,34,49,51,56].

Others Other approaches have sought to formalise UML and OCL using different types of formalisms such as interactive theorem provers [1,5,12,25,26]. Berardi et al. formalise a UML class diagram using description logic and show that the complexity of reasoning about a UML class is EXP-Time hard [9]. In fact, reasoning about a UML class diagram with OCL constraints is undecideable since it is equivalent to reasoning about first-order logic sentences [17]. Though their approach formalises features such as classes and different types of associations using description logic, no OCL constraints are covered. Queralt and Embley translate both UML class diagrams with OCL constraints into first-order logic and use a constructive query containment (CQC) method to check the integrity and properties for the given diagram and constraints [31,64]. Other techniques include formalising OCL into higher-order logic. For example, Brucker et al. propose a systematic way of translating OCL into higher-order logic (HOL), and prove correctness using Isabelle. Kyas et al. formalises OCL constraints into PVS [11,52].

Summary In summary, our approach distinguishes from the literature by addressing and solving two particularly challenging problems: (1) computing the set of achievable metamodel features based on their ranks and (2) pinpointing conflicting constraints. By solving these two problems, our approach advances current metamodel consistency checking techniques to another level and we believe that the modelling community can indeed benefit a lot from the techniques presented in this paper. Furthermore, our approach highlights the importance of finding inconsistencies.

13 Conclusions and future work

In this paper, we have presented a formal approach to finding achievable features and constraint conflicts for inconsistent metamodels. Our approach is unique in the sense that we allow users to rank individual metamodel features and find achievable features and constraint conflicts using a state-of-the-art SMT solver, Z3. The reduction (SMT encoding) described here can be used as an add-on to existing SMT-based approaches. Thus, this gives us an advantage of avoiding the tuning of existing SMT encodings. We have implemented our approach into a fully automatic tool called MaxUSE. Our evaluation results show that MaxUSE has promising capabilities for finding both achievable features and conflicting constraints for inconsistent metamodels. These results also show that MaxUSE scales reasonably well on a large number of metamodel features.

During the work described in this paper, we have identified two interesting future research directions. Firstly, the current lack of a proper OCL benchmark makes it difficult for researchers to compare and analyse the capabilities of different OCL analysis/verification tools. In particular, the scalability of each tool. As such, we aim to fill this void by proposing a systematic approach that can automatically generate user-customised OCL benchmarks [81]. Secondly, we plan to extend our approach that we have described in this paper in two directions: (1) To precisely highlight the component of an association that cause inconsistencies. For example, a lower or an upper bound of one association-end. This involves detailed analysis of formulated linear inequalities. (2) To cover OCL operational contracts, the challenge here is that the proposed SMT formulas can quantify unbounded system states triggered by each operation call. Recent work proposes bit-vector-based formulas over unrolled system states [62,71]. Currently, we are investigating a new SMT encoding that provides a high level of expressiveness, flexibility and performance [82].

Acknowledgements This work was partially supported by UK Research and Innovation, and EPSRC Hubs for Robotics and AI in Hazardous Environments: EP/R026092 (FAIR-SPACE).

References

1. Ahrendt, W., Beckert, B., Hähnle, R., Schmitt, P.H.: Key: a formal method for object-oriented systems. In: Bonsangue, M.M., Johnsen, E.B. (eds.) Formal Methods for Open Object-Based Distributed Systems, pp. 32–43. Springer, Berlin (2007)
2. Alon, N., Awerbuch, B., Azar, Y.: The online set cover problem. In: Symposium on Theory of Computing, pp. 100–105. ACM (2003)
3. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: a challenging model transformation. In: International Conference on Model Driven Engineering Languages and Systems, pp. 436–450. Springer (2007)
4. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Softw. Syst. Model. **9**(1), 69–86 (2010)
5. Balaban, M., Maraee, A.: Finite satisfiability of UML class diagrams with constrained class hierarchy. ACM Trans. Softw. Eng. Methodol. **22**(3), 24:1–24:42 (2013)
6. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: International Conference on Computer Aided Verification, pp. 171–177. Springer (2011)
7. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: version 2.0. In: International Workshop on Satisfiability Modulo Theories. Elsevier Science (2010)
8. Becker, J., Rosemann, M., von Uthmann, C.: Guidelines of business process modeling. In: van der Aalst, W., Deselm, J., Oberweis, A. (eds.) Business Process Management, Models, Techniques, and Empirical Studies, pp. 30–49. Springer, Berlin (2000)
9. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams is EXPTIME-hard. In: International Workshop on Description Logics (2003)
10. Berre, D.L., Parrain, A.: The Sat4j library, release 2.2. J. Satisf. Boolean Model. Comput. **7**(2–3), 59–64 (2010)
11. Brucker, A.D., Wolff, B.: HOL-OCL: A formal proof environment for UML/OCL. In: 11th International Conference on Fundamental Approaches to Software Engineering, pp. 97–100. Springer (2008)
12. Brucker, A.D., Wolff, B.: Semantics, calculi, and analysis for object-oriented specifications. Acta Inform. **46**(4), 255–284 (2009)
13. Büttner, F., Egea, M., Cabot, J.: On verifying ATL transformations using 'off-the-shelf' SMT solvers. In: International Conference on Model Driven Engineering Languages and Systems, pp. 432–448. Springer (2012)
14. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: International Conference on Software Testing Verification and Validation Workshop, pp. 73–80. IEEE (2008)
15. Cabot, J., Clarisó, R., Riera, D.: Verifying UML/OCL operation contracts. In: International Conference on Integrated Formal Methods, pp. 40–55. Springer (2009)
16. Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. J. Syst. Softw. **93**, 1–23 (2014)
17. Cadoli, M., Calvanese, D., Giacomo, G., Mancini, T.: Finite model reasoning on UML class diagrams via constraint programming. In: Basili, R., Pazienza, M.T. (eds.) Artificial Intelligence and Human-Oriented Computing, pp. 36–47. Springer, Berlin (2007)
18. Cadoli, M., Calvanese, D., Mancini, T.: Finite satisfiability of UML class diagrams by constraint programming. In: International Workshop on Description Logics (2004)
19. Calvanese, D.: Finite model reasoning in description logics. In: International Conference on the Principles of Knowledge Representation and Reasoning, pp. 292–303. Morgan Kaufmann (1996)
20. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL), 49:1–49:30 (2019)
21. Chvatal, V.: A greedy heuristic for the set-covering problem. Math. Oper. Res. **4**(3), 233–235 (1979)
22. Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R., Stenico, C.: Satisfiability modulo the theory of costs: foundations and applications. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 99–113. Springer (2010)
23. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathSAT5 SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 93–107. Springer (2013)
24. Clavel, M., Egea, M., de Dios, M.A.G.: Checking unsatisfiability for OCL constraints. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **24**, 1–13 (2009)
25. Dania, C., Clavel, M.: Ocl2fol+: coping with undefinedness. In: OCL@MoDELS, pp. 53–62 (2013)
26. Dania, C., Clavel, M.: Ocl2msfol: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of ocl constraints. In: International Conference on Model Driven Engineering Languages and Systems, pp. 65–75. ACM (2016)
27. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer (2008)
28. Een, N., Sörensson, N.: An extensible SAT-solver. In: International Conference on Theory and Applications of Satisfiability Testing, pp. 502–518. Springer (2005)
29. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Softw. Syst. Model. **8**(4), 479–500 (2009)
30. El Ghazi, A.A., Taghdiri, M.: Relational reasoning via SMT solving. In: International Conference on Formal Methods, pp. 133–148. Springer (2011)
31. Farré, C., Teniente, E., Urpí, T.: Checking query containment with the cqc method. Data Knowl. Eng. **53**(2), 163–223 (2005)
32. Feige, U.: A threshold of ln n for approximating set cover. J. ACM **45**(4), 634–652 (1998)
33. Filliâtre, J.C., Paskevich, A.: Why3—where programs meet provers. In: European Symposium on Programming, pp. 125–128. Springer (2013)
34. Garis, A., Cunha, A., Riesco, D.: Translating alloy specifications to UML class diagrams annotated with OCL. In: International Conference on Software Engineering and Formal Methods, pp. 221–236. Springer (2011)
35. Gheyi, R., Massoni, T., Borba, P.: A rigorous approach for proving model refactorings. In: International Conference on Automated Software Engineering, pp. 372–375. ACM (2005)
36. Gogolla, M., Büttner, F., Cabot, J.: Initiating a benchmark for UML and OCL analysis tools. In: International Conference on Tests and Proofs, pp. 115–132. Springer (2013)
37. Gogolla, M., Büttner, F., Richters, M.: USE: a UML-based specification environment for validating UML and OCL. Sci. Comput. Program. **69**(1–3), 27–34 (2007)
38. González Pérez, C.A., Buettner, F., Clarisó, R., Cabot, J.: EMFtoCSP: A tool for the lightweight verification of EMF models. In: International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches, pp. 44–50. IEEE (2012)
39. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: 32nd Conference on Programming Language Design and Implementation. PLDI '11, pp. 62–73. ACM, New York, NY, USA (2011)
40. Harris, W.R., Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program analysis via satisfiability modulo path programs. In: 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '10, pp. 71–82. ACM (2010)

41. Hoffmann, B., Minas, M.: Defining models—meta models versus graph grammars. Electron. Commun. EASST **29**, 1–14 (2010)
42. Hoffmann, B., Minas, M.: Generating instance graphs from class diagrams with adaptive star grammars. In: 3rd International Workshop on Graph Computation Models (2011)
43. Holík, L., Janků, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. Proc. ACM Program. Lang. **2**(POPL), 4:1–43:2 (2017)
44. ILOG: ILOG OPL Studio System Version 3.6.1 User's Manual. IBM (2002)
45. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002)
46. Jangda, A., Yorsh, G.: Unbounded superoptimization. In: Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2017, pp. 78–88. ACM (2017)
47. Jouault, F., Kurtev, I.: Transforming models with ATL. In: The 2005 International Conference on Satellite Events at the MoDELS, pp. 128–138. Springer (2006)
48. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W., Bohlinger, J.D. (eds.) Complexity of Computer Computations, pp. 85–103. Springer, Boston, MA, IBM Thomas J. Watson Research Center, Yorktown Heights, New York (1972)
49. Kuhlmann, M., Gogolla, M.: From uml and ocl to relational logic and back. In: 15th International Conference on Model Driven Engineering Languages and Systems, pp. 415–431. Springer (2012)
50. Kuhlmann, M., Gogolla, M.: Strengthening SAT-based validation of UML/OCL models by representing collections as relations. In: Modelling Foundations and Applications, Lecture Notes in Computer Science, vol. 7349, pp. 32–48. Springer (2012)
51. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive validation of OCL models by integrating SAT solving into USE. In; 49th International Conference on Objects. Models, Components, Patterns, pp. 290–306. Springer, Zurich, Switzerland (2011)
52. Kyas, M., Fecher, H., de Boer, F.S., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., Kugler, H.: Formalizing UML models and OCL constraints in PVS. Electron. Notes Theor. Comput. Sci. **115**, 39–47 (2005)
53. Li, Y., Albarghouthi, A., Kincaid, Z., Gurfinkel, A., Chechik, M.: Symbolic optimization with smt solvers. In: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, pp. 607–618. ACM (2014)
54. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., Deters, M.: An efficient SMT solver for string constraints. Form. Methods Syst. Des. **48**(3), 206–234 (2016)
55. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reason. **40**(1), 1–33 (2008)
56. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: class diagrams analysis using alloy revisited. In: The 14th International Conference on Model Driven Engineering Languages and Systems, pp. 592–607 (2011)
57. Maraee, A., Balaban, M.: Removing Redundancies and Deducing Equivalences in UML Class Diagrams, pp. 235–251. Springer, Berlin (2014)
58. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: a general-purpose higher-order relational constraint solver. In: 37th International Conference on Software Engineering, ICSE '15, pp. 609–619. IEEE Press (2015)
59. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0 system description. J. Satisf. Boolean Model. Comput. **9**, 53–58 (2015)
60. Nieuwenhuis, R., Oliveras, A.: On sat modulo theories and optimization problems. In: Biere, A., Gome, C.P. (eds.) Theory and Applications of Satisfiability Testing, pp. 156–169. Springer, Berlin (2006)
61. Perrouin, G., Sen, S., Klein, J., Baudry, B., Le Traon, Y.: Automated and scalable t-wise test case generation strategies for software product lines. In: The 3rd International Conference on Software Testing, Verification and Validation, pp. 459–468 (2010). doi:https://doi.org/10.1109/ICST.2010.43
62. Przigoda, N., Hilken, C., Wille, R., Peleska, J., Drechsler, R.: Checking concurrent behavior in uml/ocl models. In: 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 176–185 (2015)
63. Przigoda, N., Wille, R., Drechsler, R.: Ground setting properties for an efficient translation of OCL in SMT-based model finding. In: 19th International Conference on Model Driven Engineering Languages and Systems, pp. 261–271. ACM (2016)
64. Queralt, A., Teniente, E.: Reasoning on uml class diagrams with ocl constraints. In: Embley, D.W., Olivé, A., Ram, S. (eds.) Conceptual Modeling, pp. 497–512. Springer, Berlin (2006)
65. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 112–131. Springer International Publishing (2018)
66. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in smt. In: Computer Aided Verification, pp. 198–216. Springer International Publishing, Cham (2015)
67. Reynolds, A., King, T., Kuncak, V.: Solving quantified linear arithmetic by counterexample-guided instantiation. Form. Methods Syst. Des. **51**(3), 500–532 (2017)
68. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '13, pp. 15–26. ACM, New York, NY, USA (2013)
69. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL data types for SAT-based verification of UML/OCL models. In: 5th International Conference on Tests and Proofs, pp. 152–170. Springer, Zurich, Switzerland (2011)
70. Soeken, M., Wille, R., Drechsler, R.: Towards automatic determination of problem bounds for object instantiation in static model verification. In: 8th International Workshop on Model-Driven Engineering, Verification and Validation, pp. 2:1–2:4. ACM, Wellington, New Zealand (2011)
71. Soeken, M., Wille, R., Drechsler, R.: Verifying dynamic aspects of uml models. In: Preas, K. (ed.) Design, Automation Test in Europe, pp. 1–6. IEEE, Grenoble, France (2011)
72. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: Design. Automation Test in Europe Conference Exhibition, pp. 1341–1344. Dresden, Germany (2010)
73. Tillmann, N., De Halleux, J.: Pex: White box test generation for .NET. In: The 2nd International Conference on Tests and Proofs, pp. 134–153 (2008)
74. Torlak, E., Chang, F.S.H., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: The 15th International Symposium on Formal Methods, pp. 326–341. Springer, Turku, Finland (2008)
75. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 632–647. Springer, Braga, Portugal (2007)
76. Wille, R., Soeken, M., Drechsler, R.: Debugging of inconsistent UML/OCL models. In: 2012 Design, Automation Test in Europe Conference Exhibition, pp. 1078–1083 (2012)
77. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. Electron. Notes Theor. Comput. Sci. **211**, 159–170 (2008)

78. Wu, H.: Generating metamodel instances satisfying coverage criteria via SMT solving. In: The 4th International Conference on Model-Driven Engineering and Software Development, pp. 40–51 (2016)
79. Wu, H.: Finding achievable features and constraint conflicts for inconsistent metamodels. In: 13th European Conference on Modelling Foundations and Applications, pp. 179–196. Springer (2017)
80. Wu, H.: Maxuse: a tool for finding achievable constraints and conflicts for inconsistent UML class diagrams. In: Integrated Formal Methods, pp. 348–356. Springer (2017)
81. Wu, H.: Step 0: an idea for automatic OCL benchmark generation. In: Seidl, M., Zschaler, S. (eds.) Software Technologies: Applications and Foundations, pp. 356–364. Springer, Cham (2018)
82. Wu, H.: Synthesising call sequences from OCL operational contracts. In: 34th ACM/SIGAPP Symposium on Applied Computing (2019)
83. Wu, H., Monahan, R., Power, J.F.: Exploiting attributed type graphs to generate metamodel instances using an SMT solver. In: 7th International Symposium on Theoretical Aspects of Software Engineering. Birmingham, UK (2013)
84. Zschaler, S., Kolovos, D.S., Drivalos, N., Paige, R.F., Rashid, A.: The 2nd International Conference on Software Language Engineering, Chap. Domain-Specific Metamodelling Languages for Software Language Engineering, pp. 334–353. Springer, Berlin, Heidelberg (2010)

Hao Wu is an assistant professor in the Department of Computer Science at Maynooth University. He received his Ph.D. in computer science from Maynooth University. His current research aims to create automated software and tools for solving challenging problems in software engineering. In particular, he has strong interests in using Formal Methods (SAT/SMT) to verify different kinds of models used in Model Driven Engineering.

Marie Farrell is a Research Associate in the Department of Computer Science at the University of Manchester working primarily on the EPSRC funded FAIR-SPACE Hub but also participating in the RAIN and ORCA Hubs. She received her Ph.D. from Maynooth University, Ireland in 2017. Her current area of research is in using and combining formal methods to reason about and provide certification evidence for robotic systems that are to be deployed in hazardous environments.