Finding Achievable Features and Constraint Conflicts for Inconsistent Metamodels

Hao Wu

Department of Computer Science,
National University of Ireland, Maynooth
haowu@cs.nuim.ie

Abstract. Determining the consistency of a metamodel is a task of generating a metamodel instance that not only meets structural constraints but also constraints written in Object Constraint Language (OCL). Those constraints can be conflicting, resulting in inconsistencies. When this happens, the existing techniques and tools have no knowledge about which constraints are achievable and which ones cause the conflicts. In this paper, we present an approach to finding achievable metamodel features and constraint conflicts for inconsistent metamodels. This approach allows users to rank individual metamodel features and works by reducing it to a weighted maximum satisfiability modulo theories (MaxSMT). This reduction allows us to utilise SMT solvers to tackle multiple ranked constraints and at the same time locate conflicts among them. We have prototyped this approach, incorporated it into an existing modelling tool, and evaluated it against a benchmark. The preliminary results show that our approach is promising and scalable.

1 Introduction

The metamodelling approach plays a key role in Model-Driven Engineering (MDE), it paves the way for enabling many other MDE approaches such as model transformation, language engineering and business process modelling [1–3]. A metamodel captures the syntax for a set of *models* and allows users to form a design at a higher level of abstraction. A valid model or an *instance* of a metamodel conforms to all of the constraints imposed by its features. These constraints vary according to the metamodel structural features such as multiplicities for an association to class invariants written in Object Constraint Language (OCL). Then the task for checking consistency of a metamodel becomes finding a valid instance. However, this is a challenging task since an instance needs to meet all kinds of constraints defined over that metamodel. Recent studies have shown that this task can be tackled via well-engineered constraint solvers [4–6].

Many metamodels in practice are not consistent due to the conflicts in a number of constraints imposed by different features such as the multiplicities of an association or class invariants. These conflicts could be caused by user errors or features being over-constrained in the design. When this happens, current modelling tools terminate and report inconsistent metamodels, or are unable to generate a valid instance. However, in many cases users may wish to know how many metamodel features can be fulfilled in their current design and which constraints cause the conflicts, then use this information to further refine their metamodels. For example, a user may be interested in finding the minimum number of features that cause conflicts in a metamodel, and fix them in a new design. In other cases, users could use their domain specific knowledge to rank individual features and look for a model that could fulfill as many as features possible.

In this paper, we present an approach to finding two kinds of information when a metamodel is inconsistent. 1) The set of achievable metamodel features based on their rankings. 2) The set of structural constraints or class invariants that cause conflicts. In our approach, both kinds of information are computed 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.

Contributions. The contributions of this paper can be summarised as follows:

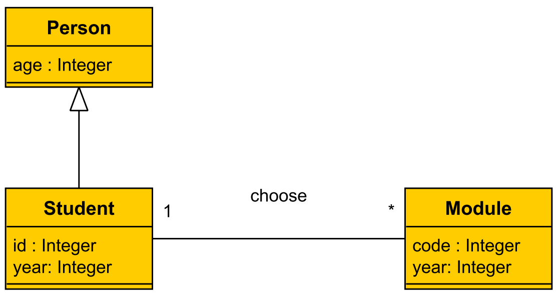
1. We present a simple annotation that allows users to rank individual metamodel features (Section 3.1), and a reduction to weighted MaxSMT problem so that we can compute the set of achievable metamodel features based on their rankings (Section 3.2).
2. Inspired by the work of Liffiton and Sakallah on extracting conflicts [7], we present a novel technique for finding constraint conflicts by solving the set cover problem (Section 3.3).
3. We have implemented a prototype tool, tapped it into an existing modelling tool and evaluated it against a benchmark for scalability (Section 4).

2 A Running Example

In this section, we provide a small example that will be used throughout this paper to illustrate our approach. This example is shown in Figure 1, representing a metamodel that models a real world example of students in a university choosing multiple modules to study. This metamodel is enriched with 8 class invariants (*inv*1 to *inv*8). Each invariant is ranked by using an integer value. For example, each student must have a unique id number (*inv*4), and can only choose modules that are in their year (*inv*5). In this example, we use numbers 1 to 6 to distinguish a student's year, and students that are in year 6 are considered as research students. Thus, a university has some non-research and research students (*inv*6).

This metamodel is inconsistent and has a maximum number of 6 achievable invariants. This is due to the two conflicts among the invariants in Figure 1. The first conflict is obvious and it is caused by the invariants *inv*1 and *inv*2 defined for the *age* attribute. However, the second conflict is not easy to identify. This conflict is caused by the invariants that there must exist some research and non-research students ($inv6$) choosing some modules ($inv7$) in their corresponding year ($inv5$). But there are modules that are only available for non-research students ($inv8$: between year 1 and 5).

However, in the real world each individual invariants may be treated differently based on user's domain specific knowledge. For example, a university may consider a registration procedure that students choosing modules in their corresponding year ($inv5$) is more important than choosing some modules ($inv7$). In this context, a maximum number of 6 invariants is achievable with the preference that $inv5$ is more favourable than $inv7$. Therefore, by allowing more favourable constraints to be achieved first is more suitable for users wishing to distinguish priorities among different invariants.



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

@StudentRank{Rank = automatic}
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)
@Rank = 5
inv7: Student.allInstances()->forAll(s|s.modules->notEmpty())

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

Fig. 1. An example of a ranked metamodel describing how a student can choose multiple modules to study. The ranks are highlighted in the shaded area. Our approach concludes that this metamodel has a maximum number of 6 achievable invariants and 2 conflicts:$(inv1, inv2)$ and $(inv5, inv6, inv7, inv8)$.

3 The Approach

Figure 2 provides an overview workflow of our approach. Briefly, this is viewed as three steps. First, users use a simple annotation to rank individual metamodel features. The approach then determines the consistency of a metamodel. If there is at least one class that cannot be instantiated, then all metamodel features along with OCL constraints will be reduced to a weighted MaxSMT problem and solved by an SMT solver. The returned solution is a set that contains all possible ways of maximising the number of achievable metamodel features based on their rankings. Finally, to find constraint conflicts among all metamodel features, the approach treats all features equally including OCL constraints, formalises them into the set cover problem and solves it by using an SMT solver.

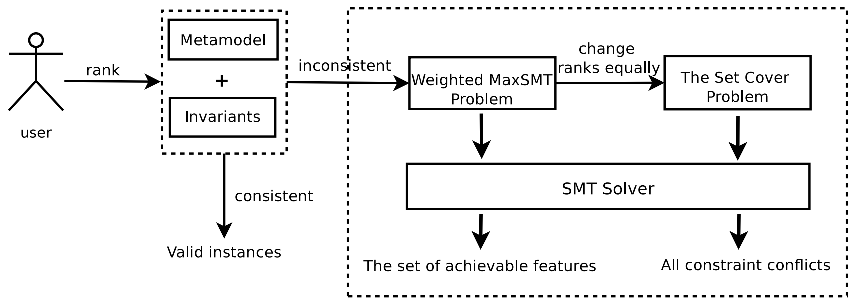


Fig. 2. An overview of our approach.

3.1 Annotation

We provide a simple annotation for users to specify a rank on individual metamodel features. This annotation has the basic form:'$@Rank = c$', where $c \in \mathbb{Z}^+$ denoting a metamodel feature is ranked via an non-negative integer c. Currently, we allow users to rank classes, associations and invariants. If a metamodel has a conflict, then any ranked features cause that conflict might be switched off during the search for the achievable features. We consider all metamodel features ranked with integer c as *soft features*. A soft feature with higher ranking is more favourable to be selected than a feature with lower ranking during the search. For example, $inv5$ in Figure 1 is more likely to be chosen compared to $inv7$. On the other hand, if a feature is *not* ranked, then it is a *hard feature* that must not be ignored during the search. For example, $inv8$ in Figure 1 must hold, no matter what. Therefore, a user could specify a set of soft and hard features over a metamodel by using this annotation.

Sometime users wish to use a single ranking criteria to treat a group of class invariants. For example, all invariants defined for a specific class are equally important. In this case, another type annotation:'$@Name\{Rank = c\}$' is introduced, where $Name$ is an identifier for the annotation, and $c \in \mathbb{Z}^+$. For example, in Figure 1 the annotation '$@StudentRank$' specifies that every invariant defined under the class *Student* is ranked using automatic ranking. However, users may override current ranking criteria by specifying a different rank through '$@Rank = c$'. For example, an automatic ranking is initially specified for $inv5$ and $inv7$ but it is overwritten by the new values of 6 and 5. The remaining invariants are ranked using automatic ranking.

Ranking Criteria. A metamodel feature can be ranked in two ways: (1) Users rank an individual metamodel feature into a soft feature based on their domain specific knowledge[1]. (2) In situations, where users feel they can let the program automatically handle a particular feature for them, an automatic ranking criteria is provided. In default settings, all metamodel features are initially treated as hard features. However, users may override default settings by using the keyword 'automatic'. All features annotated with 'automatic' are assigned a specific value internally and automatically calculated as follows.

Automatic Ranking. The automatic ranking for a class is calculated based on the number of attributes and operations (including those inherited from an abstract class) defined within. This is because a class that contains more attributes and operations typically describes more information about a system than a class with fewer attributes and operations. For an association, it is calculated by adding up the rank defined on each association end[2]. For a class invariant, we calculate the size of its abstract syntax tree (AST). The larger size of an invariant's AST, the more likely a stricter constraint will be imposed on a metamodel[3]. For example, the invariants (except for *inv*2 and *inv*7) defined for *Student* class are automatically assigned with a rank based on the size of their ASTs. In Figure 3, we can see that *inv*3 is assigned with a value of 9.

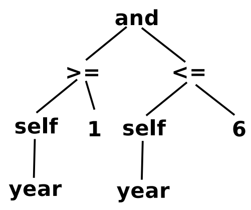


Fig. 3. The abstract syntax tree for *inv*3 from Figure 1 has a total of 9 nodes.

3.2 Reducing to Weighted MaxSMT

Our reduction to SMT is a procedure that traverses a set of soft features defined on a metamodel and automatically generates a set of SMT formulas. Each generated formula consists of two parts.

The first part is an SMT encoding of a specific metamodel feature[4]. Currently, the formula in this part is similar to the encoding of metamodel features into first-order logic (FOL) [8]. We support an encoding of a variety of metamodel features

[1] Note that a metamodel could be ranked in 3 different scenarios: (1) Partially ranked (a mixture of soft and hard features). (2) Totally ranked (soft features only). (3) Not ranked (hard features only).

[2] Currently, we assume that each association end is owned by a class.

[3] An invariant could be written in multiple ways. Here, we assume all class invariants were written in a consistent way. For example, using $self$ to constrain attributes and $allInstances()$ for quantifiers and navigations.

[4] Note that for this part a user could still use an existing SMT encoding, no changes are required.

such as classes, inheritance, associations, and class invariants. For invariants written in OCL, we also support navigation, nested quantifiers and operations on generic collection data types such as *include*. These encodings are similar to those used in [9]. Currently, we do not support *string* operations.

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 on a specific metamodel feature when the constraint imposed by that feature is achievable.

Given a total k number of soft features, we let F_i be an SMT formula that encodes the *ith* soft feature in a metamodel. We now introduce an integer type auxiliary variable Aux_i whose range is $\{0, 1\}$. We then generate Formula 1. The idea behind this formula 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* in Formula 1 since both parts can not be satisfiable simultaneously. Therefore, we can check whether a feature encoded by formula F_i is achievable via testing the satisfiability of Formula 1.

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

Now 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 2. The implication of this formula is built on Formula 1. If Formula 1 is satisfiable, then each $Aux_i = 0$ and F_i must also be satisfiable. This means that the constraint imposed by *ith* soft feature can be achieved. Thus, we must assign an integer constant c 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 assigning 0 to V^{W_i}.

$$\bigwedge_{i=1}^{k} \left(\Big((Aux_i = 0) \Rightarrow (V^{W_i} = c)\Big) \wedge \Big((Aux_i = 1) \Rightarrow (V^{W_i} = 0)\Big)\right), \text{where } c > 0. \tag{2}$$

Finally, we form a weighted MaxSMT problem by generating Formula 3. We generate this formula only when Formula 1 is not satisfiable. This is because if Formula 1 is satisfiable, then a metamodel is consistent. Intuitively, we know some F_is are not satisfiable, and both *part a* and *part b* from Formula 1 cannot be satisfiable at the same time. Now to make Formula 1 become satisfiable, we remove *part b* (Formula 1) and rewrite it as *part c* (Formula 3). This forces some of the auxiliary variables (Aux_i in Formula 1) 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. Thus, in this way we can work out m number of constraints imposed by metamodel features that cannot be fulfilled. In the meantime, we also check whether it is possible to achieve a total of rank of c based on the remaining number of metamodel features (*part d* in Formula 3). If c is the maximum number we can find to make Formula 3 satisfiable, then c is a solution to our weighted MaxSMT problem.

$$\underbrace{\Big(\Big(\sum_{i=1}^{k} Aux_i\Big) = m\Big)}_{part\ c} \wedge \underbrace{\Big(\Big(\sum_{i=1}^{k} V^{W_i}\Big) = c\Big)}_{part\ d}, \text{where } 1 \leq m \leq k, 1 \leq c \leq \sum_{i=1}^{k} W_i. \tag{3}$$

Now that we have formed weighted MaxSMT problem from ranked metamodel, the goal here is to find a maximum total rank from all ranked metamodel features, namely weighted MaxSMT solution. To reduce the number of satisfiability checks, we employ a binary-search based algorithm to search for this maximum total rank. This algorithm iteratively asks an SMT solver to solve Formula 3 and look for an integer that could maximise the rank. If the maximum rank is found, the algorithm then enumerates all possible ways of achieving this value by blocking all previous successful assignments until no more weighted MaxSMT solutions can be found. Note that if a metamodel contains hard features only, then the algorithm returns a maximum number of achievable metamodel features.

3.3 Finding Constraint Conflicts

In [7], the authors reveal that the set of conflicts among SAT formulas can be captured by the set cover problem[5]. Inspired by their work, we directly use this information to 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 essentially is a *minimal unsat core*. This core is a set of unsatisifiable SMT formulas and all *proper* subsets of the core are satisfiable. Though only few of the SMT solvers provide unsat core extraction, such extraction is not guaranteed to find *all minimal* unsat cores [10]. For example, the Z3 SMT solver only finds one conflict ($inv1$, $inv2$) for the example in Figure 1.

Relationship to the Set Cover Problem. Formally, a set cover problem can be defined as: 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 is minimum if it uses fewest I_is to cover U and such collection is called *a minimum set*.

To illustrate that the set cover problem captures the conflicts among the set of metamodel features. We use the example from Figure 1 except that we treat all invariants ($inv1$ to $inv8$) *equally* this time and solve them to derive a total of 8 different solutions ($S_1, S_2, ..., S_8$), as shown in Figure 4. Each solution describes a way of maximising the number of class invariants in Figure 1, namely they are MaxSMT solutions. A matrix is then formed with each row describing one solution and each column denoting a class invariant from Figure 1. For example, in Figure 4 row $S_1 = \{I_2, I_5\}$ denotes a way of achieving 6 numbers of invariants by deactivating 2 invariants $inv2$ and $inv5$ (in Figure 1). In the first row, we use

[5] The hitting set problem is an instance of the set cover problem.

MaxSMT Solutions	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
$S_1 = \{I_2, I_5\}$	0	**1**	0	0	**1**	0	0	0
$S_2 = \{I_2, I_6\}$	0	**1**	0	0	0	**1**	0	0
$S_3 = \{I_2, I_7\}$	0	**1**	0	0	0	0	**1**	0
$S_4 = \{I_2, I_8\}$	0	**1**	0	0	0	0	0	**1**
$S_5 = \{I_1, I_5\}$	**1**	0	0	0	**1**	0	0	0
$S_6 = \{I_1, I_6\}$	**1**	0	0	0	0	**1**	0	0
$S_7 = \{I_1, I_7\}$	**1**	0	0	0	0	0	**1**	0
$S_8 = \{I_1, I_8\}$	**1**	0	0	0	0	0	0	**1**

Fig. 4. An example that illustrates how the set cover problem captures the conflicts among metamodel features. For example, a conflict between *inv*1 (I_1) and *inv*2 (I_2) in Figure 1 can be identified here, since I_1 covers $\{S_5, S_6, S_7, S_8\}$ and I_2 covers $\{S_1, S_2, S_3, S_4\}$.

a 1 to mark these two invariants that can *not* be achieved, and 0 to mark the remaining invariants that can be achieved.

To find conflicts among these invariants, consider each column $I_i \subseteq \{S_1, ..., S_8\}$ that covers only rows marked with a 1 in that column. We say an S_i is covered if and only if at least one of the elements is covered. For example, column I_1 covers row S_5, S_6, S_7, and S_8, while column I_3 covers no rows. A conflict can now be identified by finding a sub collection (set) of I_is such that the union of I_is covers all rows (S_1 to S_8). 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, set $\{I_1, I_2\}$ forms a minimal unsat core and thus *inv*1 and *inv*2 (from Figure 1) conflict with each other. Another conflict can be identified by forming the set $\{I_5, I_6, I_7, I_8\}$.

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 [11]. To tackle this problem, we present a novel technique that allows us to find all metamodel constraint conflicts via SMT solving. This technique first computes a set of achievable metamodel features (MaxSMT solutions) and formulates an $m \times n$ matrix M similar to the one in Figure 4. Then it automatically generates a set of SMT formulas capturing the set cover problem and uses an SMT solver to find metamodel constraint conflicts.

The core idea of this technique is to formalise 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 this $m \times n$ matrix M in Figure 5:

- each entry $a_{ij} \in \{0, 1\}$ is an element from a set (S_i and 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 *can not* be achieved.
- each I_j denotes a subset of S_is in the *jth* column, depending on whether $a_{ij} = 1$.

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

Fig. 5. A matrix M representing the set cover problem.

Let 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 are $\{0, 1\}$. We now 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.

We first generate Formula 4 stating that S_i is selected (covered) if one of the a_{ij}s in *ith* row is selected. Otherwise if all a_{ij}s (in *ith* row) are not chosen, then S_i can not be covered. For example, in Figure 4, we say S_1 can be covered by either the entry in the 1st row and 2nd column (a_{12}) or another entry in the 1st row and 5th column (a_{15}), as both of them are set to 1 ($S_1 = \{a_{12}, a_{15}\}$).

$$\bigwedge_{i=1}^{m} \left(\left(\Big(\bigvee_{\substack{j=1 \\ a_{ij} \in S_i}}^{n} V^{a_{ij}} = 1 \Big) \Rightarrow \Big(V^{S_i} = 1 \Big) \right) \wedge \left(\Big(\bigwedge_{\substack{j=1 \\ a_{ij} \in S_i}}^{n} V^{a_{ij}} = 0 \Big) \Rightarrow \Big(V^{S_i} = 0 \Big) \right) \right) \tag{4}$$

Intuitively, Formula 5 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 the elements of it should be selected. This condition is enforced by using a conjunction to connect all elements in I_j to make sure that none of its elements are selected. For example, if subset I_5 in Figure 4 is not chosen, then its two elements at the 5*th* column, marked as 1 (a_{15} and a_{55}) are also not selected ($I_5 = \{a_{15}, a_{55}\}$).

$$\bigwedge_{j=1}^{n} \left(\left(\Big(V^{I_j} = 1 \Big) \Rightarrow \Big(\bigwedge_{\substack{i=1 \\ a_{ij} \in I_j}}^{m} V^{a_{ij}} = 1 \Big) \right) \wedge \left(\Big(V^{I_j} = 0 \Big) \Rightarrow \Big(\bigwedge_{\substack{i=1 \\ a_{ij} \in I_j}}^{m} V^{a_{ij}} = 0 \Big) \right) \right) \tag{5}$$

Finally, we generate an integer equality shown in Formula 6 describing 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 the algorithm in Figure 6 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).

$$\Bigg(\underbrace{\Big(\sum_{i=1}^{m} V^{S_i}\Big) = m}_{part\ a}\Bigg) \wedge \Bigg(\underbrace{\Big(\sum_{j=1}^{n} V^{I_j}\Big) = k}_{part\ b}\Bigg), \text{ where } 1 \leq k \leq n. \tag{6}$$

```
   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 $Solver.add(Formula\ 4 \wedge Formula\ 5 \wedge Formula\ 6[part\ a])$
 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 $s$
```

Fig. 6. An algorithm that 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.

4 Implementation and Evaluation

We have prototyped the approach described in Section 3 into a tool called MaxUSE[6] and incorporated it into the exisiting USE modelling tool [12]. We choose 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 reads in a metamodel that is fully or partially ranked. It traverses a metamodel and automatically generates a set of SMT2 formulas [13]. MaxUSE currently uses Z3 as its solving engine [10]. It incrementally solves these formulas and interprets each successful assignment as a solution. Our implementation is approximately 7000 lines of Java code.

[6] available at https://github.com/classicwuhao/maxuse

4.1 Evaluation

Forming Benchmark. To extensively evaluate MaxUSE's capability, we first collect a group of metamodels (Group A in Table 1) from [14], and use them as candidate metamodels. For each candidate metamodel, we calculate a configuration in terms of its number of classes, associations (different multiplicities), invariants, conflicts, navigations, quantifiers, logic/arithmetic operators, and breadth/depth of inheritance trees. We then develop a generator that can generate USE specifications based on different sized configurations. This generator is approximately 2100 lines of Java code. We use this generator to generate another four groups (Group B, C, D and E in Table 1) of metamodels using the configurations calculated from the metamodels in Group A. Currently, MaxUSE supports OCL constructs used in these metamodels.[7]. For each group, we generate 5 metamodels ranging from small to large size. Finally, we randomly inject a number of conflicts into each metamodel and gather them as a benchmark as shown in Table 1. For example, for Group D we use a configuration that allows us to specify the number of diamond shapes and OCL constraints over an inheritance tree. This is because we use the DS metamodel from Group A as a candidate and this metamodel contains an OCL constraint over a diamond shaped inheritance tree. Therefore, every metamodel in Group D also has a number of constraints over this property based on its size.
Performance Evaluation. We evaluate MaxUSE on an Intel(R) Xeon(R) machine with eight 3.2GHz cores. However, our current implementation uses only one core. Table 1 records MaxUSE's performance against different sized metamodels. For each group in Table 1, we first randomly rank each metamodel including the use of automatic rankings and run MaxUSE to find one solution. We then equally rank each metamodel and ask MaxUSE to find all possible solutions including conflicts. This is because an equally ranked metamodel more likely to have multiple solutions. All the performances are recorded in the 'Single' and 'All' columns in Table 1. We observe that MaxUSE takes less than *one second* to determine whether a metamodel is consistent or not, and find the maximum weight and conflicts within a reasonable amount of time in most cases. The longest time taken by MaxUSE is approximately 8 hours to get 174 solutions for the D5 metamodel. In general, MaxUSE finds all conflicts much faster than finding all weighted MaxSMT solutions. This is because searching for an optimal solution requires significant computation by Z3. Once all solutions are found, MaxUSE can utilise them to solve the set cover problem much faster. In some cases, MaxUSE could not find solutions. This is mainly due to Z3 spending a significant amount of time on solving a large number of formulas combining nested quantifiers and inequalities. For example, for the E5 metamodel, Z3 was stuck with a particular value and could not progress to next possible optimal value within 9 hours. In general, it is an extremely challenging task for any algorithms 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

[7] MaxUSE cannot handle the OAI metamodel due to recursive structures. Instead, we add the SM metamodel into Group A (similar to Figure 1 in Section 2).

		Number of				Rank		Single (sec) Mix Ranked		All (#/sec) Eq Ranked	
		Classes	**Assocs**	**Invs**	**Formulas**	**Max**	**Total**	**Solution**	**Conflict**	**Solution(s)**	**Conflict(s)**
Group A	CS	3	1	6	19	15	15	NA	NA	NA	NA
	WR	2	2	7	33	36	40	4.01	0.16	1/3.53	1/0.5
	DS	4	0	1	16	14	16	0.14	0.19	2/0.14	1/0.89
	OAI†	1	1	7	NA	NA	NA	NA	NA	NA	NA
	SM	3	1	8	31	79	91	0.33	0.08	8/0.44	2/0.15
Group B	B1	13	5	27	169	490	498	3.97	0.43	2/2.22	2/0.44
	B2	24	9	45	285	521	556	15.90	0.16	12/22.53	5/0.83
	B3	33	14	68	420	792	821	31.49	0.58	6/54.89	5/11.92
	B4	46	15	90	539	620	622	67.37	0.22	2/100.87	2/1.42
	B5	57	19	136	729	881	894	560.12	1.45	24/1609.12	6/3.48
Group C	C1	13	5	29	171	237	268	5.59	0.05	12/18.49	7/0.59
	C2	24	11	43	276	470	478	15.76	0.83	4/14.70	2/0.84
	C3	35	17	66	418	570	581	59.46	0.07	1/68.79	2/1.12
	C4	46	15	98	549	605	630	342.98	1.50	4/226.05	4/1.66
	C5	57	15	156	765	1004	1045	2853.65	0.57	72/5467.75	11/5.49
Group D	D1	13	2	22	136	171	189	3.03	0.17	1/4.17	6/0.46
	D2	26	9	47	294	259	329	17.56	0.23	1/23.09	13/0.91
	D3	33	3	61	329	520	596	21.74	0.42	6/34.74	9/0.98
	D4	46	9	101	525	452	651	68.17	0.39	3/90.08	34/1.24
	D5	56	18	166	805	1089	1291	15904.21	2.33	174/29368.16	46/19.22
Group E	E1	10	6	31	162	69	72	6.26	0.07	1/7.38	1/0.32
	E2	15	12	39	224	217	233	83.36	0.08	2/66.48	4/0.55
	E3	30	18	37	312	238	243	392.22	0.729	1/47.75	1/0.71
	E4	18	18	105	511	483	515	405.51	0.68	7/5959.61	20/3.90
	E5*	18	18	167	698	NA	415	NA	NA	NA	NA

Table 1. The benchmark for evaluating MaxUSE. 'Formulas' denotes the number of SMT2 formulas generated. 'Rank' denotes the achieved maximum rank ('Max') out of a total rank distributed ('Total'). 'Single' and 'All' denote the time (in seconds) spent by MaxUSE on finding a single and all possible solutions respectively. '#/sec' means that the number of solutions and seconds used. '†' indicates that MaxUSE determines that a metamodel is consistent. '*' denotes that MaxUSE cannot find solutions within 9 hours.

a massive search space.

Quality of Computed Constraint Conflicts. For the conflicts found in these metamodels from the benchmark in Table 1, we compare them against actual injected conflicts to assess how accurate they are. The injected conflicts covers a wide range of different metamodel features including multiplicities on association ends, different type of attributes and inheritance relationships among multiple classes. We classify our comparison results as either "exact", "near" or "miss" and record them into Table 2. Here,"exact" means that MaxUSE finds conflicts that match exactly with injected conflicts. In other words, each one (set) is minimal and removal of any members can make a metamodel become consistent. "near" means that MaxUSE is able to identify all conflicts that are close enough to the injected ones. We consider they are "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 returned conflicts. Thus, users could easily understand this information and use this in a latter stage for debugging or fixing conflicts. "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 heuristic algorithms used internally in Z3. Despite this inaccuracy, we believe that the results here show the potential of our approach to finding constraint conflicts for inconsistent metamodels.

Group A		Group B		Group C		Group D		Group E	
CS	NA	B1	exact	C1	near	D1	exact	E1	near
WR	exact	B2	near	C2	exact	D2	near	E2	near
DS	exact	B3	exact	C3	near	D3	exact	E3	exact
OAI	NA	B4	near	C4	near	D4	near	E4	miss
SM	exact	B5	exact	C5	near	D5	near	E5	NA

Table 2. Quality of computed constraint conflicts for metamodels in Table 1.

Lessons Learnt. From the evaluation results, we have learned three important lessons:

1. MaxUSE can maximise the number of achievable features and pinpoint conflicting constraints without the need for manual interactions. However, in some cases when Z3 is unable to handle formulas, an interactive mode is necessary. For example, when Z3 could not solve formulas generated for the $E5$ metamodel within a specified time frame, we pause MaxUSE and manually choose a possible optimal value. MaxUSE is then able to resume the search. However, selecting such a value is quite tricky and requires that one has knowledge about how things work inside the solver.
2. In terms of scalability, the number of ranked features is proportional to the solving time of MaxUSE. Additionally, we suggest that one could gain better performance by ranking individual metamodel features into hard features or using a set of relatively smaller ranks. For example, if a metamodel has 100 features one may consider to rank them using a range of integers from 1 to 100 rather than choosing from 101 to 200.

3. Computing all constraint conflicts sometimes can be significantly more expensive than finding one conflict since there could be an exponential number of them. In this case, we find it is necessary to let users decide when to stop MaxUSE for enumerating all constraint conflicts. This is because some constraint conflicts are not independent. Therefore, the dependent conflicts can be used to identify other conflicts without exhaustive enumeration. In the future, we plan to address this issue and enhance our algorithm for finding constraint conflicts.

4.2 Threats to Validity

The major threat to external validity concerns the benchmark we form in Table 1. This benchmark is based on the metamodels collected from [14]. Since these metamodels do not cover the full set of OCL constructs, this introduces a gap between our implementation and full OCL constructs. We acknowledge that the evaluation results of this benchmark only give us a preliminary assessment of MaxUSE. In the future, we plan to cover more OCL constructs including operational constraints and string operators.

The most significant threat to internal validity concerns the performance of MaxUSE which is mainly dependent on the Z3 SMT solver. In some cases, the first run of Z3 fails to find solutions. However, further runs typically resolve this issue. This introduces an additional performance overhead. We surmise that this is caused by the heuristic algorithms used in Z3. In the future, we plan to overcome this by plugging in multiple SMT solvers and allow users to switch among them for the best performance.

5 Related Work

The majority of the research in metamodel/UML class diagram-based reasoning/verification concentrates on answering the question [9, 15, 6, 16, 5, 17, 18]: whether a metamodel is consistent or not. We focus on the situation when a metamodel is not consistent, then what information we should give back to users to help them refine their metamodels. We believe that providing the maximum number of achievable features and finding constraint conflicts among them is useful for users to further refine their metamodels. Moreover, this paper also demonstrates the feasibility and scalability of tackling *ranked* metamodel features in an existing modelling environment by introducing SMT solving.

Much research work has sought to formalise metamodels or UML class diagrams into different types of logics [19–22, 9, 8, 23–30]. With recent advances in constraint solving, SAT/SMT solvers have been widely adapted to verifying metamodel/UML class diagrams. Among them, Büttner et al [8] and Clavel et al [9, 28] directly map a metamodel and its OCL constraints into first-order logic 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. We use a similar idea to encode the metamodel and OCL constraints, but differ by solving ranked OCL constraints and the set cover problem. By introducing ranked features to a metamodel and solving the set cover problem, users are able to maximise the number of features based on their domain specific knowledge and find constraint conflicts.

Cabot et al. propose a detailed systematic procedure that uses constraint programming to program UML/OCL class diagrams into a Constraint Satisfaction Problem (CSP) [31, 32, 16]. The main advantage is that CSP provides a high-level language so that a particular constraint problem is programmable. Their approach can check a variety of correctness properties including weak and strong satisfiability by generating a different number of instances for every class. Instead of presenting an encoding of metamodel and OCL constraints, our work focuses on reducing a set of ranked metamodel features to a weighted MaxSMT problem and finding a maximum number of achievable features and conflicts at the same time. Further, our approach presented in this paper can be easily incorporated into existing SAT/SMT based approaches without tuning original encodings.

Alloy uses first-order relational logic as its specification language to model the problem domain and reduce it to SAT instances [33–35]. It directly supports finding minimal conflicts in the specification [36]. However, this functionality is not guaranteed to find all minimal conflicts. Therefore, approaches using Alloy as a basis for constraint solving engine are also restricted by this functionality.[37, 4, 38–40]. Further, Alloy's engine is limited to unranked constraints so users are not able to rank individual constraints, whereas our approach focuses on maximising all ranked features.

6 Conclusion

In this paper, we have presented an 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 by using a state-of-the-art SMT solver. Further, our SMT encodings presented in this paper could 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. To demonstrate feasibility and scalability, we have implemented this approach into a prototype tool and evaluated it against a benchmark. Our evaluation results suggest that the approach is promising and scales reasonably well on a large number of metamodel features. In the future, we plan to extend this approach to metamodel transformation verification and develop a technique that is able to guide users step-by-step in refining/synthesizing transformation rules based on their specified preferences.

References

1. Jouault, F., Kurtev, I.: Transforming models with ATL. In: The 2005 International Conference on Satellite Events at the MoDELS, Springer (2006) 128–138
2. Zschaler, S., Kolovos, D.S., Drivalos, N., Paige, R.F., Rashid, A.: Domain-Specific Metamodelling Languages for Software Language Engineering. In: The 2nd SLE. Springer (2010) 334–353
3. Becker, J., Rosemann, M., Uthmann, C.v.: Guidelines of business process modeling. In: Business Process Management, Models, Techniques, and Empirical Studies, Springer (2000) 30–49
4. 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, Zurich, Switzerland, Springer (2011) 290–306
5. Wille, R., Soeken, M., Drechsler, R.: Debugging of inconsistent UML/OCL models. In: 2012 DATE. (2012) 1078–1083
6. Wu, H., Monahan, R., Power, J.F.: Exploiting attributed type graphs to generate metamodel instances using an SMT solver. In: 7th TASE, Birmingham, UK (2013)
7. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reason. **40**(1) (Janurary 2008) 1–33
8. Büttner, F., Egea, M., Cabot, J.: On verifying ATL transformations using 'off-the-shelf' SMT solvers. In: 15th MoDELS. (2012) 432–448
9. Clavel, M., Egea, M., de Dios, M.A.G.: Checking unsatisfiability for OCL constraints. Electronic Communication of the European Association of Software Science and Technology **24** (2009)
10. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: 14th TACAS, Budapest, Hungary, Springer (2008) 337–340
11. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations. (1972) 85–103
12. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Science of Computer Programming **69**(1-3) (2007) 27–34
13. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: 8th International Workshop on Satisfiability Modulo Theories, Edinburgh, UK, Elsevier Science (2010)
14. Gogolla, M., Büttner, F., Cabot, J. In: Initiating a Benchmark for UML and OCL Analysis Tools. Springer (2013) 115–132
15. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL data types for SAT-based verification of UML/OCL models. In: 5th TAP, Springer (2011) 152–170
16. Cabot, J., Clarisó, R., Riera, D.: On the verification of UML/OCL class diagrams using constraint programming. Journal of Systems and Software **93** (2014) 1–23
17. Balaban, M., Maraee, A.: Finite Satisfiability of UML Class Diagrams with Constrained Class Hierarchy. ACM Transaction on Software Engineering and Methodology **22**(3) (2013) 24:1–24:42
18. Wu, H.: Generating metamodel instances satisfying coverage criteria via SMT solving. In: The 4th MODELSWARD. (2016) 40–51
19. Beckert, B., Keller, U., Schmitt, P.H.: Translating the Object Constraint Language into first-order predicate logic. In: Verify Workshop at FLoC, Copenhagen, Denmark (2002)
20. Maraee, A., Balaban, M.: Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In: 3rd ECMDA, Springer (2007) 17–31

21. Brucker, A.D., Wolff, B.: HOL-OCL – A Formal Proof Environment for UML/OCL. In: The 11th FASE, Springer (2008) 97–100
22. 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}. Electronic Notes in Theoretical Computer Science **115** (2005) 39–47
23. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using boolean satisfiability. In: DATE (2010) 1341–1344
24. Queralt, A., Artale, A., Calvanese, D., Teniente, E.: Ocl-lite: Finite reasoning on UML/OCL conceptual schemas. Data & Knowledge Engineering **73** (2012) 1 – 22
25. Dania, C., Clavel, M.: Ocl2fol+: Coping with undefinedness. In: OCL@MoDELS. (2013)
26. Semeráth, O., Vörös, A., Varró, D.: Iterative and incremental model generation by logic solvers. In: 19th FASE, Springer (2016) 87–103
27. Przigoda, N., Wille, R., Drechsler, R.: Ground setting properties for an efficient translation of OCL in SMT-based model finding. In: 19th MoDELS, ACM (2016) 261–271
28. Dania, C., Clavel, M.: Ocl2msfol: A mapping to many-sorted first-order logic for efficiently checking the satisfiability of ocl constraints. In: 19th MoDELS, ACM (2016) 65–75
29. Wu, H., Monahan, R., Power, J.F.: Metamodel instance generation: A systematic literature review. CoRR **abs/1211.6322** (2012)
30. Wu, H.: An SMT-based approach for generating coverage oriented metamodel instances. International Journal of Information System Modeling and Design **7 (3)** (2016)
31. González Pérez, C.A., Buettner, F., Clarisó, R., Cabot, J.: EMFtoCSP: A tool for the lightweight verification of EMF models. In: Formal Methods in Software Engineering: Rigorous and Agile Approaches, Zurich, Suisse (2012)
32. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: IEEE ICST V&V Workshop, Berlin, Germany, IEEE Computer Society (2008) 73–80
33. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering Methodologies **11**(2) (2002) 256–290
34. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: 13th TACAS, Braga, Portugal, Springer (2007) 632–647
35. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: A general-purpose higher-order relational constraint solver. In: 37th ICSE, IEEE Press (2015)
36. Torlak, E., Chang, F.S.H., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: The 15th International Symposium on Formal Methods, Turku, Finland, Springer (2008) 326–341
37. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: ACM/IEEE 10th MoDELS, Nashville, TN, Springer (2007) 436–450
38. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class diagrams analysis using alloy revisited. In: The 14th MoDELS. (2011) 592–607
39. Garis, A., Cunha, A., Riesco, D.: Translating Alloy Specifications to UML Class Diagrams Annotated with OCL. In: 9th SEFM, Montevideo, Uruguay, Springer (2011) 221–236
40. Kuhlmann, M., Gogolla, M.: From UML and OCL to relational logic and back. In: 15th MoDELS, Springer (2012) 415–431